Lecture Notes in Computer Science 1848
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Raffaele Giancarlo David Sankoff (Eds.)

Combinatorial Pattern Matching

11th Annual Symposium, CPM 2000
Montreal, Canada, June 21-23, 2000
Proceedings

Springer

Volume Editors

Raffaele Giancarlo
Università die Palermo
Dipartimento di Matematica ed Applicazioni
Via Archirafi 34, 90123 Palermo, Italy
E-mail: raffaele@altair.math.unipa.it

David Sankoff
Université de Montréal
Centre de recherches mathématiques
CP 6128 succursale Centre-Ville
Montréal, Québec, Canada H3C 3J7
E-mail: sankoff@ere.umontreal.ca

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Combinatorial pattern matching : 11th annual symposium ; proceedings /
CPM 2000, Montréal, Canada, June 21 - 23, 2000. Raffaele Giancarlo ;
David Sankoff (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ;
Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2000
(Lecture notes in computer science ; Vol. 1848)
ISBN 3-540-67633-3

CR Subject Classification (1998): F.2.2, I.5.4, I.5.0, I.7.3, H.3.3, E.4, G.2.1

ISSN 0302-9743
ISBN 3-540-67633-3 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Christian Grosche, Hamburg
Printed on acid-free paper SPIN: 10722094 06/3142 5 4 3 2 1 0

Foreword

The papers contained in this volume were presented at the 11th Annual Symposium on Combinatorial Pattern Matching, held June 21-23, 2000 at the *Université de Montréal*. They were selected from 44 abstracts submitted in response to the call for papers. In addition, there were invited lectures by Andrei Broder (*AltaVista*), Fernando Pereira (*AT&T Research Labs*), and Ian H. Witten (*University of Waikato*).

The symposium was preceded by a two-day summer school set up to attract and train young researchers. The lecturers at the school were Greg Butler, Clement Lam, and Gus Grahne: *BLAST! How do you search sequence databases?*, David Bryant: *Phylogeny*, Raffaele Giancarlo: *Algorithmic aspects of speech recognition*, Nadia El-Mabrouk: *Genome rearrangement*, Laxmi Parida: *Flexible-pattern discovery*, and Ian H. Witten: *Adaptive text mining: inferring structure from sequences*.

Combinatorial Pattern Matching (CPM) addresses issues of searching and matching strings and more complicated patterns such as trees, regular expressions graphs, point sets, and arrays. The goal is to derive non-trivial combinatorial properties of such structures and to exploit these properties in order to achieve superior performance for the corresponding computational problems.

Over recent years a steady flow of high-quality research on this subject has changed a sparse set of isolated results into a fully-fledged area of algorithmics. This area is continuing to grow even further due to the increasing demand for speed and efficiency that comes from important and rapidly expanding applications such as the World Wide Web, computational biology, and multimedia systems, involving requirements for information retrieval, data compression, and pattern recognition. The objective of the annual CPM gatherings is to provide an international forum for research in combinatorial pattern matching and related applications.

The first ten meetings were held in Paris (1990), London (1991), Tucson (1992), Padova (1993), Asilomar (1994), Helsinki (1995), Laguna Beach (1996), Aahrus (1997), Piscataway (1998), and Warwick (1999). After the first meeting, a selection of papers appeared as a special issue of *Theoretical Computer Science* in Volume 92. The proceedings of the third to tenth meetings appeared as volumes 644, 684, 807, 937, 1075, 1264, 1448, and 1645 of the Springer LNCS series.

The general organization and orientation of CPM conferences is coordinated by a steering committee composed of:

The program committee of CPM 2000 consisted of:

Amihood Amir,
Bar Ilan University
Bonnie Berger,
MIT
Byron Dom,
IBM Almaden
Raffaele Giancarlo, Co-chair,
University of Palermo
Dan Gusfield,
University of California, Davis
Monika Henzinger,
Google, Inc.
John Kececioglu,
University of Georgia
Gad Landau,
University of Haifa
& Polytechnic University
Wojciech Rytter,
University of Warsaw
& University of Liverpool
Marie-France Sagot,
Institut Pasteur
Cenk Sahinalp,
Case Western Reserve University
David Sankoff, Co-chair,
Université de Montréal
Jim Storer,
Brandeis University
Esko Ukkonen,
University of Helsinki

The local organizing committee, all from the *Université de Montréal*, consisted of:

Nadia El-Mabrouk
Louis Pelletier
David Sankoff
Sylvain Viart

The conference was supported by the **Centre de recherches mathématiques** of the *Université de Montréal*, in the context of a thematic year on Mathematical Methods in Biology and Medecine (2000-2001).

April 2000

Raffaele Giancarlo
David Sankoff

List of Reviewers

O. Arbel
D. Brown
D. Bryant
C. Constantinescu
C. Cormode
K. Diks
F. Ergun
R. Fagin
P. Ferragina
R. Grossi
R. Kumar
A. Malinowski
D. Modha
M. Nykanen
W. Plandowski
A. Piccolboni
R. Sprugnoli
M. Sciortino
D. Shapira
J. Sharp
L. Stockmeyer

Table of Contents

Invited Lectures

Summer School Lectures

Contributed Papers

Identifying and Filtering Near-Duplicate Documents

Andrei Z. Broder*

AltaVista Company, San Mateo, CA 94402, USA
andrei.broder@av.com

Abstract. The mathematical concept of document resemblance captures well the informal notion of syntactic similarity. The resemblance can be estimated using a fixed size "sketch" for each document. For a large collection of documents (say hundreds of millions) the size of this sketch is of the order of a few hundred bytes per document.
However, for efficient large scale web indexing it is not necessary to determine the actual resemblance value: it suffices to determine whether newly encountered documents are duplicates or near-duplicates of documents already indexed. In other words, it suffices to determine whether the resemblance is above a certain threshold. In this talk we show how this determination can be made using a "sample" of less than 50 bytes per document.
The basic approach for computing resemblance has two aspects: first, resemblance is expressed as a set (of strings) intersection problem, and second, the relative size of intersections is evaluated by a process of random sampling that can be done independently for each document. The process of estimating the relative size of intersection of sets and the threshold test discussed above can be applied to arbitrary sets, and thus might be of independent interest.
The algorithm for filtering near-duplicate documents discussed here has been successfully implemented and has been used for the last three years in the context of the AltaVista search engine.

1 Introduction

A Communist era joke in Russia goes like this: Leonid Brezhnev (the Party leader) wanted to get rid of the Premier, Aleksey Kosygin. (In fact he did, in 1980.) So Brezhnev, went to Kosygin and said: "My dear friend and war comrade Aleksey, I had very disturbing news: I just found out that you are Jewish: I have no choice, I must ask you to resign." Kosygin, in total shock says: "But Leonid, as you know very well I am not Jewish!"; then Brezhnev says: "Well, Aleksey, then think about it..."

* Most of this work was done while the author was at Compaq's System Research Center in Palo Alto. A preliminary version of this work was presented (but not published) at the "Fun with Algorithms" conference, Isola d'Elba, 1998.

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 1–10, 2000.

What this has to do with near-duplicate documents? In mid 1995, the AltaVista web search engine was built at the Digital research labs in Palo Alto (see [10]). Soon after the first internal prototype was deployed, a colleague, Chuck Thacker, came to me and said: "I really like AltaVista, but it is very annoying that often half the first page of answers is just the same document in many variants." "I know" said I. "Well," said Chuck, "you did a lot of work on fingerprinting documents; can you make up a fingerprinting scheme such that two documents that are near-duplicate get the same fingerprint?" I was of course indignant: "No way!! You miss the idea of fingerprints completely: fingerprints are such that with high probability two distinct documents will have different fingerprints, no matter how little they differ! Similar documents getting the same fingerprint is entirely against their purpose." So, of course, Chuck said: "Well, then think about it..." ...and as usual, Chuck was right.

Eventually I found found a solution to this problem, based on a mathematical notion called *resemblance* [4]. Surprisingly, fingerprints play an essential role in it.

The resemblance measures whether two (web) documents are roughly the same, that is, they have the same content except for modifications such as formatting, minor corrections, capitalization, web-master signature, logo, etc. The resemblance is a number between 0 and 1, defined precisely below, such that when the resemblance is close to 1 it is likely that the documents are roughly the same. To compute the resemblance of two documents it suffices to keep for each document a "sketch" of a few (three to eight) hundred bytes consisting of a collection of fingerprints of "shingles" (contiguous subsequences of words, sometimes called "q-grams"). The sketches can be computed fairly fast (linear in the size of the documents) and given two sketches the resemblance of the corresponding documents can be computed in linear time in the size of the sketches. Furthermore, clustering a collection of m documents into sets of closely resembling documents can be done in time proportional to $m \log m$ rather than m^2.

This first use of this idea was in a joint work with Steve Glassman, Mark Manasse, and Geoffrey Zweig to cluster a collection of over 30,000,000 documents into sets of closely resembling documents (above 50% resemblance). The documents were retrieved from a month long "full" crawl of the World Wide Web performed by AltaVista in April 96. (See [7].) (It is amusing to note that three years later, by mid 1999, AltaVista was crawling well over 20 million pages *daily*.)

Besides fingerprints, another essential ingredient in the computation of resemblance is a pseudo-random permutation of a large set, typically the set $[0, \ldots, 2^{64} - 1]$. In turns out that to achieve the desired result, the permutation must be drawn from a *min-wise independent* family of permutations. The concept of min-wise independence is in the same vein as the well known concept of pair-wise independence, and has many interesting properties. Moses Charikar, Alan Frieze, Michael Mitzenmacher, and I studied this concept in a paper [5].

The World Wide Web continues to expand at a tremendous rate. It is estimated that the number of pages doubles roughly every nine moths to year [2,14].

Hence the problem of eliminating duplicates and near-duplicates from the index is extremely important. The fraction of the total WWW collection consisting of duplicates and near-duplicates has been estimated at 30 to 45%. (See [7] and [13].) These documents arise innocently (e.g. local copies of popular documents, mirroring), maliciously (e.g., "spammers" and "robot traps"), and erroneously (crawler or server mistakes). In any case they represent a serious problem for indexing software for two main reasons: first, indexing of duplicates wastes expensive resources and second, users are seldom interested in seeing documents that are "roughly the same" in response to their queries.

However, when applying the sketch computation algorithm to the entire corpus indexed by AltaVista then even the modest storage costs described above become prohibitive. On the other hand, we are interested only whether the resemblance is above a very high threshold; the actual value of the resemblance does not matter.

This paper describes how to apply further processing to the sketches mentioned above to construct for each document a short vector of "features." With high probability, two documents share more than a certain number of features if and only if their resemblance is very high. For instance, using 6 features of 8 bytes, that is, 48 bytes/document, for a set of 200,000,000 documents:

- The probability that two documents that have resemblance greater than 97.5% *do not share* at least two features is less than 0.01. The probability that two documents that have resemblance greater than 99% *do not share* at least two features is less than 0.00022.
- The probability that two documents that have resemblance less than 77% *do share* two or more features is less than 0.01 The probability that two documents that have resemblance less than 50% share two or more features is less than 0.6×10^{-7}.

Thus the feature based mechanism for near-duplicate detection has excellent filtering properties. The probability of acceptance for this example (that is more than 2 common features) as a function of resemblance is graphed in Figure 1 on a linear scale and in Figure 2 on a logarithmic scale.

2 Preliminaries

We start by reviewing some concepts and algorithms described in more detail in [4] and [7].

The basic approach for computing resemblance has two aspects: First, resemblance is expressed as a set intersection problem, and second, the relative size of intersections is evaluated by a process of random sampling that can be done independently for each document. (The process of estimating the relative size of intersection of sets can be applied to arbitrary sets.)

The reduction to a set intersection problem is done via a process called *shingling*. Via shingling each document D gets an associated set S_D. This is done as follows:

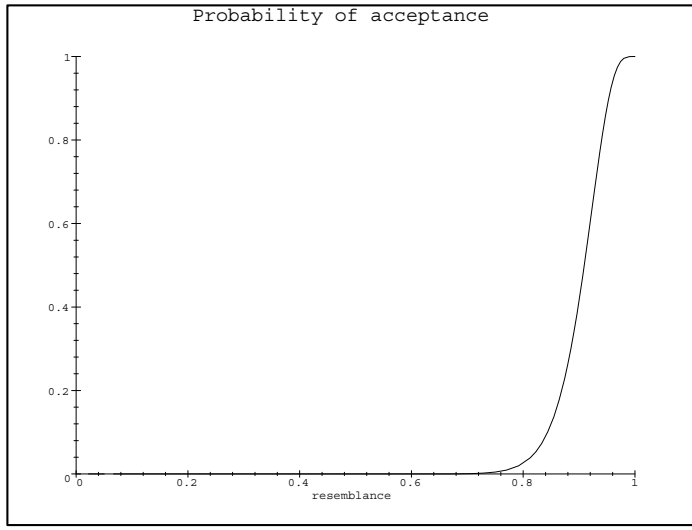


Fig. 1. The graph of $P_{6,14,2}(x)$ – linear scale

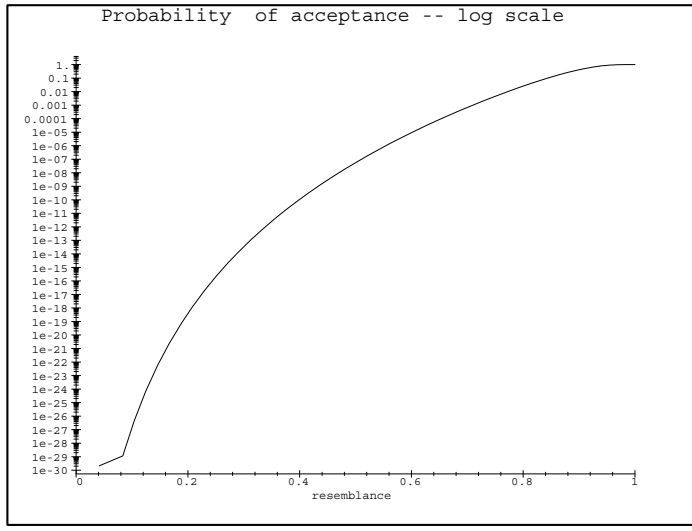


Fig. 2. The graph of $P_{6,14,2}(x)$ – logarithmic scale

We view each document as a sequence of tokens. We can take tokens to be letters, or words, or lines. We assume that we have a parser program that takes an arbitrary document and reduces it to a canonical sequence of tokens. (Here "canonical" means that any two documents that differ only in formatting or other information that we chose to ignore, for instance punctuation, formatting commands, capitalization, and so on, will be reduced to the same sequence.) So from now on a document means a canonical sequence of tokens.

A contiguous subsequence of w tokens contained in D is called a *shingle*. A shingle of length q is also known as a *q-gram*, particularly when the tokens are alphabet letters. Given a document D we can associate to it its *w-shingling* defined as the set of all shingles of size w contained in D. So for instance the 4-shingling of

$$(\texttt{a,rose,is,a,rose,is,a,rose})$$

is the set

$$\{(\texttt{a,rose,is,a}), (\texttt{rose,is,a,rose}), (\texttt{is,a,rose,is})\}$$

(It is possible to use alternative definitions, based on multisets. See [4] for details.)

Rather than deal with shingles directly, it is more convenient to associate to each shingle a numeric uid (unique id). This done by *fingerprinting* the shingle. (Fingerprints are short tags for larger objects. They have the property that if two fingerprints are different then the corresponding objects are certainly different and there is only a small probability that two different objects have the same fingerprint. This probability is typically exponentially small in the length of the fingerprint.)

For reasons explained in [4] it is particularly advantageous to use Rabin fingerprints [15] that have a very fast software implementation [3]. Rabin fingerprints are based on polynomial arithmetic and can be constructed in any length. It is important to choose the length of the fingerprints so that the probability of collisions (two distinct shingles getting the same fingerprint) is sufficiently low. (More about this below.) In practice 64 bits Rabin fingerprints are sufficient.

Hence from now on we associate to each document D a set of numbers S_D that is the result of fingerprinting the set of shingles in D. Note that the size of S_D is about equal to the number of words in D and thus storing S_D on-line for every document in a large collection is infeasible.

The *resemblance* $r(A, B)$ of two documents, A and B, is defined as

$$r(A, B) = \frac{|S_A \cap S_B|}{|S_A \cup S_B|}.$$

Experiments seem to indicate that high resemblance (that is, close to 1) captures well the informal notion of "near-duplicate" or "roughly the same". (There are analyses that relate the "q-gram distance" to the edit-distance – see [16].)

Our approach to determining syntactic similarity is related to the sampling approach developed independently by Heintze [8], though there are differences

in detail and in the precise definition of the measures used. Related sampling mechanisms for determining similarity were also developed by Manber [9] and within the Stanford SCAM project [1,11,12].

To compute the resemblance of two documents it suffices to keep for each document a relatively small, fixed size *sketch*. The sketches can be computed fairly fast (linear in the size of the documents) and given two sketches the resemblance of the corresponding documents can be computed in linear time in the size of the sketches.

This is done as follows. Assume that for all documents of interest $S_D \subseteq \{0, \ldots, n-1\} \stackrel{def}{=} [n]$. (As noted, in practice $n = 2^{64}$.) Let π be chosen uniformly at random over S_n, the set of permutations of $[n]$. Then

$$\Pr\big(\min\{\pi(S_A)\} = \min\{\pi(S_B)\}\big) = \frac{|S_A \cap S_B|}{|S_A \cup S_B|} = r(A, B). \tag{1}$$

Proof. Since π is chosen uniformly at random, for any set $X \subseteq [n]$ and any $x \in X$, we have

$$\Pr\big(\min\{\pi(X)\} = \pi(x)\big) = \frac{1}{|X|}. \tag{2}$$

In other words all the elements of any fixed set X have an equal chance to become the minimum element of the image of X under π.

Let α be the smallest image in $\pi(S_A \cup S_B)$. Then $\min\{\pi(S_A)\} = \min\{\pi(S_B)\}$, if and only if α is the image of an element in $S_A \cap S_B$. Hence

$$\begin{aligned}\Pr\big(\min\{\pi(S_A)\} = \min\{\pi(S_B)\}\big) &= \Pr(\pi^{-1}(\alpha) \in S_A \cap S_B) \\ &= \frac{|S_A \cap S_B|}{|S_A \cup S_B|} = r_w(A, B).\end{aligned}$$

Hence, we can choose, once and for all, a set of t independent random permutations $\pi_1, \ldots, \pi_t$. (For instance we can take $t = 100$.) For each document D, we store a *sketch*, which is the list

$$\bar{S}_A = (\min\{\pi_1(S_A)\}, \min\{\pi_2(S_A)\}, \ldots, \min\{\pi_t(S_A)\}).$$

Then we can readily estimate the resemblance of A and B by computing how many corresponding elements in $\bar{S}_A$ and $\bar{S}_B$ are equal. (In [4] it is shown that in fact we can use a single random permutation, store the t smallest elements of its image, and then merge-sort the sketches. However for the purposes of this paper independent permutations are necessary.)

In practice, we have to deal with the fact it is impossible to choose and represent π uniformly at random in S_n for large n. We are thus led to consider smaller families of permutations that still satisfy the *min-wise independence condition* given by equation (2), since min-wise independence is necessary and sufficient for equation (1) to hold. This is further explored in [5] where it is shown that random linear transformations are likely to suffice in practice. See also [6] for an alternative implementation. We will ignore this issue in this paper.

So far we have seen how to estimate the resemblance of a pair of documents. For this purpose the shingle fingerprints can be quite short since collisions have only a modest influence on our estimate if we first apply a random permutation to the shingles and then fingerprint the minimum value.

However sketches allow us to group a collection of m documents into sets of closely resembling documents in time proportional to $m \log m$ rather than m^2, assuming that the clusters are well separated which is the practical case.

We perform the clustering algorithm in four phases. In the first phase, we calculate a sketch for every document as explained. This step is linear in the total length of the documents.

To simplify the exposition of the next three phases we'll say temporarily that each sketch is composed of shingles, rather than images of the fingerprint of shingles under random permutations of $[n]$.

In the second phase, we produce a list of all the shingles and the documents they appear in, sorted by shingle value. To do this, the sketch for each document is expanded into a list of ⟨shingle value, document ID⟩ pairs. We simply sort this list. This step takes time $O(m \log m)$ where m is the number of documents.

In the third phase, we generate a list of all the pairs of documents that share any shingles, along with the number of shingles they have in common. To do this, we take the file of sorted ⟨shingle, ID⟩ pairs and expand it into a list of ⟨ID, ID, count of common shingles⟩ triplets by taking each shingle that appears in multiple documents and generating the complete set of ⟨ID, ID, 1⟩ triplets for that shingle. We then apply a merge-sort procedure (adding the counts for matching ID - ID pairs) to produce a single file of all ⟨ID, ID, count⟩ triplets sorted by the first document ID. This phase requires the greatest amount of disk space because the initial expansion of the document ID triplets is quadratic in the number of documents sharing a shingle, and initially produces many triplets with a count of 1. Because of this fact we must choose the length of the shingle fingerprints so that the number of collisions is small. To ensure this we can take it to be say $2 \log_2 m + 20$. In practice 64 bits fingerprints suffice.

In the final phase, we produce the complete clustering. We examine each ⟨ID, ID, count⟩ triplet and decide if the document pair exceeds our threshold for resemblance. If it does, we add a link between the two documents in a union-find algorithm. The connected components output by the union-find algorithm form the final clusters.

3 Filtering Near-Duplicates

Consider two documents, A and B, that have resemblance ρ. If ρ is close to 1, then almost all the elements of $\bar{S}_A$ and $\bar{S}_B$ will be pairwise equal. The idea of duplicate filtering is to divide every sketch into k groups of s elements each. The probability that all the elements of a group are pair-wise equal is simply ρ^s and the probability that two sketches have r or more equal groups is

$$P_{k,s,r} = \sum_{r \le i \le k} \binom{k}{i} \rho^{s \cdot i} (1 - \rho^s)^{k-i}.$$

The remarkable fact is that for suitable choices of $[k, s, r]$ the polynomial $P_{k,s,r}$ behaves as a very sharp high-band pass filter even for small values of k. For instance Figure 1 graphs $P_{6,14,2}(x)$ on a linear scale and Figure 2 graphs it on a logarithmic scale. The sharp drop-off is obvious.

To use this fact, we first compute for each document D the sketch $\bar{S}_D$ as before, using $k \cdot s$ independent permutations. (We can now be arbitrarily generous with the length of the fingerprints used to create shingle uid's; however 64 bits are plenty for our situation.) We then split $\bar{S}_D$ into k groups of s elements and fingerprint each group. (To avoid dependencies, we use a different irreducible polynomial for these fingerprints.) We can also concatenate to each group a group id number before fingerprinting.

Now all we need to store for each document is these k fingerprints, called "features". Because fingerprints could collide the probability that two features are equal is

$$\rho^s + p_f,$$

where p_f is the collision probability. This would indicate that it suffices to use fingerprints long enough to so that p_f is less than say 10^{-6}. However, when applying the filtering mechanism to a large collection of documents, we again use the clustering process described above, and hence we must avoid spurious sharing of features. Nevertheless, for our problem 64 bits fingerprints are again sufficient.

It is particularly convenient, if possible, to choose the threshold r to be 1 or 2. If $r = 2$ then the third phase of the merging process becomes much simpler since we don't need to keep track of how many features are shared by various pairs of documents: we simply keep a list of pairs known to share at least one feature. As soon as we discover that one of these pairs shares a second feature, we know that with high probability the two documents are near-duplicates, and thus one of them can be removed from further consideration. If $r = 1$ the third phase becomes moot. In general it is possible to avoid the third phase if we again group every r features into a super-feature, but this forces the number of features per document to become $\binom{k}{r}$.

4 Choosing the Parameters

As often the case in filter design, choosing the parameters is half science, half black magic. It is useful to start from a target threshold resemblance ρ_0. Ideally

$$P_{k,s,r}(\rho) = \begin{cases} 1, \text{ for } \rho \geq \rho_0; \\ 0, \text{ otherwise.} \end{cases}$$

Clearly, once s is chosen, r should be approximately $k \cdot \rho_0^s$ and the larger k (and r) the sharper the filter. (Of course, we are restricted to integral values for k, s, and r.)

If we make the (unrealistic) assumption that resemblance is uniformly distributed between 0 and 1 within the set of pairs of documents to be checked,

then the total error is proportional to

$$\int_0^{\rho_0} P_{k,s,r}(x)\,dx + \int_{\rho_0}^1 \big(1 - P_{k,s,r}(x)\big)\,dx$$

Differentiating with respect to ρ_0 we obtain that this is minimized when $P(\rho_0) = 1/2$. To continue with our example we have $P_{6,14,2}(x) = 1/2$ for $x = 0.909\ldots$.

A different approach is to chose s so that the slope of x^s at $x = \rho_0$ is maximized. This happens when

$$\frac{\partial}{\partial s}\left(s\rho_0^{s-1}\right) = 0 \tag{3}$$

or $s = 1/\ln(1/\rho_0)$. For $s = 14$ the value that satisfies (3) is $\rho_0 = 0.931\ldots$.

In practice these ideas give only a starting point for the search for a filter that provides the required trade-offs between error bounds, time, and space. It is necessary to graph the filter and do experimental determinations.

5 Conclusion

We have presented a method that can eliminate near-duplicate documents from a collection of hundreds of millions of documents by computing independently for each document a vector of features less than 50 bytes long and comparing only these vectors rather than entire documents. The entire processing takes time $O(m \log m)$ where m is the size of the collection. The algorithm described here has been successfully implemented and is in current use in the context of the AltaVista search engine.

Acknowledgments

I wish to thank Chuck Thacker who challenged me to find an efficient algorithm for filtering near-duplicates. Some essential ideas behind the resemblance definition and computation were developed in conversations with Greg Nelson. The prototype implementation for AltaVista was done in collaboration with Mike Burrows and Mark Manasse.

References

1. S. Brin, J. Davis, H. García-Molina. Copy Detection Mechanisms for Digital Documents. *Proceedings of the ACM SIGMOD Annual Conference*, May 1995.
2. K. Bharat and A. Z. Broder. A Technique for Measuring the Relative Size and Overlap of Public Web Search Engines. In *Proceedings of Seventh International World Wide Web Conference*, pages 379–388, 1998.
3. A. Z. Broder. Some applications of Rabin's fingerprinting method. In R. Capocelli, A. De Santis, and U. Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

4. A. Z. Broder. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of Sequences 1997*, pages 21–29. IEEE Computer Society, 1997.
5. A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-Wise Independent Permutations. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 327–336, 1998.
6. A. Z. Broder and U. Feige. Min-Wise versus Linear Independence. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 147–154, 2000.
7. A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the Web. In *Proceedings of the Sixth International World Wide Web Conference*, pages 391–404, 1997.
8. N. Heintze. Scalable Document Fingerprinting. *Proceedings of the Second USENIX Workshop on Electronic Commerce*, pages 191–200, 1996.
9. U. Manber. Finding similar files in a large file system. *Proceedings of the Winter 1994 USENIX Conference*,pages 1–10, 1994.
10. R. Seltzer, E. J. Ray, and D. S. Ray. *The AltaVista Search Revolution: How to Find Anything on the Internet.* McGraw-Hill, 1996.
11. N. Shivakumar, H. García-Molina. SCAM: A Copy Detection Mechanism for Digital Documents. *Proceedings of the 2nd International Conference on Theory and Practice of Digital Libraries*, 1995.
12. N. Shivakumar and H. García-Molina. Building a Scalable and Accurate Copy Detection Mechanism. *Proceedings of the 3nd International Conference on Theory and Practice of Digital Libraries*, 1996.
13. N. Shivakumar and H. García-Molina. Finding near-replicas of documents on the web. In *Proceedings of Workshop on Web Databases (WebDB'98)*, March 1998.
14. Z. Smith. The Truth About the Web: Crawling Towards Eternity, *Web Techniques Magazine*, May 1997.
15. M. O. Rabin. Fingerprinting by random polynomials. Center for Research in Computing Technology, Harvard University, Report TR-15-81, 1981.
16. E. Ukkonen. Approximate string-matching distance and the q-gram distance. In R. Capocelli, A. De Santis, and U. Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*, pages 300–312. Springer-Verlag, 1993.

Machine Learning for Efficient Natural-Language Processing

Fernando Pereira

AT&T Labs, Shannon Laboratory, 180 Park Ave.
Florham Park, NJ 07932, USA
pereira@research.att.com

Abstract. Much of computational linguistics in the past thirty years assumed a ready supply of general and linguistic knowledge, and limitless computational resources to use it in understanding and producing language. However, accurate knowledge is hard to acquire and computational power is limited. Over the last ten years, inspired in part by advances in speech recognition, computational linguists have been investigating alternative approaches that take advantage of the statistical regularities in large text collections to automatically acquire efficient approximate language processing algorithms. Such machine-learning techniques have achieved remarkable successes in tasks such as document classification, part-of-speech tagging, named-entity recognition and classification, and even parsing and machine translation.

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 11–11, 2000.

Browsing around a Digital Library: Today and Tomorrow

Ian H. Witten

Department of Computer Science, University of Waikato
Hamilton, New Zealand
ihw@cs.waikato.ac.nz

Abstract. What will it be like to work in tomorrow's digital library? We begin by browsing around an experimental digital library of the present, glancing at some collections and showing how they are organized. Then we look to the future. Although present digital libraries are quite like conventional libraries, we argue that future ones will feel qualitatively different. Readers—and writers—will work in the library using a kind of context-directed browsing. This will be supported by structures derived from automatic analysis of the contents of the library—not just the catalog, or abstracts, but the full text of the books and journals—using new techniques of text mining.

1 Introduction

Over sixty years ago, science fiction writer H.G. Wells was promoting the concept of a "world brain" based on a permanent world encyclopedia which "would be the mental background of every intelligent [person] in the world. It would be alive and growing and changing continually under revision, extension and replacement from the original thinkers in the world everywhere. ... even journalists would deign to use it" [14]. Eight years later, Vannevar Bush, the highest-ranking scientific administrator in the U.S. war effort, invited us to "consider a future device for individual use, which is a sort of mechanized private file and library ... a device in which an individual stores all his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility" [2]. Fifteen years later, J.C.R. Licklider, head of the U.S. Department of Defense's Information Processing Techniques Office, envisioned that human brains and computing machines would be coupled together very tightly, and imagined this to be supported a "network of 'thinking centers' that will incorporate the functions of present-day libraries together with anticipated advances in information storage and retrieval" [8]. Thirty-five years later we became accustomed to hearing similar pronouncements from the U.S. Presidential office.

Digital libraries, conceived by visionary thinkers and fertilized with resources by today's politicians, are undergoing a protracted labor and birth. Libraries are society's repositories for knowledge, and digital libraries are of the utmost strategic importance in a knowledge-based economy. Not surprisingly, many countries

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 12–26, 2002.

have initiated large-scale digital library projects. Some years ago the DLI initiative was set up in the U.S. (and has now entered a second phase); in the U.K. the Elib program was set up at about the same time; other countries in Europe and the Pacific Rim have followed suit. Digital libraries will likely figure amongst the most important and influential institutions of the 21st Century.

But what is a digital library? Ten definitions of the term have been culled from the literature by Fox [4], and their spirit is captured in the following brief characterization [1]:

> *A focused collection of digital objects, including text, video, and audio, along with methods for access and retrieval, and for selection, organization, and maintenance of the collection.*

This definition gives equal weight to user (access and retrieval) and librarian (selection, organization and maintenance). Other definitions in the literature, emanating mostly from technologists, omit—or at best downplay—the librarian's role, which is unfortunate because it is the selection, organization, and maintenance that will distinguish digital libraries from the anarchic mess that we call the World Wide Web. However, digital libraries tend to blur what used to be a sharp distinction between user and librarian—because the ease of augmenting, editing, annotating and re-organizing electronic collections means that they will support the development of new knowledge *in situ*.

What's it like to work in a digital library? Will it feel like a conventional library, but more computerized, more networked, more international, more all-encompassing, more convenient? I believe the answer is no: it will feel qualitatively different. Not only will it be with you on your desktop (or at the beach, or in the plane), but information workers will work "inside" the library in a way that is quite unlike how they operate at present. It's not just that knowledge and reference services will be fully portable, operating round the world, around the clock, throughout the year, freeing library patrons from geographic and temporal constraints—important and liberating as these are. It's that when new knowledge is created it will be fully contextualized and both sited within and cited by existing literature right from its conception.

In this paper, we browse around a digital library, looking at tools and techniques under development. "Browse" is used in a dual sense. First we begin by browsing a particular collection, and then look briefly at some others. Second, we examine the digital library's ability to support novel browsing techniques. These situate browsing within the reader's current context and unobtrusively guide them in ways that are relevant to what they are doing. Context-directed browsing is supported by structures derived from automatic analysis of the library's contents—not just the catalog, or abstracts, but the *full text* of the documents—using techniques that are being called "text mining." Of course, other ways of finding information are important too—user searching, librarian recommendations, automatic notification, group collaboration—but here we focus on browsing. The work described was undertaken by members of the New Zealand Digital Library project.

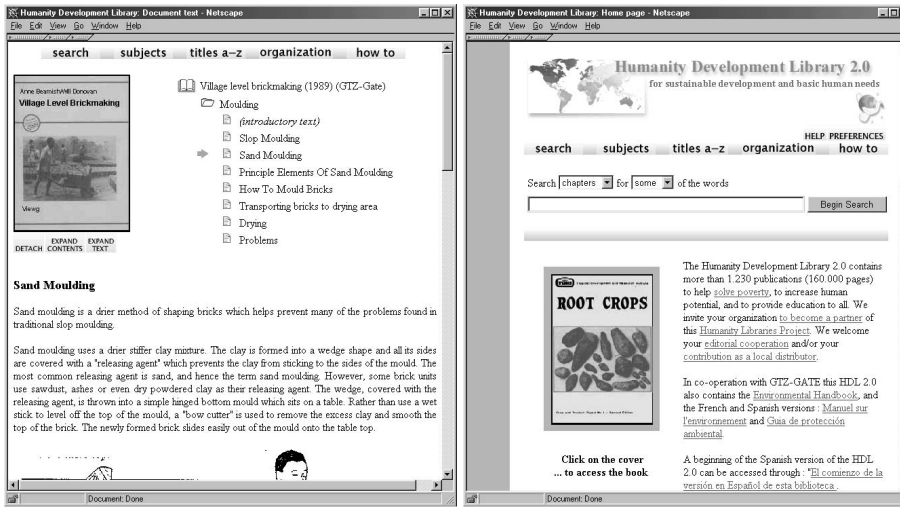


Fig. 1. (a) *Village Level Brickmaking*, (b) The collection's home page

2 The Humanity Development Library

Figure 1a shows a book in the *Humanity Development Library*, a collection of humanitarian information put together by the Global Help Project to address the needs of workers in developing countries (`www.nzdl.org/hdl`). This book might have been reached by a directed full-text search, or by browsing one of a number of access structures, or by clicking on one of a gallery of images. On opening the book, which is entitled *Village Level Brickmaking*, a picture of its cover appears at the top, beside a hierarchical table of contents. In the figure, the reader has drilled down into a chapter on *moulding* and a subsection on *sand moulding*, whose text appears below. Readers can expand the table of contents from the section to the whole book; and expand the text likewise (which is very useful for printing). The ever-present picture of the book's cover gives a feeling of physical presence and a constant reminder of the context.

Readers can browse the collection in several different ways, as determined by the editor who created it. Figure 1b shows the collection's home page, at the top of which (underneath the logo) is a bar of five buttons that open up different access mechanisms. A subject hierarchy provides a tree-structured classification scheme for the books. Book titles appear in an alphabetical index. A separate list gives participating organizations and the material that they contributed. A "how-to" list of helpful hints, created by the collection's editor, allows a particular book to be accessed from brief phrases that describe the problems the book addresses. However a book is reached, it appears in the standard form illustrated in Figure 1a, along with the cover picture to give a sense of presence. The different access mechanisms help solve the librarian's dilemma of where to

shelve books [9]: each one appears on many different virtual shelves, shelves that are organized in different ways.

Full-text search of titles and entire documents provide important additional access mechanisms. The search engine that we use, MG [15], supports searching over the full text of the document—not merely a document surrogate as in conventional digital library retrieval systems. User feedback from an earlier version of this collection indicated that Boolean searching was more confusing than helpful for the targeted users. Previous research suggests that difficulties with Boolean syntax and semantics are widespread, and transaction log analysis of several library retrieval systems indicates that by far the most popular Boolean operator is AND; the others are rarely used. For all these reasons, the interface default for this collection is ranked queries. However, to enable users to construct high-precision conjunctive searches where necessary, selecting "search ... for *all* the words" in the query dialog produces the syntax-free equivalent of a conjunctive query.

Just as libraries display new acquisitions or special collections in the foyer to pique the reader's interest, this collection's home page (Figure 1b) highlights a particular book that changes every few seconds: it can be opened by clicking on the image. This simple display is extraordinarily compelling. And just as libraries may display a special book in a glass case, open at a different page each day, a "gallery" screen can show an ever-changing mosaic of images from pages of the books, remarkably informative images that, when clicked, open the book to that page. Or a scrolling "Times Square" display of randomly selected phrases that, when clicked, take you to the appropriate book. The possibilities are endless.

The *Humanity Development Library* is a focused collection of 1250 books—miniscule by library standards, but nevertheless comprehensive within the targeted domain. It contains 53,000 chapters, 62 million words, and 32,000 pictures. Although the text occupies 390 MB, it compresses to 102 MB and the two indexes—for titles and chapters respectively—compress to less than 80 MB. The images (mostly in PNG format) occupy 290 MB. Associated files bring the total size of the collection to 505 MB. Even if there were twice as much text, and the same images, it would still fit comfortably on a CD-ROM, along with all the necessary software. A single DVD-ROM would hold a collection twenty times the size—still small by library standards, but immense for a fully portable collection.

3 An Experimental Testbed: The NZDL

The *Humanity Development Library* is just one of two dozen publicly-available collections produced by the New Zealand Digital Library (NZDL) project and listed on the project's home page (`www.nzdl.org`), part of which is shown in Figure 2a—this illustrates the the wide range of collections. This project aims to develop the underlying infrastructure for digital libraries and provide example collections that demonstrate how it can be used. The library is international,

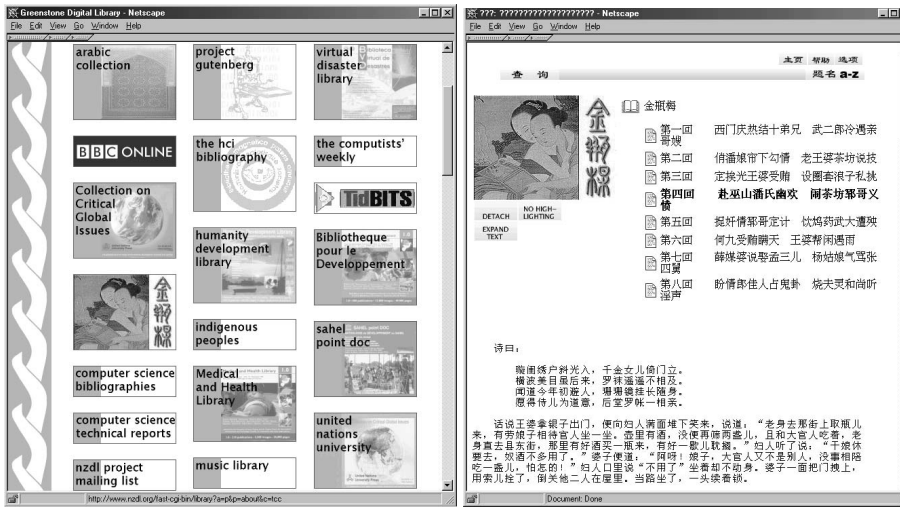


Fig. 2. (a) Some of the NZDL collections, (b) Reading a Chinese book

and the Unicode character set is used throughout: there are interfaces in English, Maori, French, German, Arabic, and Chinese, and collections have been produced in all these languages. Digital libraries are particularly empowering for the disabled, and there is a text-only version of the interface intended for visually impaired users.

The editors of the *Humanity Development Library* have gone to great lengths to provide a rich set of access structures. However, this is a demanding, labor-intensive task, and most collections are not so well organized. The basic access tool in the NZDL is full-text searching, which is available for all collections and is provided completely automatically when a collection is built. Some collections allow, in addition, traditional catalog searching based on author, title, and keywords, and full-text search within abstracts. Our experience is that while the user interface is considerably enhanced when traditional library cataloging information is available, it is often prohibitively expensive to create formal cataloging information for electronically-gathered collections. With appropriate indexes, full-text retrieval can be used to approximate the services provided by a formal catalog.

3.1 Collections

The core of any library is the collections it contains. A few examples will illustrate the variety and scope of the services provided.

The historically first collection, *Computer Science Technical Reports*, now contains 46,000 reports—1.3 million pages, half a billion words—extracted automatically from 34 GB of raw PostScript. There is no bibliographic or "metadata" information: we have only the contents of the reports (and the names of the FTP

sites from which they were gathered). Many are Ph.D. theses which would otherwise be effectively lost except to a miniscule community of cognoscenti: full-text search reaches right inside the documents and makes them accessible to anyone looking for information on that topic.

As well as the simplified searching interface for the *Humanity Development Library* described above, users can choose a more comprehensive query interface (via a *Preferences* page). Case-folding and stemming can be independently enabled or disabled, and full Boolean query syntax is supported as well as ranked queries. Moreover, in the *Computer Science Technical Reports* searches can be restricted to the first page of reports, which approximates an author/title search in the absence of specific bibliographic details of the documents. Although this is a practical solution, the collection nevertheless presents a raw, unpolished appearance compared with the *Humanity Development Library*, reflecting the difference between a carefully-edited set of documents, including hand-prepared classification indexes and other metadata, and a collection of information pulled mechanically off the Web and organized without any human intervention at all.

There are several collections of books, including, for example, the English books entered by the Gutenberg project. Figure 2b shows a book in a collection of classical Chinese literature. The full text was available on the Web; we automatically extracted the section headings to provide the table of contents visible at the upper right, and scanned the book's cover to generate the cover image. One can perform full-text search on the complete contents or on section headings alone, using the Chinese language (of course your Web browser must be set up correctly to work in Chinese). There is also a browsable list of book titles.

An expressly bilingual collection of *Historic New Zealand Newspapers* contains issues of forty newspapers published between 1842 and 1933 for a Maori audience. Collected on microfiche, these constitute 12,000 page images. Although they represent a significant resource for historians, linguists and social scientists, their riches remain largely untapped because of the difficulty of accessing, searching and browsing material in unindexed microfiche form. Figure 3 shows the parallel English-Maori text retrieved from the newspaper *Te Waka Maori* of August 1878 in response to the query *Rotorua*, a small town in New Zealand. Searching is carried out on electronic text produced using OCR; once the target is identified, the corresponding page image can be displayed.

3.2 The Greenstone Software

All these collections are created using the Greenstone software developed by the NZDL project [17]. Information collections built by Greenstone combine extensive full-text search facilities with browsing indexes based on different metadata types. There are several ways for users to find information, although they differ between collections depending on the metadata available and the collection design. You can *search for particular words* that appear in the text, or within a section of a document, or within a title or section heading. You can *browse documents by title*: just click on the displayed book icon to read it. You can *browse documents by subject*. Subjects are represented by bookshelves: just click

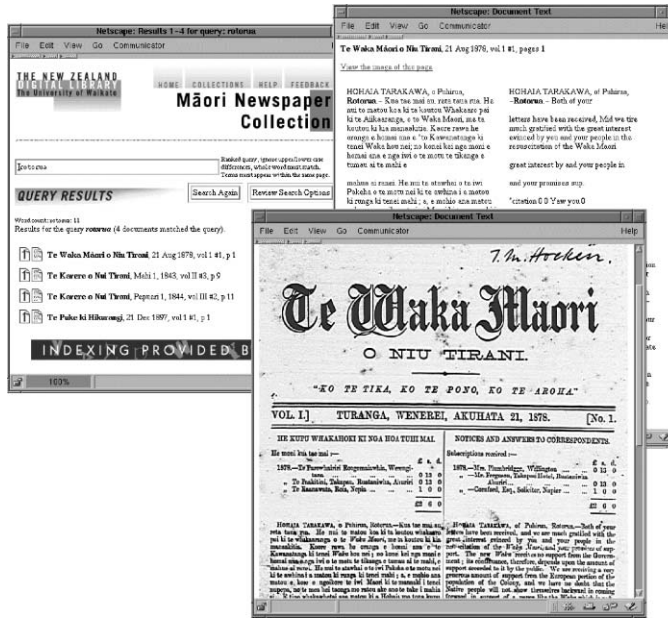


Fig. 3. Searching the *Historic New Zealand newspapers* collection

on a shelf to see the books. Where appropriate, documents come complete with a table of contents (constructed automatically): you can click on a chapter or subsection to open it, expand the full table of contents, or expand the full document.

A distinction is made between *searching* and *browsing*. Searching is full-text, and—depending on the collection's design—the user can choose between indexes built from different parts of the documents, or from different metadata. Some collections have an index of full documents, an index of sections, an index of paragraphs, an index of titles, and an index of section headings, each of which can be searched for particular words or phrases. Browsing involves data structures created from metadata that the user can examine: lists of authors, lists of titles, lists of dates, hierarchical classification structures, and so on. Data structures for both browsing and searching are built according to instructions in a configuration file, which controls both building and serving the collection.

Rich browsing facilities can be provided by manually linking parts of documents together and building explicit indexes and tables of contents. However, manually-created linking becomes difficult to maintain, and often falls into disrepair when a collection expands. The Greenstone software takes a different tack: it facilitates *maintainability* by creating all searching and browsing structures automatically from the documents themselves. No links are inserted by hand. This means that when new documents in the same format become available, they can be added automatically. Indeed, for some collections this is done by processes

that wake up regularly, scout for new material, and rebuild the indexes—all without manual intervention.

Collections comprise many documents: thousands, tens of thousands, or even millions. Each document may be hierarchically organized into *sections* (subsections, sub-subsections, and so on). Each section comprises one or more *paragraphs*. Metadata such as author, title, date, keywords, and so on, may be associated with documents, or with individual sections of documents. This is the raw material for indexes. It must either be provided explicitly for each document and section (for example, in an accompanying spreadsheet) or be derivable automatically from the source documents. Metadata is converted to Dublin Core and stored with the document for internal use.

In order to accommodate different kinds of source documents, the software is organized so that "plugins" can be written for new document types. Plugins exist for plain text documents, HTML documents, email documents, and bibliographic formats. Word documents are handled by saving them as HTML; PostScript ones by applying a preprocessor [12]. Specially written plugins also exist for proprietary formats such as that used by the BBC archives department. A collection may have source documents in different forms: it is just a matter of specifying all the necessary plugins. In order to build browsing indexes from metadata, an analogous scheme of "classifiers" is used: classifiers create indexes of various kinds based on metadata. Source documents are brought into the Greenstone system through a process called *importing*, which uses the plugins and classifiers specified in the collection configuration file.

The system includes an "administrative" function whereby specified users can examine the composition of all collections, protect documents so that they can only be accessed by registered users on presentation of a password, and so on. Logs of user activity are kept that record all queries made to every Greenstone collection (though this facility can be disabled).

Although primarily designed for Internet access over the World-Wide Web, collections can be made available, in precisely the same form, on CD-ROM. In either case they are accessed through any Web browser. Greenstone CD-ROMs operate on a standalone PC under Windows 3.X, 95, 98, and NT, and the interaction is identical to accessing the collection on the Web—except that response is faster and more predictable. The requirement to operate on early Windows systems is a significant practical impediment to the software design, but is crucial for many users—particularly those in underdeveloped countries seeking access to humanitarian aid collections. If the PC is connected to a network (intranet or Internet), a custom-built Web server provided on each CD makes exactly the same information available to others through their standard Web browser. The use of compression ensures that the greatest possible volume of information can be packed on to a CD-ROM.

The collection-serving software operates under Unix and Windows NT, and works with standard Web servers. A flexible process structure allows different collections to be served by different computers, yet be presented to the user in the same way, on the same Web page, as part of the same digital library [10].

Existing collections can be updated and new ones brought on-line at any time, without bringing the system down; the process responsible for the user interface will notice (through periodic polling) when new collections appear and add them to the list presented to the user.

4 Browsing in the Digital Library of the Future

Current digital library systems often contain handcrafted indexes and links to provide different entry points into the information, and to bind it together into a coherent whole. This can produce high-quality, focused collections—but it is basically unscalable. Excellent new material will, of course, continue to be produced using manual techniques, but it is infeasible to suppose that the mass of existing, archival material will be manually "converted" into high-quality digital collections. The only scalable solution that is used currently for amorphous information collections is the ubiquitous search engine—but browsing is poorly supported by standard search engines. They operate at the wrong level, indexing words whereas people think in terms of topics, and returning individual documents whereas people often seek a more global view.

Suppose you are browsing a large collection of information such as a digital library—or a large Web site. Searching is easy, if you know what you are looking for—and can express it as a query at the lexical level. But current search mechanisms are not much use if you are not looking for a specific piece of information, but are generally exploring the collection. Studies of browsing have shown that it is a rich and fundamental human information behavior, a multifaceted and multidimensional human activity [3]. But it is not well-supported for large digital collections.

We look at three browsing interfaces that capitalize on automatically-generated phrases and keyphrases for a document collection. The first of these is a phrase-based browser that is specifically designed to support subject-index-style browsing of large information collections. The second uses more specific phrases and concentrates on making it convenient to browse closely-related documents. The third is a workbench that facilitates skimming, reading, and writing documents within a digital library—a qualitatively different experience from working in a library today. All three are based on phrases and keyphrases extracted automatically from the document text itself.

4.1 Emulating Subject Indexes: Phrase Browsing

Phrases extracted automatically from a large information collection form an excellent basis for browsing and accessing it. We have developed a phrase-based browser that acts an interactive interface to a phrase hierarchy that has been extracted automatically from the full text of a document collection. It is designed to resemble a paper-based subject index or thesaurus.

We illustrate the application of this scheme to a large Web site: that of the United Nations *Food and Agriculture Organization* (`www.fao.org`), an international organization whose mandate is to raise levels of nutrition and standards

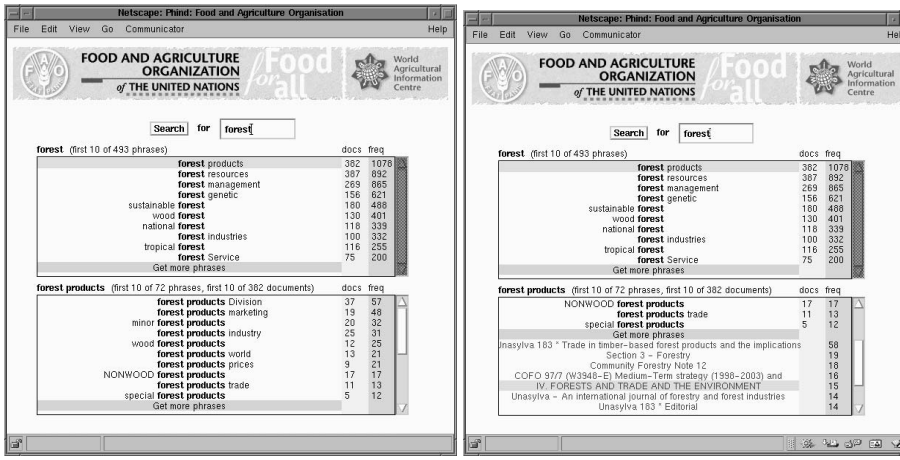


Fig. 4. (a) Browsing for information about *forest*, (b) Expanding on *forest products*

of living, to improve agricultural productivity, and to better the condition of rural populations. The site contains 21,700 Web pages, as well as around 13,700 associated files (image files, PDFs, etc). This corresponds to a medium-sized collection of approximately 140 million words of text. It exhibits many problems common to large, public Web sites. It has existed for some time, is large and continues to grow rapidly. Despite strenuous efforts to organize it, it is becoming increasingly hard to find information. A search mechanism is in place, but while this allows some specific questions to be answered it does not really address the needs of the user who wishes to browse in a less directed manner.

Figure 4a shows the phrase browsing interface in use. The user enters an initial word in the search box at the top. On pressing the *Search* button the upper panel appears. This shows the phrases at the top level in the hierarchy that contain the search word-in this case the word *forest*. The list is sorted by phrase frequency; on the right is the number of times the phrase appears, and to the left of that is the number of documents in which it appears.

Only the first ten phrases are shown, because it is impractical with a Web interface to download a large number of phrases, and many of these phrase lists are very large. At the end of the list is an item that reads *Get more phrases* (displayed in a distinctive color); clicking this will download another ten phrases, and so on. A scroll bar appears to the right for use when more than ten phrases are displayed. The number of phrases appears above the list: in this case there are 493 top-level phrases that contain the term *forest*.

So far we have only described the upper of the two panels in Figure 4a. The lower one appears as soon as the user clicks one of the phrases in the upper list. In this case the user has clicked *forest products* (that is why that line is highlighted in the upper panel) and the lower panel, which shows phrases containing the text *forest products*, has appeared.

If one continues to descend through the phrase hierarchy, eventually the leaves will be reached. A leaf corresponds to a phrase that occurs in only one document of the collection (though the phrase may appear several times in that document). In this case, the text above the lower panel shows that the phrase *forest products* appears in 72 phrases (the first ten are shown), and, in addition, appears in a unique context in 382 documents. The first ten of these are available too, though the list must be scrolled down to make them appear in the visible part of the panel. Figure 4b shows this. In effect, the panel shows a phrase list followed by a document list. Either of these lists may be null (in fact the document list is null in the upper panel, because the word *forest* appears only in other phrases or in individual unique contexts). The document list displays the titles of the documents.

It is possible, in both panels of Figures 4a and b, to click *Get more phrases* to increase the number of phrases that are shown in the list of phrases. It is also possible, in the lower panels, to click *Get more documents* (again it is displayed at the end of the list in a distinctive color, but to see that entry it is necessary to scroll the panel down a little more) which increases the number of documents that are shown in the list of documents.

Clicking on a phrase will expand it. The page holds only two panels, and if a phrase in the lower panel is clicked the contents of that panel move up into the top one to make space for the phrase's expansion. Alternatively, clicking on a document will open that document in a new window. In fact, the user in Figure 4b has clicked on *IV FORESTS AND TRADE AND THE ENVIRONMENT*, and this brings up a Web page with that title. As Figure 4b indicates, that page contains 15 occurrences of the phrase *forest products*.

We have experimented with several different ways of creating a phrase hierarchy from a document collection. An algorithm called SEQUITUR builds a hierarchical structure containing every single phrase that occurs more than once in the document collection [11]. We have also worked on a scheme called KEA which extracts keyphrases from scientific papers. This produces a far smaller, controllable, number of phrases per document [5]. The scheme that we use for the interface in Figure 4 is an amalgam of the two techniques [13].

The phrases extracted represent the topics present in the *Food and Agriculture Organization* site, as described in the terminology of the document authors. We have investigated how well this set of phrases matches the standard terminology of the discipline by comparing the extracted phrases with phrases used by the AGROVOC agricultural thesaurus. There is a substantial degree of overlap between the two sets of phrases, which provides some confirmation of the quality of the extracted phrases as subject descriptors.

4.2 Improved Browsing Using Keyphrase Indexes

Another a new kind of search interface that is explicitly designed to support browsing is based on keyphrases automatically extracted from the documents [6] using the KEA system [5]. A far smaller number of phrases are selected than in

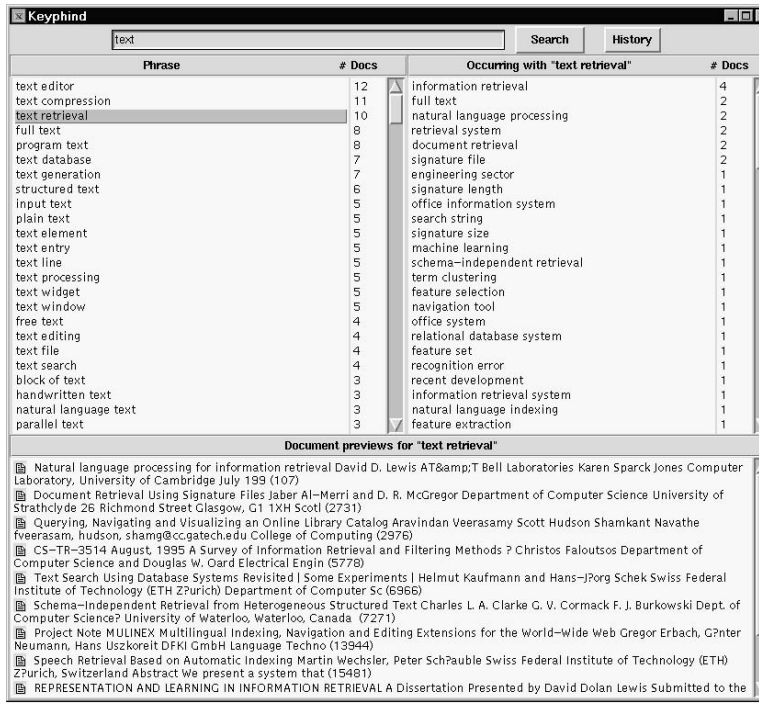


Fig. 5. Browsing a keyphrase index to find out about topics involving *text*

the system described above—only four or five per document. The automatically-extracted keyphrases form the basic unit of both indexing and presentation, allowing users to interact with the collection at the level of topics and subjects rather than words and documents. The system displays the topics in the collection, indicates coverage in each area, and shows all ways a query can be extended and still match documents.

The interface is shown in Figure 5. A user initiates a query by typing words or phrases and pressing the *Search* button, just as with other search engines. However, what is returned is not a list of documents, but a list of keyphrases containing the query terms. Since all phrases in the database are extracted from the source documents, every returned phrase represents one or more documents in the collection. Searching on the word *text*, for example, returns a list of phrases including *text editor* (a keyphrase for twelve documents), *text compression* (eleven documents), and *text retrieval* (ten documents), as shown in Figure 5. The phrase list provides a high-level view of the topics represented in the collection, and indicates, by the number of documents, the coverage of each topic.

Following the initial query, a user may choose to refine the search using one of the phrases in the list, or examine a topic more closely. Since they are derived from the collection itself, any further search with these phrases is guaranteed to

produce results—and furthermore, the user knows exactly how many documents to expect. To examine the documents associated with a phrase, the user selects it from the list, and previews of documents for which it is a keyphrase are displayed in the lower panel of the interface. Selecting any preview shows the document's full text.

Experiments with users show that this interface is superior to a traditional search system for answering particular kinds of questions: evaluating collections ("what's in this collection"), exploring areas ("what subtopics are available in area X"), and general information about queries ("what kind of queries will succeed in area X", "how can I specialize or generalize my query"). Note that many of these questions are as relevant to librarians as they are to library users. However, this mechanism is not intended to replace conventional search systems for specific queries about specific documents.

4.3 Reading and Writing in a Digital Library

A third prototype system, developed by Jones [7], shows how phrases can assist with skimming, reading, and writing documents in the digital library. It uses the keyphrases extracted from a document collection as link anchors to point to other documents. When reading a document, the keyphrases in it are highlighted. When writing one, phrases are dynamically linked, and highlighted, as you type.

Figure 6 shows the interface. To the left is the document being examined (read or authored); in the center is the keyphrase pane; and to the right is the library access pane. Keyphrases that appear in documents in the collection are highlighted; this facilitates rapid skimming of the content because the darker text points out items that users often highlight manually with a marker pen. Different gray levels reflect the "relevance" of the keyphrase to the document, and the user can control the intensity to match how they skim. Each phrase is hyperlinked, using multiple-destination links, to other documents for which it is a keyphrase (the anchor is the small spot that follows the phrase). The center panel shows all the keyphrases that appear in this document, with their frequency and the number of documents in the library for which they are keyphrases. Controls are available to sort the list in various different ways. Some of these phrases have been selected by the user, and on the right is a ranked list of items in the library that contain them as keyphrases—ranked according to a special metric designed for use with keyphrases.

With this interface, hurried readers can skim a document by looking at the highlighted phrases. In-depth readers can instantly access other relevant documents (including dictionaries or encyclopaedias). They can select a subset of relevant phrases and instantly have the library searched on that set. Writers—as they type—can immediately gain access to documents that are relevant to what they are writing.

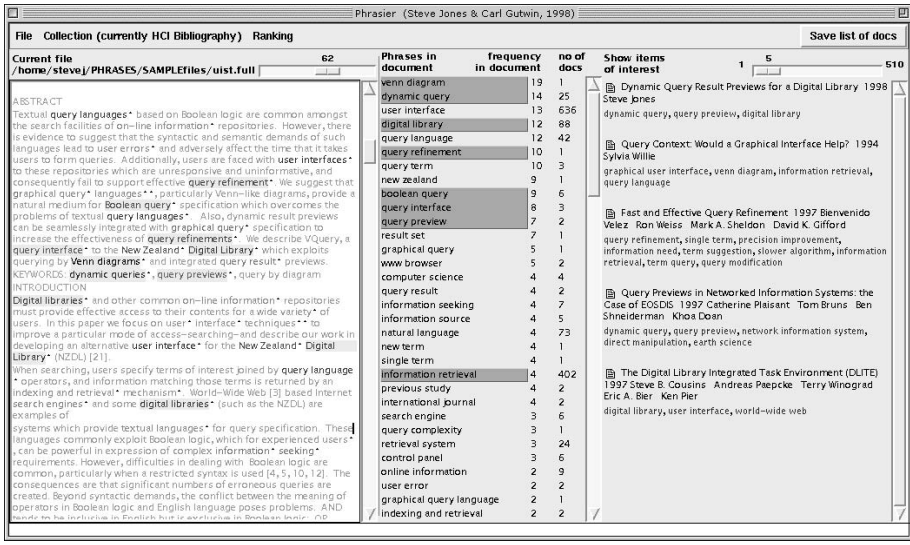


Fig. 6. Working on a paper inside the digital library

5 Conclusion

Digital libraries have finally arrived. They are different from the World Wide Web: libraries are focused collections, and it is the act of selection that gives them focus. For many practical reasons (including copyright, and the physical difficulty of digitization), digital libraries will not vie with archival national collections, not in the foreseeable future. Their role is in specialist, targeted collections of information.

Established libraries of printed material have sophisticated and well-developed human and computer-based interfaces to support their use. But they are not well integrated for working with computer tools: a bridging process is required. Information workers can immerse themselves physically in the library, but they cannot take with them their tasks, tools, and desktop workspaces. The digital library will be different: we will work "inside" it in a sense that it totally new.

But even for a focused collection, creating a high-quality digital library is a highly labor-intensive process. To provide the richness of access and interconnection that makes a digital library comfortable requires enormous editorial effort. And when the collection changes, maintenance becomes an overriding issue. Fortunately, techniques of text mining are emerging that offer the possibility of automatic identification of semantic items from plain text. Carefully-constructed user interfaces can take advantage of the information that they generate to provide a library experience that is qualitatively different from a physical library—not just in access and convenience, but in terms of the quality of browsing and information accessibility. Tomorrow, digital libraries will put the right information at your fingertips.

Acknowledgments

Many thanks to members of the New Zealand Digital Library project, whose work is described in this paper, particularly David Bainbridge, George Buchanan, Stefan Boddie, Rodger McNab and Bill Rogers who built the Greenstone system; Eibe Frank and Craig Nevill-Manning who worked on phrase extraction; Carl Gutwin, Steve Jones and Gordon Paynter who created phrase-based interfaces; and Mark Apperley, Sally Jo Cunningham, Te Taka Keegan and Malika Mahoui for their help and support. Rob Akscyn, Michel Loots and Harold Thimbleby also made valued contributions.

References

1. Akscyn, R.M. and Witten, I.H. (1998) "Report on First Summit on International Cooperation on Digital Libraries." `ks.com/idla-wp-oct98`.
2. Bush, V. (1947) "As we may think." *The Atlantic Monthly*, Vol. 176, No. 1, pp. 101–108.
3. Chang, S.J. and Rice, R.E. (1993) "Browsing: A multidimensional framework." *Annual Review of Information Science and Technology*, Vol. 28, pp. 231–276.
4. Fox, E. (1998) "Digital library definitions." `ei.cs.vt.edu/~fox/dlib/def.html`.
5. Frank, E., Paynter, G.W., Witten, I.H., Gutwin, C. and Nevill-Manning, C. (1999) "Domain-specific keyphrase extraction." *Proc Int Joint Conference on Artificial Intelligence*, Stockholm, Sweden, pp. 668–673.
6. Gutwin, C., Paynter, G., Witten, I.H., Nevill-Manning, C., and Frank, E. (in press) "Improving browsing in digital libraries with keyphrase indexing." *Decision Support Systems.*
7. Jones, S. (1999) "Phrasier: An interactive system for linking and browsing within document collections using keyphrases." *Proc. Interact 99: Seventh IFIP Conference On Human-Computer Interaction*, Edinburgh, Scotland, pp. 483–490.
8. Licklider, J.C.R. (1960) "Man-computer symbiosis." *IRE Trans HFE-1*, pp. 4–11.
9. Mann, T. (1993) *Library research models.* Oxford University Press, NY.
10. McNab, R.J., Witten, I.H. and Boddie, S.J. (1998) "A distributed digital library architecture incorporating different index styles." *Proc. IEEE Advances in Digital Libraries*, Santa Barbara, CA, pp. 36–45.
11. Nevill-Manning, C.G., Witten, I.H. and Paynter, G.W. (1997) "Browsing in digital libraries." *Proc. ACM Digital Libraries 97*, pp. 230–236.
12. Nevill-Manning, C.G., Reed, T., and Witten, I.H. (1998) "Extracting text from PostScript." *Software—Practice and Experience*, Vol. 28, No. 5, pp. 481–491.
13. Paynter, G.W., Witten, I.H., Cunningham, S.J. and Buchanan, G. (2000) "Scalable browsing for large collections: A case study." *Proc. Digital Libraries 2000*, Austin, TX.
14. Wells, H.G. (1938) *World Brain*. Doubleday, NY.
15. Witten, I.H., Moffat, A., and Bell, T.C. (1999) *Managing Gigabytes: Compressing and indexing documents and images.* Morgan Kaufmann, San Francisco.
16. Witten, I.H., Bray, Z., Mahoui, M. and Teahan, W. (1999) "Text mining: A new frontier for lossless compression." *Proc. Data Compression Conference*, pp. 198–207.
17. Witten, I.H., McNab, R.J., Boddie, S.J. and Bainbridge, D. (2000) "Greenstone: a comprehensive open-source digital library software system." *Proc. ACM Digital Libraries*, San Antonio, TX.

Algorithmic Aspects of Speech Recognition: A Synopsis

Adam L. Buchsbaum[1] and Raffaele Giancarlo[2⋆]

[1] AT&T Labs, Shannon Laboratory
180 Park Ave., Florham Park, NJ 07932, USA
alb@research.att.com
[2] Dipartimento di Matematica ed Applicazioni, Universitá di Palermo
Via Archirafi 34, 90123 Palermo, Italy
raffaele@altair.math.unipa.it

Abstract. Speech recognition is an area with a sizable literature, but there is little discussion of the topic within the computer science algorithms community. Since many of the problems arising in speech recognition are well suited for algorithmic studies, we present them in terms familiar to algorithm designers. Such cross fertilization can breed fresh insights from new perspectives.

This material is abstracted from A. L. Buchsbaum and R. Giancarlo, *Algorithmic Aspects of Speech Recognition: An Introduction*, ACM Journal of Experimental Algorithmics, Vol. 2, 1997, http://www.jea.acm.org.

1 Introduction

Automatic recognition of human speech by computers (ASR, for short) has been an area of scientific investigation for over forty years. (See, for instance, Waibel and Lee [21].) Its nature is inherently interdisciplinary, because it involves expertise and knowledge coming from such diverse areas as signal processing, artificial intelligence, statistics, and natural languages. Intuitively, one can see the core computational problems in ASR in terms of searching large, weighted, spaces. They therefore naturally lend themselves to the formulation of algorithmic questions. Historically, however, advances in ASR and in design of algorithms have found different places in the literature and in fact mostly address separate audiences (with a few exceptions [9]). This is unfortunate, because ASR challenges algorithm designers to find solutions that are asymptotically efficient and also practical in real cases.

The general problem areas that are involved in ASR—in particular, graph searching and automata manipulation—are well known to and have been extensively studied by algorithms experts, sometimes resulting in very tight theoretical bounds and even good practical implementations (e.g., shortest path finding

⋆ Part of this work was done while the author was an MTS at AT&T Bell Labs and continued while visiting AT&T Labs. Part of the author's research is supported by the Italian Ministry of Scientific Research, Project "Bioinformatica e Ricerca Genomica."

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 27–32, 2000.

and finite state automata minimization). The manifestations of these problems in ASR, however, are so large as to defy those solutions. The result is that most of the progress in speech recognition to date is due to clever heuristic methods that solve special cases of the general problems. Good characterizations of these special cases, as well as theoretical studies of their solutions, remain for the most part lacking.

While ASR is well suited for exploration and experimentation by algorithm theorists and designers, there are obvious obstacles in learning the formalisms and the design and experimental methodologies developed by the ASR community. The aim of this abstract is to present, at a very high level, a few research areas in ASR, extracted from a sizable body of literature in that area. This work abstracts an earlier paper [3], in which we present a more detailed view of ASR from an algorithmic perspective. Here we give only a high-level overview of ASR and some relevant problems with a technical formalism familiar to algorithm designers. We refer the reader to the previous paper [3] for a more detailed presentation of these research areas, as well as a proper identification of the context from which they have been abstracted.

2 Algorithmic Research Areas in ASR

We present ASR in terms of the *maximum likelihood paradigm* [1], which has become dominant in the area. Fig. 1 gives a block diagram and a short description of each module of a speech recognizer. Intuitively, one can think of a *lattice* as a data structure representing a set of strings.

Hidden Markov models (*HMM*s, for short) are basic building blocks of ASR systems [17]. *HMM*s are most commonly used to model *phones*, which are the basic units of sound to be recognized. For instance, they are the models underlying the acoustic-phonetic recognizer of Fig. 1.

A great deal of research has been invested in good algorithms for training the models and for their subsequent use in ASR systems. In abstract terms, an *HMM* is a finite state machine that takes as input a string and produces a probability for that string matching the model. The probability gives the likelihood that the input string actually belongs to the set of strings on which the *HMM* has been trained. Every path through the machine contributes to the probability assigned to an input string. *The forward procedure* [17], a dynamic programming algorithm, computes that probability in time proportional to the product of the length of the string and the size of the *HMM*. In algorithmic terms, the two parameters of interest here are the speed of the algorithm performing the "matching" and the size of the model. Given that the *HMM*s used in ASR have a very special topology, it is quite natural to consider the two following areas.

Research Area 1. *Devise faster methods to compute the probability that an HMM matches a given input string. In particular, can the topology of the HMM be exploited towards this end?*

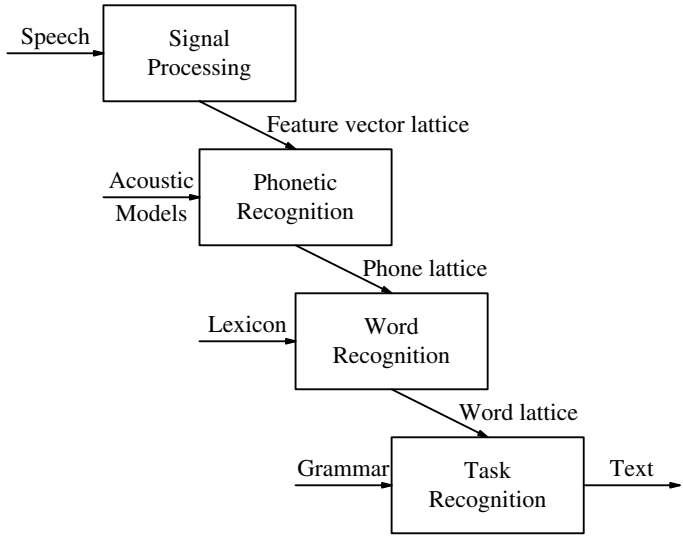


Fig. 1. Block diagram of a speech recognizer. Input speech is digitized into a sequence of feature vectors. An acoustic-phonetic recognizer transforms the feature vectors into a time-sequenced lattice of phones. A word recognition module transforms the phone lattice into a word lattice, with the help of a lexicon. Finally, in the case of continuous or connected word recognition, a grammar is applied to pick the most likely sequence of words from the word lattice.

Research Area 2. *Devise algorithms to reduce the size of an HMM. This is analogous to the determinization and minimization problems on finite-state automata, which will be discussed later.*

A simple way to speed the estimation of the probability of a string matching the model is to resort to an approximation: find the most probable path in the model that generates the string (rather than the sum of the probabilities of all the matching paths). One can find the desired path using the *Viterbi algorithm* [20], which computes a dynamic programming recurrence referred to as the *Viterbi equation*. Close examination reveals that the Viterbi equation is a variant of the Bellman-Ford equations for computing shortest paths in unweighted graphs [6], in which the weight of an edge is a **function of time**. That is, the weight depends on the time instant (the position being matched in the string) in which we are actually using the edge. For this latter class of shortest path problems, not much is known [13].

Research Area 3. *Devise faster algorithms to solve the Viterbi equation. As with Research Area 1, investigate how to characterize and exploit the particular graph topologies that arise in speech recognition.*

Another avenue of research is to compute iteratively better approximations to the final values of the Viterbi equation, analogously to the scaling algorithms used for standard shortest path problems [7].

Research Area 4. *Devise an analogue of the scaling technique that would apply to the computation of the Viterbi Equation.*

The models describing the ASR process tend to be large and highly structured. One can exploit that structure by using heuristics to prune part of the search space. In these settings, the A^* algorithm from artificial intelligence is often used [8]. We are now at the lexicon or grammar level shown in Fig. 1. In general, the A^* algorithm will find an "optimal solution" only when the heuristic used is *admissible*. Unfortunately, most of the heuristics used in ASR cede admissibility for speed. No analytic guarantees on the quality of the solutions are available.

Research Area 5. *Investigate the potential for admissible heuristics that will significantly speed computation of the A^* algorithm, or determine how to measure theoretically the error rates of fast but inadmissible heuristics.*

Pereira et al. [14,15] have recently devised a new paradigm to describe the entire ASR process. They formalize it in terms of *weighted transductions*. In fact, one can see the entire process in Fig. 1 as a cascade of translations of strings in one "language" into strings of another. This new paradigm is far reaching: it gives (a) a modular approach to the design of the various components of an ASR system; and (b) a solid theoretic foundation to which to attach many of the problems that seemed solvable only by heuristics. Moreover, it has relevant impact on automata and transduction theory [12]. Essential to this new paradigm are the problems of determinizing and minimizing *weighted automata*. Despite the vast body of knowledge available in formal languages and automata theory, those two fundamental problems were open until Mohri solved them [11]. As one might guess, determinization of a weighted automaton is a process that can take at least exponential time and produce a similar increase in the number of states of the input automaton. It is remarkable that, when the determinization algorithm by Mohri is applied to weighted automata coming from ASR, the resulting deterministic automata are, in most cases, **smaller** than their nondeterministic inputs. Given the importance of reducing the size of the automata produced in ASR, the following two areas are extremely important from the practical point of view and also challenging from the theoretical point of view.

Research Area 6. *Characterize the essential properties of sequential weighted automata that permit efficient determinization.*

We separately [4] report partial progress in Research Area 6, resulting in an approximation strategy [5] that increases the space reduction of Mohri's algorithm when applied to weighted automata from ASR.

Research Area 7. *Unify the results of Mohri [10] and Breslauer [2] (provably optimal minimization for sequential transducers, a special class of transducers) with those of Roche [19] (minimization of transducers without proven asymptotic size reductions, but with good practical performance).*

In fact, minimization of transducers in general is an open area of research, although partial progress towards a theoretical foundation exists [18].

3 Sources of Code and Data

One important aspect of designing algorithms for ASR is experimentation. To be accepted, new algorithms must be compared to a set of standard benchmarks. Here we give a limited set of pointers to find code and data for proper assessments; we refer the reader to the full paper [3] for additional sources.

Commercially available speech recognition products are typically designed for use by applications developers rather than researchers. One product, though, called HTK, provides a more low-level toolkit for experimenting with speech recognition algorithms in addition to an application-building interface. It is available from Entropic Research Lab, Inc., at `http://www.entropic.com/htk/`.

To obtain the finite-state toolkit developed by Pereira et al. [14] one can send mail to `fsm@research.att.com`.

A large variety of speech and text corpora is available from the Linguistic Data Consortium (`http://www.ldc.upenn.edu/`). This service is not free, although membership in LDC reduces the cost per item. The following are some of the commonly used speech corpora available.

TIMIT	Acoustic-Phonetic Continuous Speech Corpora.
RM	Resource Management Corpora.
ATIS	Air Travel Information System.
CSR	Continuous Speech Recognition.
SWITCHBOARD	Switchboard Corpus of Recorded Telephone Conversations.

For the multilingual experimenter, there is the *Oxford Acoustic Phonetic Database* on CDROM [16]. It is a set of two CDs that contain digitized recordings of isolated lexical items plus isolated monophthongs from each of the following eight languages/dialects: American English, British English, French, German, Hungarian, Italian, Japanese, and Spanish.

References

1. L. R. Bahl, F. Jelinek, and R. L. Mercer. A maximum likelihood approach to continuous speech recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5:179–190, 1983.
2. D. Breslauer. The suffix tree of a tree and minimizing sequential transducers. *Theoretical Computer Science*, 191(1-2):131–44, 1998.

3. A. L. Buchsbaum and R. Giancarlo. Algorithmic aspects in speech recognition: An introduction. *ACM Journal of Experimental Algorithmics*, 2, 1997. `http://www.jea.acm.org`.
4. A. L. Buchsbaum, R. Giancarlo, and J. R. Westbrook. On the determinization of weighted automata. In *Proc. 25th International Colloquium on Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 482–93, 1998.
5. A. L. Buchsbaum, R. Giancarlo, and J. R. Westbrook. Shrinking language models by robust approximation. In *Proc. IEEE Int'l. Conf. on Acoustics, Speech, and Signal Processing '98*, volume 2, pages 685–8, 1998. To appear in *Algorithmica* as "An Approximate Determinization Algorithm for Weighted Finite-State Automata".
6. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1991.
7. H. N. Gabow. Scaling algorithms for network problems. *Journal of Computer and System Sciences*, 31:148–68, 1985.
8. P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science, and Cybernetics*, 4:100–7, 1968.
9. J. B. Kruskal and D. Sankoff, editors. *Time Wraps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, 1983.
10. M. Mohri. Minimization of sequential transducers. In *Proc. 5th Symposium on Combinatorial Pattern Matching*, volume 807 of *Lecture Notes in Computer Science*, pages 151–63, 1994.
11. M. Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 29:269–311, 1997.
12. M. Mohri, F. C. N. Pereira, and M. Riley. The design principles of a weighted finite-state transducer library. *Theoretical Computer Science*, 231:17–32, 2000.
13. A. Orda and R. Rom. Shortest path and minimum delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, 37:607–25, 1990.
14. F. Pereira and M. Riley. Speech recognition by composition of weighted finite automata. In *Finite-State Language Processing*. MIT Press, 1997.
15. F. Pereira, M. Riley, and R. Sproat. Weighted rational transductions and their application to human language processing. In *Proc. ARPA Human Language Technology Conf.*, pages 249–54, 1994.
16. J. B. Pickering and B. S. Rosner. *The Oxford Acoustic Phonetic Database on Compact Disk*. Oxford University Press, 1993.
17. L. Rabiner and B.-H. Juang. *Fundamentals of Speech Recognition*. Prentice Hall Signal Processing Series. Prentice Hall, Englewood Cliffs, NJ, 1993.
18. C. Reutenauer and M.-P. Schützenberger. Minimization of rational word functions. *SIAM Journal on Computing*, 20(4):669–85, 1991.
19. E. Roche. Smaller representations for finite-state transducers and finite-state automata. In *Proc. 6th Symposium on Combinatorial Pattern Matching*, volume 937 of *Lecture Notes in Computer Science*, pages 352–65, 1995.
20. A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IEEE Transactions on Information Theory*, IT-13:260–9, 1967.
21. A. Waibel and K.-F. Lee, editors. *Readings in Speech Recognition*. Morgan Kaufmann, 1990.

Some Results on Flexible-Pattern Discovery

Laxmi Parida

Bioinformatics and Pattern Discovery
IBM Thomas J. Watson Research Center
Yorktown Heights, NY10598, USA
parida@us.ibm.com

Abstract. Given an input sequence of data, a "rigid" pattern is a repeating sequence, possibly interspersed with "dont care" characters. In practice, the patterns or motifs of interest are the ones that also allow a variable number of gaps (or "dont care" characters): we call these the flexible motifs. The *number* of rigid motifs could potentially be exponential in the size of the input sequence and in the case where the input is a sequence of real numbers, there could be uncountably infinite number of motifs (assuming two real numbers are equal if they are within some $\delta > 0$ of each other). It has been shown earlier that by suitably defining the notion of maximality and redundancy, there exists only a *linear* (or no more than $3n$) number of irredundant motifs and a polynomial time algorithm to detect these irredundant motifs. Here we present a uniform framework that encompasses both rigid and flexible motifs with generalizations to sequence of sets and real numbers and show a somewhat surprising result that the number of irredundant flexible motifs still have a linear bound. However, the algorithm to detect them has a higher complexity than that of the rigid motifs.

1 Introduction

Given an input sequence of data, a "rigid" pattern is a repeating sequence, possibly interspersed with "dont care" characters. For example given a string $s = abcdaXcdabbcd$, $m = a.cd$ is a pattern that occurs twice in the data at positions 1 and 5 in s. The data could be a sequence of characters or sets of characters or even real values. In practice, the patterns or motifs of interest are the ones that also allow a variable number of gaps (or "dont care" characters): we call these the flexible motifs. In the above example, the flexible motif would occur three times at positions 1, 5 and 9. At position 9 the dot character represents two gaps instead of one. Flexible pattern is an extension of the *generalized regular pattern* described in [BJEG98] in the sense that the input here could also be a sequence of real numbers.

Pattern discovery in biomolecular data has been often closely intertwined with the problem of alignment of sequences [ZZ96], [Alt89], [CL88], [HTHI95], [HS88], [MSZM97], [Wat94], [LSB+93], [MVF96] sequences [Roy92], [JCH95], [NG94], [SV96], [WCM+96], [SNJ95], [BG98], [FRP+99], [GYW+97], [RFO+99], [RGFp99]. The reader may refer to [RFP+00] for a history in this regard: the

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 33–45, 2002.

paper traces the interesting journey from inception to the current research in pattern discovery [1]. [BJEG98] surveys approaches to the pattern discovery problem in biological applications and presents a systematic formal framework that enables an easy comparison of the power of the various algorithms/software system available: the focus is on the different algorithms and the class of patterns being handled. There also has been work in literature that deals primarily with pattern discovery [DT93,WCM+94,RF98,Cal00]. A very closely related class of problems is that of data mining and rule discovery [AS95,AMS+95,Bay92,DGM97]. However, in this paper we intend take a different route to understanding pattern discovery in the context of any application: be it in biomolecular data, market survey data, English usage, system log data etcetera. The reader should bear in mind that the definition of patterns/motifs emerge from the actual usage and the contexts in which they are used.

The task of discovering patterns must be clearly distinguished from that of matching a given pattern in a database. In the latter situation we know what we are looking for, while in the former we do not know what is being sought. Hence a pattern discovery algorithm must report *all* patterns. The total *number* of rigid motifs could potentially be exponential in the size of the input sequence and in the case where the input is a sequence of real numbers, there could be uncountably infinite number of motifs (assuming two real numbers are equal if they are within some $\delta > 0$ of each other). It has been shown earlier [Par99,PRF+00] that by suitably defining the notion of maximality and redundancy, there exists only a *linear* (or no more than $3n$) number of irredundant motifs. This is meaningful also from an algorithmic viewpoint, since a polynomial time algorithm has been presented to detect these irredundant motifs. This family of irredundant motifs is also very characteristic of the family of all the motifs: in applications such as multiple sequence alignment, it has been shown that the irredundant motifs suffice to obtain the alignment [PFR99a,PFR99b]. This bound on the number of useful motifs gives validation to motif-based approaches, since the total number of irredundant motifs does not explode. This result is of significance to most applications that use pattern discovery as the basic engine such as data mining, clustering and matching.

Flexible motifs obviously capture more information than rigid motifs and have been found to be very useful in different applications [BJEG98]. Here we use a uniform framework that encompasses both rigid and flexible motifs. We give some very natural definitions of maximality and redundancy for the flexible motifs and show a somewhat surprising result that the number of irredundant flexible motifs is still bounded by $3n$. However, the algorithm to detect them has a higher complexity than that of the rigid motifs.

In most of the previous work mentioned above, the discovery process is closely tied with the applications. Here we separate the two: this is a useful viewpoint since it allows for a better understanding of the problem and helps capture the commonality in the different applications. In almost all applications, the size of

[1] The reader may also visit `http://www.research.ibm.com/bioinformatics` for some current work in this area.

the data is very large which justifies the task of automatic or unaided pattern discovery. Since it is very hard to verify if all the patterns have been discovered, it becomes all the more important to assure that *all* patterns (depending on the definition) have been found. As the definition of a pattern becomes relaxed, the total number of candidate patterns increases substantially. For instance, the total number of rigid patterns with no "dont care" characters can be no more than n on an input of size n. However, if "dont care" characters are allowed the number of motifs is exponential in n. The next task, which is application specific, is to prune this large set of patterns by a certain boolean criterion $\mathcal{C}$. Usually $\mathcal{C}$ is based on a real valued function say $\mathcal{C}'$ and $\mathcal{C} = TRUE$ if and only if $\mathcal{C}' \geq t$, for some threshold t.If $\mathcal{C}(m) = TRUE$, motif m is of interest and if $\mathcal{C}(m) = FALSE$, m need not be reported. $\mathcal{C}$ is a measure of significance depending on the application. There are at least two issues that need to be recognized here: one is a lack of proper understanding (or a consensus amongst researchers) of the domain to define appropriate $\mathcal{C}$ and the other is that even if $\mathcal{C}$ is overwhelmingly acceptable, the guarantee that *all* patterns m with $\mathcal{C}(m) = TRUE$ have been detected is seldom provided. The former issue is a debate that is harder to resolve and the latter exists because of the inherent difficulty of the models (function $\mathcal{C}$).

In our study of the problem in the following paragraphs, we will assume that to apply $\mathcal{C}$, we have *all* the patterns at our disposal. We will focus on the task of trying to obtain the patterns in a modelless manner. However, $\mathcal{C}$ may be applied in conjunction with our algorithm to obtain *only* those m's that satisfy $\mathcal{C}(m) = TRUE$.

In the rest of the paper the terms pattern and motif will be used interchangeably.

2 Basics

Let s be a sequence on an alphabet Σ, '.' $\notin \Sigma$. A character from Σ, say σ, is called a solid character and '.' is called a "dont care" character. For brevity of notation, if x is a sequence, then $|x|$ denotes the length of the sequence and if x is a set of elements then $|x|$ denotes the cardinality of the set. The j^{th} $(1 \leq j \leq |s|)$ character of the sequence is given by $s[j]$.

Definition 1. *($\sigma_1 \prec, =, \preceq \sigma_2$) If σ_1 is a "dont care" character then $\sigma_1 \prec \sigma_2$. If both σ_1 and σ_2 are identical characters in Σ, then $\sigma_1 = \sigma_2$. If either $\sigma_1 \prec \sigma_2$ or $\sigma_1 = \sigma_2$ holds, then $\sigma_1 \preceq \sigma_2$.*

Definition 2. *(Annotated Dot Character, $.^x$) An annotated "." character is written as $.^x$ where x is a set of positive integers $\{x_1, x_2, \ldots, x_k\}$ or an interval $x = [x_l, x_u]$, representing all integers between x_l and x_u including x_l and x_u.*

To avoid clutter, the annotation superscript x, in the rest of the paper, will be an integer interval.

Definition 3. *(Realization) Let p be a string on Σ and annotated dot characters. If p' is a string obtained from p by replacing each annotated dot character $.^x$ by l dot characters where $l \in x$, p' is a realization of p.*

For example, if $p = a.^{[3,6]}b.^{[2,4]}cde$, then $p' = a...b...cde$ is a realization of p and so is $p'' = a...b.....cde$.

Definition 4. *(p Occurs at l) A string, p, on $\Sigma \cup \{.\}$, occurs at position l on s if $p[j] \preceq s[l+j]$ holds for $1 \leq j \leq |p|$. A string, p, on Σ and annotated dot characters, occurs at position l in s if there exists a realization p' of p that occurs at l.*

If p is flexible then p could possibly occur *multiple times* at a location on a string s. For example, if $s = axbcbc$, then $p = a.^{[1,3]}b$ occurs twice at position 1 as $\mathbf{a}x\mathbf{bc}bc$ and $\mathbf{a}xbc\mathbf{bc}$. Let $\#l$ denote the number of occurrences at location l. In this example $\#l = 2$.

Definition 5. *(Motif m, Location List $\mathcal{L}_m$) Given a string s on alphabet Σ and a positive integer k, $k \leq |s|$, a string m on Σ and annotated dot characters is a k-motif with location list $\mathcal{L}_m = (l_1, l_2, \ldots, l_p)$, if $m[1], m[|m|] \in \Sigma^2$ and m occurs at each $l \in \mathcal{L}_m$ with $p \geq k$.*

If m is a string on $\Sigma \cup \{.\}$, m is called a *rigid motif*, and if m is a string on Σ and annotated dot characters, where at least one annotation x in m represents more than one integer, then m is called a *flexible motif*.

Definition 6. *($m_1 \preceq m_2$) Given two motifs m_1 and m_2 with $|m_1| \leq |m_2|$, $m_1 \preceq m_2$ holds if for every realization m_1' of m_1 there exists a realization m_2' such that $m_1'[j] \preceq m_2'[j]$, $1 \leq j \leq |m_1|$.*

For example, let $m_1 = AB..E$, $m_2 = AK..E$ and $m_3 = ABC.E.G$. Then $m_1 \preceq m_3$, and $m_2 \not\preceq m_3$. The following lemmas are straightforward to verify.

Lemma 1. *If $m_1 \preceq m_2$, then $\mathcal{L}_{m_1} \supseteq \mathcal{L}_{m_2}$. If $m_1 \preceq m_2$ and $m_2 \preceq m_3$, then $m_1 \preceq m_3$.*

Definition 7. *(Sub-motifs of Motif m, $\mathcal{S}^{[j_1,j_2]}(m)$) Given a motif m let $m[j_1]$, $m[j_2]$, ... $m[j_l]$ be the l solid characters in the motif m. Then the sub-motifs of m are given as $\mathcal{S}^{[j_i,j_k]}(m)$, $1 \leq i < k \leq l$, which is obtained by dropping all the characters before (to the left of) j_i and all characters after (to the right of) j_k in m.*

Definition 8. *(Maximal Motif) Let p_1, p_2, ..., p_k be the motifs in a sequence s. Define $p_i[j]$ to be '.', if $j > |p_i|$. A motif p_i is maximal in composition if and only if there exists no p_l, $l \neq i$ with $\mathcal{L}_{p_i} = \mathcal{L}_{p_j}$ and $p_i \preceq p_l$. A motif p_i, maximal in composition, is also maximal in length if and only if there exists no motif p_j, $j \neq i$, such that p_i is is a sub-motif of p_j and $|\mathcal{L}_{m_i}| = |\mathcal{L}_{m_j}|$. A maximal motif is maximal both in composition and in length.*

[2] The first and last characters of the motif are solid characters; if "dont care" characters are allowed at the ends, the motifs can be made arbitrarily long in size without conveying any extra information.

Bounding the Total Number of Multiple Occurrences. The maximum number of occurrences at a position is clearly bounded by n, thus the total number of occurrences in the location list is bounded by n^2. Is this bound actually attained? The following is an example to show that such a bound is achieved. Let s be the input string that has $n/2$ a's followed by $n/2$ b's. Consider the motif $m = a.^{[n/2-1,n-1]}b$. At positions 1 to $n/2$, m occurs $n/2$ times in each. Thus the total number of occurrences is clearly $\Omega(n^2)$.

Can Density Constraint Reduce the Number of Motifs? Density could be specified by insisting that every realization of the motif has no more than d consecutive "dont care" characters. Is it possible to have a much smaller set of motifs under this definition (where d is a small constant). We show with the following example that the density constraint does not affect the worst case situations. Let the input string s have the following form:

$$ac_1c_2c_3baXc_2c_3bYac_1Xc_3bYYac_1c_2Xb$$

Then the maximal motifs (which are $2^{\Omega(\sqrt{n})}$ in number) are $a...b$, $a..c_3b$, $a.c_2.b$, $ac_1..b$, $a.c_2c_3b$, $ac_1.c_3b$, $ac_1c_2.b$. Let $d = 1$. Consider the input string in the last example. We construct a new motif by placing a new character Z between every two characters as follows:

$$aZc_1Zc_2Zc_3ZbaZXZc_2Zc_3ZbYaZc_1ZXZc_3ZbYYaZc_1Zc_2ZXZb$$

The length of the string just doubles, at most whereas the number of maximal motifs, that have no more than one consecutive dot character is at least as many as it was before.

See [Par98,Par99,PRF+00] for the motivation and a general description of the notion of redundancy. A formal definition is given below.

Definition 9. *(Redundant, Irredundant Motif) A maximal motif m, with location list $\mathcal{L}_m$, is redundant if there exist maximal motifs m_i, $1 \leq i \leq p$, $p \geq 1$, such that $\mathcal{L}_m = \mathcal{L}_{m_1} \cup \mathcal{L}_{m_2} \ldots \cup \mathcal{L}_{m_p}$ A maximal motif that is not redundant is called an* irredundant *motif.*

Notice that for a rigid motif $p > 1$ (p in the Definition 9) since each location list corresponds to a unique motif whereas for a flexible motif p could have a value 1. For example, let $s = axfygbapgrfb$. Then $m_1 = a.^{[1,3]}fb$, $m_2 = a.^{[1,3]}gb$, $m_3 = a....b$ with $\mathcal{L}_{m_1} = \mathcal{L}_{m_2} = \mathcal{L}_{m_3} = \{1, 7\}$. But m_3 is redundant since $m_3 \preceq m_1, m_2$. Also $m_1 \not\preceq m_2$ and $m_2 \not\preceq m_1$, hence both are irredundant.

Generating Operations. The redundant motifs need to be generated from the irredundant ones, if required. We define the following generating operations. Let m, m_1 and m_2 be motifs.

Prefix operator, $P^\delta(m)$, $1 < \delta < |m|$: This is a valid operation when δ is an integer and $m[\delta]$ is a solid character, since all the operations are closed under motifs. $P^\delta(m)$ is the string given by $m[1 \ldots \delta]$. For example, if $m = AB..CDE$,

then $P^3(m)$ is not a valid operation since $m[3]$ is a dot-character. Also, $P^5(m) = AB..C$.

Binary AND operator, $m_1 \bigoplus m_2$: $m = m_1 \bigoplus m_2$, where m is such that $m \preceq m_1, m_2$ and there exists no m' with $m \preceq m'$. For example if $m_1 = A.D.^{[2,4]}G$ and $m_2 = AB.^{[1,5]}FG$. Then, $m = m_1 \bigoplus m_2 = A...^{[2,4]}G$.

Binary OR operator, $m_1 \bigotimes m_2$: $m = m_1 \bigotimes m_2$, where m is such that $m_1, m_2 \preceq m$ and there exists no m' with $m' \preceq m$. For example if $m_1 = A..D..G$ and $m_2 = AB...FG$. Then, $m = m_1 \bigotimes m_2 = AB.D.FG$.

3 Bounding the Irredundant Flexible Motifs

Definition 10. *(Basis) Given a sequence s on an alphabet Σ, let $\mathcal{M}$ be the set of all maximal motifs on s. A set of maximal motifs $\mathcal{B}$ is called a basis of $\mathcal{M}$ iff the following hold: for each $m \in \mathcal{B}$, m is irredundant with respect to $\mathcal{B} - \{m\}$, and, let $\mathbf{G}(\mathcal{X})$ be the set of all the redundant maximal motifs generated by the set of motifs $\mathcal{X}$, then $\mathcal{M} = \mathbf{G}(\mathcal{B})$.*

In general, $|\mathcal{M}| = \Omega(2^n)$. The natural attempt now is to obtain as small a basis as possible.

Theorem 1. *Let s be a string with $n = |s|$ and let $\mathcal{B}$ be a basis or a set of irredundant motifs. Then $|\mathcal{B}| \leq 3n$.*

Proof. A proof for the special case when all the motifs are rigid, i.e., defined on $\Sigma \cup \{.\}$ appeared in [Par99,PRF+00]. The framework has been extended to incorporate the flexible motifs. Consider $\mathcal{B}^*$ ($\mathcal{B} \subseteq \mathcal{B}^*$) where the motifs in $\mathcal{B}^*$ are *not* maximal and *redundant*. Every position, $x \in \mathcal{L}_{m_1}, \mathcal{L}_{m_2}, \ldots, \mathcal{L}_{m_l}$, is assigned ON/OFF with respect to m as follows: If m_i, $1 \leq i \leq l$, is such that there exists no $j \neq i$, $1 \leq j \leq l$ so that $m_i \preceq m_j$ holds, then x is marked ON, otherwise it is marked OFF w.r.t. m_i. Further if $\mathcal{L}_{m_1} = \mathcal{L}_{m_2} = \ldots = \mathcal{L}_{m_l}$, then the position is marked ON w.r.t. each of m_i, $1 \leq i \leq l$. We make the following claims due to this ON/OFF marking:

Claim. At the end of this step, every motif m that is *not* redundant, has at least one location $x \in \mathcal{L}_m$ marked ON w.r.t. m.

Proof. This follows directly from the definition of redundancy. Also a redundant motif m may not have any position marked ON w.r.t m. ◇

Definition 11. *($\mathcal{L}_a$ Straddles $\mathcal{L}_b$) A set $\mathcal{L}_a$ straddles a set $\mathcal{L}_b$ if $\mathcal{L}_a \cap \mathcal{L}_b \neq \phi$, $\mathcal{L}_a - \mathcal{L}_b \neq \phi$ and $\mathcal{L}_b - \mathcal{L}_a \neq \phi$.*

Notice that if $\mathcal{L}_a$ straddles $\mathcal{L}_b$, then $\mathcal{L}_b$ straddles $\mathcal{L}_a$.

Claim. If location x is marked ON w.r.t. motifs $m_1, m_2, \ldots, m_l$, where no two location lists are identical, then every pair of location lists $\mathcal{L}_{m_i}, \mathcal{L}_{m_j}$, $i \neq j$ must straddle.

Proof. Assume this does not hold, then $\mathcal{L}_{m_i} \subseteq \mathcal{L}_{m_j}$, for some i and j. In that case the location x is marked OFF w.r.t. m_j which is a contradiction. ◇

For each motif m, define $c(m)$ to be the charge which is a positive integer. This is initialized to 1 for every motif. In the counting process, when there is difficulty in accounting for a motif m, a charge or count for m is assigned at some other other motif m': thus m' would account for itself and all the other motifs whose charge it is carrying (thus m' is the *banker*, as defined in the next step, for all these other motifs). For each motif m, define $B(m)$ to be the *banker* of m, which is a motif m' that is carrying the charge for m. For each m, initialize $B(m) = m$. Every motif is marked LIVE/DEAD. At the initialization step every motif that is not redundant (see Claim 3) is marked LIVE. If there exists a position x that is marked ON w.r.t only one LIVE motif m, m is marked DEAD. Repeat this process until no more motifs can be marked DEAD.

In some sense, every DEAD motif at this stage is such that there is a unique position (x of last paragraph), that can be uniquely assigned to it. The number of DEAD motifs $\leq n$.

We begin by introducing some more definitions.

Definition 12. *(Instance) An* instance *of a realization of a motif m is the motif at some location $x \in \mathcal{L}_m$ on the input string s.*

For example, let $s = abccdabed$ and let $m = ab.^{[1,2]}d$. Then one instance of m on s, shown in bold, is **abccd***abed* and the other instance is *abccd***abed**. The solid characters in the instances of m, shown with a bar, are as follows: $\overline{\mathbf{abccd}}abed$ in the first and $abcd\overline{\mathbf{abed}}$ in the second.

Definition 13. *(i-Connected) An instance of a realization of a motif m_1 is i-connected to an instance of a realization of a motif m_2 if the two instances have at least i common solid characters.*

Let $\bar{m}_a^x$ be an instance of m_a where $x \in \mathcal{L}_{m_a}$. To avoid clutter we refer to an instance of motif m_a simply as $\bar{m}_a$.

Lemma 2. *Consider an instance each of a set of motifs $m_1, m_2, \ldots, m_l$ such that for that instance of the realization of m_i, the starting position $x \in \mathcal{L}_{m_i}$ is marked ON w.r.t m_i, $1 \leq i \leq l$, and, for every motif m_i there exists a realization of motif m_j, $j \neq i$, $1 \leq i, j \leq l$, such that the two instances are 2-connected, then there exist* distinct *positions $j_1, j_2, \ldots, j_l$ on the input string s, with the corresponding positions, $j'_1, j'_2, \ldots, j'_l$ such that $m_1[j'_1]$, $m_2[j'_2]$, …, $m_l[j'_l]$ are solid characters.*

Proof. Assume this does not hold, then there exists instances of realizations of motifs m_{j_a}, m_{j_b}, as m'_{j_a}, m'_{j_b} respectively $1 \leq j_a, j_b \leq l$ with $m'_{j_b} \preceq m'_{j_a}$. Consider m''_{j_a}, a realization of the sub-motif of m'_{j_a} which starts at the starting position on m_{j_b} and ends at the ending position of m_{j_b}. If the position w.r.t. m''_{j_a} is ON, it is a contradiction since then the position at which m_{j_b} is incident must be marked OFF. However, if the position w.r.t. m''_{j_a} is OFF, then there exists an instance of a realization of m_{j_c}, m'_{j_c} such that $m''_{j_a} \preceq m'_{j_c}$ But $m'_{j_b} \preceq m'_{j_c}$ and both are ON, which is again a contradiction. ◇

Next, we define an *operational connectedness* on ON-marked instances of LIVE motifs m_a and m_b, called the *o-connectedness* which holds if $\bar{m}_a$ is 2-connected to $\bar{m}_b$, or. there exists $\bar{m}_c$, where the instance is ON-marked w.r.t LIVE motif m_c, and $\bar{m}_a$ is o-connected to $\bar{m}_c$ and $\bar{m}_c$ is o-connected to $\bar{m}_b$.

Lemma 3. *o-connectedness is an equivalence relation.*

Proof. It can be easily verified that o-connectedness is reflexive, symmetric and transitive. Thus all the ON-marked instances of the LIVE motifs can be partitioned into equivalence classes. . ◇

Claim. Using Lemmas 2 and 3, every instance of a realization of a LIVE motif m_a has a solid character at position j_a associated with it. Let $D(\bar{m}_a) = S^{[j_a,|m_a|]}(m_a)$.

Charging Scheme. We next describe a charging (or counting) scheme by which we count the number of motifs. This is best described as an iterative process as follows.

While there exists position x on s such that x is marked ON w.r.t LIVE motifs $m_1, m_2, \ldots, m_l$, $l > 1$, do the following for $1 \leq i \leq l$:
1. Let $B(m_i) = D(\bar{m}_i)$ (see Step 1.3 and Claim 3).
2. $c(B(m_i)) = c(B(m_i)) + c(m_i)$ (see Step 1.2).
3. Mark m_i DEAD (see Step 1.4) [3].

Claim. The loop terminates.

Proof. At every iteration at least two distinct LIVE motifs are marked DEAD, hence the loop must terminate. ◇

Claim. At the end of the loop, all the LIVE motifs are such that for every pair m_i, m_j: $\mathcal{L}_{m_i}$ and $\mathcal{L}_{m_j}$ do not straddle and $m_j \not\preceq m_i$, without loss of generality.

Proof. The first condition holds obviously since otherwise the loop would not terminate since $x \in \mathcal{L}_{m_i} \cap \mathcal{L}_{m_j}$ would be marked ON w.r.t m_i and m_j. The second condition also hold obviously since if $m_j \not\preceq m_i$, then motif m_i is marked DEAD (Step 1). ◇

Next, we need to show that the charge $c(m)$ carried by every LIVE motif m at the end of the loop, can be accounted for by $\mathcal{L}_m$. In this context we make the following observation about the charge: the iterative assignment of charges to a motif has a tree structure. This is best described using an example: see Figure 1. Each level of the tree corresponds to an iteration in the loop. For instance, the top level in the left tree denotes that at iteration 1, $B(a.bxyc..ab) = B(xyxyc..ab) = B(c...dxyc..ab) = xyc..ab$ and $c(xyc..ab) = 1 + c(a.bxyc..ab) + c(xyxyc..ab) + c(c...dxyc..ab)$. At the end of this iteration motifs $a.bxyc..ab$, $xyxyc..ab$ and $c...dxyc..ab$ are marked DEAD. At the second iteration, $B(xyc..ab) = yc..ab$ and $c(yc..ab) = 1 + c(xyc..ab)$ and motif $xyc..ab$ is marked DEAD and so on.

[3] The only exception is made when $B(m_i) = m_i$. In this case m_i remains LIVE.

Claim. Let L denote the number of leaf nodes (nodes with no incoming edges) in the charge tree of motif m at the end of the while loop, then $|\mathcal{L}_m| \geq L$.

Proof. Such a claim holds since we know that by our choice of $B(.)$, if $B(m_1) = B(m_2) = \ldots = B(m_l) = m'$ then by Lemma 2 m' must have l distinct instances, each instance in a distinct equivalent class of realization of motif instances (Lemma 3). However, the instance of m' may not be distinct from each of these instances; hence the non-leaf nodes may not be accounted for but the leaf nodes are. Hence $|\mathcal{L}_m| \geq L$. ◇

At an iteration if a motif m is charged by more than one motif (or in the charge tree, the node has more than one incident edge), m is certainly maximal. However if it is charged by exactly one motif then it may or may not be maximal; if it is maximal, it *must* have an extra instance. We use the following folk-lore lemma to bound the size of I, the number of non-leaf nodes in the charge-tree.

Lemma 4. *Given a tree $\mathcal{T}$, where each node, except the leaf nodes, must have at least two children, the number of non-leaf nodes, I is no more than the number of leaf nodes L.*

We are not interested in counting non-maximal motifs and these are the only motifs that contribute to a single child for a node in a tree. Thus the number of maximal motifs that were marked LIVE at the start of Step 2 is no more than $2n$, using Claim 3 and Lemma 4.

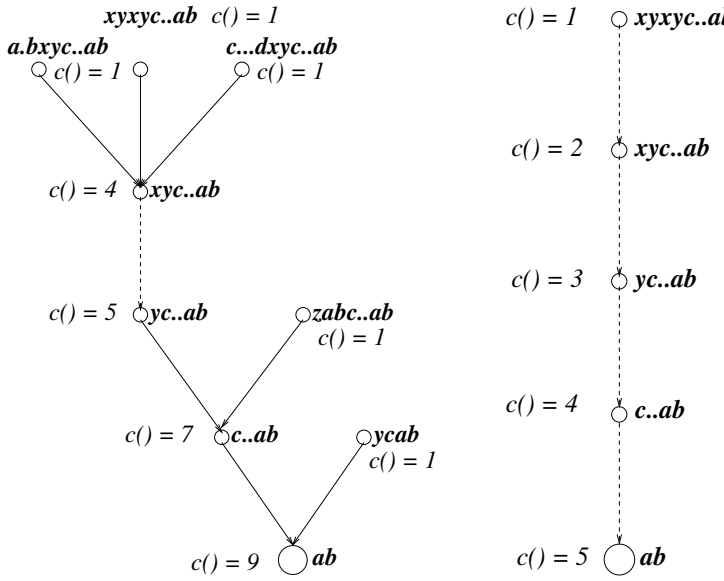


Fig. 1. Two examples showing the different steps in the assignment of charge to motif *ab*: Every level of the tree corresponds to an iteration in the while loop. The dashed edge indicates that the motif at the "to" end of the edge could possibly be non-maximal.

Using Step 1.4, we have number of maximal and non-redundant motifs $\leq (n + 2n)$. This concludes the proof of the theorem. Δ

Corollary 1. *Given a string s, the basis $\mathcal{B}$ is unique.*

Assume the contrary that there exists two distinct bases, $\mathcal{B}$ and $\mathcal{B}'$. Without loss of generality, let $m \in \mathcal{B}$ and $m \notin \mathcal{B}'$. Then $\mathcal{L}_m = \mathcal{L}_{m_1} \cup \mathcal{L}_{m_2} \cup \ldots \cup \mathcal{L}_{m_p} \cup$, $m_i \in \mathcal{B}'$, $1 \leq i \leq p$. Now, $\mathcal{L}_{m_i} = \sum_{j=1}^{p^i} \mathcal{L}_{m_j^i}$ and $m_j^i \in \mathcal{B}$. Hence $m \notin \mathcal{B}$, which is a contradiction. Δ

Corollary 2. *Given a string s, let $\mathcal{M}$ be the set of all motifs. Let $\mathcal{M}' \subseteq \mathcal{M}$ be an arbitrary set of maximal motifs. Then the basis $\mathcal{B}'$ of $\mathcal{M}'$ is such that $|\mathcal{B}'| < 3n$.*

This follows immediately since the proof of the theorem does not use the fact that $\mathcal{M}$ is the set of *all* motifs of s. It simply works on the location lists (sets) of this special set of maximal motifs. Δ

Algorithm to Detect the Irredundant Motifs. The algorithm is exactly along the lines of the one presented in [Par99,PRF+00] for the rigid motifs with the following exceptions. At the very first step all the rigid motifs with exactly two solid characters are constructed. The rigid motifs are consolidated to form flexible motifs. For example, motifs $a.b$, $a..b$, $a...b$ are consolidated to form $a.^{[1,3]}b$. Also $\mathcal{L}_{a.^{[1,3]}b}$ is set to $\mathcal{L}_{a.b} \cup \mathcal{L}_{a..b} \cup \mathcal{L}_{a...b}$. Also, each location is annotated for multiple occurrences, ie. all the *distinct* end locations of the realization of the motifs are stored at that location. For example if $x \in \mathcal{L}_{a.b}, \mathcal{L}_{a..b}$, then x is annotated to note that the motif occurs twice at that location ending in two different positions. When a pair of motifs are considered for concatenation, these multiple occurrences are taken into account.

Lemma 5. *The algorithm takes $O(n^5 \log n)$ time.*

Proof. At each iteration there are only $O(n)$ motifs after the pruning. Due to the multiple occurrences, there are $O(n^4)$ comparisons made at each iteration. Since the location list can be no more that n each with at most n occurrences, the amount of work done is $O(n^5 I)$, where I is the number of iterations. But $I = logL$ where L is the maximum number of solid characters in a motif, since the motif grows by concatenating two smaller motifs. But $L < n$, hence the result. Δ

Generalizations to Sets of Characters and Real Numbers. All of the discussion above can be easily generalized to dealing with input on sets of character and even real numbers. In the latter case, two real numbers r_1 and r_2 are deemed equal if for a specified δ, $|r_1 - r_2| < \delta$. Due to space constraints, we do not elaborate on this further, the details of handling rigid patterns appear in [Par99,PRF+00].

4 Conclusions

We have presented a uniform framework to study the problem of pattern discovery. We have separated the application from the task of discovering patterns in order to better understand the space of all patterns on a given string. The framework treats uniformly rigid and flexible motifs including strings of sets of characters and even real numbers. Through appropriate definitions of maximality and irredundancy we show that there exist a small set of crucial motifs called the *basis*. Since a polynomial time algorithm has been presented to detect the basis, this has algorithmic implications as well. It is possible that this basis would be critical in designing efficient algorithms to detect *all* or *significant* motifs depending on the application.

References

Alt89. S. Altschul. Gap costs for multiple sequence alignment. *J. Theor. Biol.*, 138:297–309, 1989.

AMS+95. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Advances in knowledge discovery and data mining, chapter 12,. In *Fast Discovery of Association Rules*. AAAI/MIT Press, MA, 1995.

AS95. R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on DataEngineering (ICDE95)*, pages 3–14. 1995.

Bay92. R. J. Bayardo. Efficiently mining long patterns from databases. In *Proc. of the 1998 ACM-SIGMOD Conference on Management of Data*, pages 85–93, 1992.

BG98. T. L. Bailey and M. Gribskov. Methods and statistics for combining motif match scores. *Journal of Computational Biology*, 5:211–221, 1998.

BJEG98. Alvis Brazma, Inge Jonassen, Ingvar Eidhammer, and David Gilbert. Approaches to the automatic discovery of patterns in biosequences. *Journal of Computational Biology*, 5(2):279–305, 1998.

Cal00. Andrea Califano. SPLASH: structural pattern localization algorithm by sequential histogramming. *Bioinformatics (under publication)*, 2000.

CL88. H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM Journal of Applied Mathematics*, pages 1073–1082, 1988.

DGM97. G. Das, D. Gunopulous, and H. Mannila. Finding similar time series. In *Priniciples of Knowledge Discovery and Data Mining*, 1997.

DT93. V Dhar and A Tuzhilin. Abstract-driven pattern discovery in databases. In *IEEE Transactions on Knowledge and Data Engineering*, 1993.

FRP+99. Aris Floratos, Isidore Rigoutsos, Laxmi Parida, Gustavo Stolovitzky, and Yuan Gao. Sequence homology detection through large-scale pattern discovery. In *Proceedings of the Annual Conference on Computational Molecular Biology (RECOMB99)*, pages 209–215. ACM Press, 1999.

GYW+97. Y. Gao, M. Yang, X. Wang, K. Mathee, and G. Narasimhan. Detection of HTH motifs via data mining. *International Conference on Bioinformatics*, 1997.

HS88. D. Higgins and P. Sharpe. CLUSTAL: A package for performing multiple sequence alignment on a microcomputer. *Gene*, 73:237–244, 1988.

HTHI95. M. Hirosawa, Y. Totoki, M. Hoshida, and M. Ishikawa. Comprehensive study on iterative algorithms of multiple sequence alignment. *CABIOS*, 11(1):13–18, 1995.

JCH95. I. J. F. Jonassen, J. F. Collins, and D. G. Higgins. Finding flexible patterns in unaligned protein sequences. *Protein Science*, pages 1587–1595, 1995.

LSB+93. C.E. Lawrence, S.F.Altschul, M.S. Boguski, J.S. Liu, A.F.Neuwald, and J.C. Wooton. Detecting subtle sequence signals: A Gibbs sampling strategy for multiple sequence alignment. *Science*, 262:208–214, 1993.

MSZM97. G. Myers, S. Selznick, Z. Zhang, and W. Miller. Progressive multiple alignment with constraints. In *Proceedings of the First Annual Conference on Computational Molecular Biology (RECOMB97)*, pages 220–225. ACM Press, 1997.

MVF96. M. A. McClure, T. K. Vasi, and W. M. Fitch. Comparative analysis of multiple protein-sequence alignment methods. *Mol. Bio. Evol.*, 11(4):571–592, 1996.

NG94. A. F. Neuwald and P. Green. Detecting patterns in protein sequences. *Journal of Molecular Biology*, pages 698–712, 1994.

Par98. L. Parida. *Algorithmic Techniques in Computational Genomics*. PhD thesis, Courant Institute of Mathematical Sciences, New York University, September 1998.

Par99. L. Parida. Bound of $3n$ on irredundant motifs on sequences of characters, character sets & real numbers and an $O(n^3 \ln(n))$ pattern detection algorithm. *IBM Technical Report*, 1999.

PFR99a. L. Parida, A. Floratos, and I. Rigoutsos. An approximation algorithm for alignment of mulitple sequences using motif discovery. *To appear in Journal of Combinatorial Optimization*, 1999.

PFR99b. L. Parida, A. Floratos, and I. Rigoutsos. MUSCA: An algorithm for constrained alignment of multiple data sequences. *Genome Informatics*, No 9:112–119, 1999.

PRF+00. Laxmi Parida, Isidore Rigoutsos, Aris Floratos, Dan Platt, and Yuan Gao. Pattern discovery on character sets and real-valued data: Linear bound on irredundant motifs and an efficient polynomial time algorithm. In *Proceedings of the eleventh ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 297 – 308. ACM Press, 2000.

RF98. I. Rigoutsos and A. Floratos. Motif discovery in biological sequences without alignment or enumeration. In *Proceedings of the Annual Conference on Computational Molecular Biology (RECOMB98)*, pages 221–227. ACM Press, 1998.

RFO+99. I. Rigoutsos, A. Floratos, C. Ouzounis, Y. Gao, and L. Parida. Dictionary building via unsupervised hierarchical motif discovery in the sequence space of natural proteins. *Proteins: Structure, Function and Genetics*, 37(2), 1999.

RFP+00. Isidore Rigoutsos, Aris Floratos, Laxmi Parida, Yuan Gao, and Daniel Platt. The emergence of pattern discovery techniques in computational biology. In *Journal of metabolic engineering*, 2000. To appear.

RGFp99. I. Rigoutsos, Y. Gao, A. Floratos, and L. parida. Building dictionaries of 2d and 3d motifs by mining the unaligned 1d sequences of 17 archeal and bacterial genomes. In *Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology*. AAAI Press, 1999.

Roy92. M. A. Roytberg. A search for common patterns in many sequences. *CABIOS*, pages 57–64, 1992.

SNJ95. M. Suyama, T. Nishioka, and O. Junícht. Searching for common sequence patterns among distantly related proteins. *Protein Engineering*, pages 366–385, 1995.

SV96. M. F. Sagot and A. Viari. A double combinatorial approach to discovering patterns in biological sequences. *Proceedings of the 7th symposium on combinatorial pattern matching*, pages 186–208, 1996.

Wat94. M.S. Waterman. Parametric and ensemble alignment algorithms. *Bulletin of Mathematical Biology*, 56(4):743–767, 1994.

WCM+94. J. T. L. Wang, G. W. Chrin, T. G. Marr, B. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1994.

WCM+96. J. Wang, G. Chirn, T. G. Marr, B. A. Shapiro, D. Shasha, and K. Jhang. Combinatorial pattern discovery for scientific data: Some preleminary results. *Proceedings of the ACM SIGMOD conference on management of data*, pages 115–124, 1996.

ZZ96. W. Miller Z. Zhang, B. He. Local multiple alignment vis subgraph enumeration. *Discrete Applied Mathematics*, 71:337–365, 1996.

Explaining and Controlling Ambiguity in Dynamic Programming

Robert Giegerich

Faculty of Technology, Bielefeld University
33501 Bielefeld, Germany
robert@techfak.uni-bielefeld.de

Abstract. Ambiguity in dynamic programming arises from two independent sources, the non-uniqueness of optimal solutions and the particular recursion scheme by which the search space is evaluated. Ambiguity, unless explicitly considered, leads to unnecessarily complicated, inflexible, and sometimes even incorrect dynamic programming algorithms. Building upon the recently developed algebraic approach to dynamic programming, we formalize the notions of ambiguity and canonicity. We argue that the use of canonical yield grammars leads to transparent and versatile dynamic programming algorithms. They provide a master copy of recurrences, that can solve all DP problems in a well-defined domain. We demonstrate the advantages of such a systematic approach using problems from the areas of RNA folding and pairwise sequence comparison.

1 Motivation and Overview

1.1 Ambiguity Issues in Dynamic Programming

Dynamic Programming (DP) solves combinatorial optimization problems. It is a classical programming technique throughout computer science [3], and plays a dominant role in computational biology [4, 10]. A typical DP problem spawns a search space of potential solutions in a recursive fashion, from which the final answer is selected according to some criterion of optimality. If an optimal solution can be derived recursively from optimal solutions of subproblems [1], DP can evaluate a search space of exponential size in polynomial time and space.

Sources of Ambiguity. By ambiguity in dynamic programming we refer to the following facts which complicate the understanding and use of DP algorithms:

- *Co-optimal and near-optimal solutions:* It is well known that the "optimal" solution found by a DP algorithm normally is not unique, and there may be relevant near-optimal solutions. A single, "optimal" answer is often unsatisfactory. Considerable work has been devoted to this problem, producing algorithms providing near-optimal [15, 17] and parametric [11] solutions.
- *Duplicate solutions:* While there is a general technique to enumerate all solutions to a DP problem (possibly up to some threshold value) [21, 22],

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 46–59, 2000.

such enumeration is hampered by the fact that the algorithm may produce the same solution several times – and in fact, this may lead to combinatorial explosion of redundancy. Heuristic enumeration techniques, and post-facto filtering as a safeguard against duplicate answers are employed e.g. in [23].
- *(Non-)canonical solutions:* Often, the search space exhibits additional redundancy in terms of solutions that are represented differently, but are equivalent from a more semantic point of view. Canonization is important in evaluating statistical significance [14], and also in reducing redundancy among near-optimal solutions.

Ambiguity Examples. Strings `aaaccttaa` and `aaaggttaa` are aligned below. Alignments (1) and (2) are equivalent under most scoring schemes, while (3) may even be considered a mal-formed alignment, as it shows two deletions separated by an insertion.

```
aaacc--ttaa        aaa--ccttaa        aaac--cttaa
aaa--ggttaa        aaagg--ttaa        aaa-gg-ttaa
    (1)                (2)                (3)
```

In the RNA folding domain, each DP algorithm seems to be a one-trick pony. Different recurrences have been developed for counting or estimating the number of various classes of feasible structures of a sequence of given length [12], for structure enumeration [22], energy minimization [25], and base pair maximization [19]. Again, enumerating co-optimal answers will produce duplicates in the latter two cases. In [4](p. 272) a probabilistic scoring scheme is suggested to find the *most likely* RNA secondary structure – this is a valid idea, but will work correctly only if the underlying recursion scheme considers each feasible structure exactly once.

Our Main Contributions. The recently developed technique of *algebraic* dynamic programming (ADP), summarized in Section 2, uses yield grammars and evaluation algebras to specify DP algorithms on a rather high level of abstraction. DP algorithms formulated this way *can* have all the ambiguity problems illustrated above. However, the ADP framework also helps to analyse and avoid these problems.

1. In this article, we devise a formal framework to explain and reason about ambiguity in its various forms.
2. Introducing *canonical* yield grammars, we show how to construct a "master copy" of a DP algorithm for a given problem class. This single set of recurrences can correctly and efficiently perform all analyses in this problem class, including optimization, complete enumeration, sampling and statistics.
3. Re-use of the master recurrences for manifold analyses provides a major advantage from a software-engineering point of view, as it enhances not only programming economy, but also program reliability.

2 A Short Review of Algebraic Dynamic Programming

ADP introduces a *conceptual* splitting of a DP algorithm into a recognition and an evaluation phase. A *yield grammar* is used to specify the recognition phase (i. e. the search space of the optimization problem). A particular parsing technique turns the grammar directly into an efficient dynamic programming scheme. The evaluation phase is specified by an *evaluation algebra*, and each grammar can be combined with a variety of algebras to solve different problems over the same data domain, for which heretofore DP recurrences had to be developed independently.

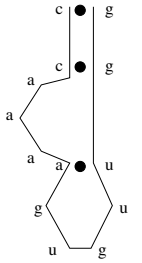
2.1 Basic Notions

Let $\mathcal{A}$ be an alphabet. $\mathcal{A}^*$ denotes the set of finite strings over $\mathcal{A}$, and `++` denotes string concatenation. Throughout this article, $x, y \in \mathcal{A}^*$ denote input strings to various problems, and $|x| = n$. A subword is indicated by its boundaries – $x_{(i,j)}$ denotes $x_{i+1}...x_j$.

An algebra is a set of values and a family of functions over this set. We shall allow that these functions take additional arguments from $\mathcal{A}^*$. An algebraic data type $\mathcal{T}$ is a type name and a family of typed function symbols, also called operators. It introduces a language of (well-typed) formulas, called the term algebra. An algebra that provides a function for each operator in $\mathcal{T}$ is a $\mathcal{T}$-algebra. The interpretation $t_{\mathcal{I}}$ of a term t in a $\mathcal{T}$-algebra $\mathcal{I}$ is obtained by substituting the corresponding function of the algebra for each operator. Thus, $t_{\mathcal{I}}$ evaluates to a value in the base set of $\mathcal{I}$.

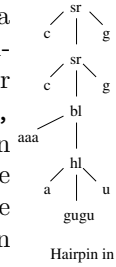
Terms as syntactic objects can be equivalently seen as trees, where each operator is a node, which has its subterms as subtrees. Tree grammars over $\mathcal{T}$ describe specific subsets of the term algebra. A regular tree grammar over $\mathcal{T}$ [2, 7] has a set of nonterminal symbols, a designated axiom symbol, and productions of the form $X \rightarrow t$ where X is a nonterminal symbol, and t is a tree pattern, i.e. a tree over $\mathcal{T}$ which may have nonterminals in leaf positions.

2.2 ADP – The Declarative Level



Simple hairpin structure

In the sequel, we assume that $\mathcal{T}$ is some fixed data type, and $\mathcal{A}$ a fixed alphabet. As a running example, let $\mathcal{A} = \{$`a, c, g, u`$\}$, representing the four bases in RNA, and let $\mathcal{T}$ consist of the operators `sr, hl, bl, br, il`, representing structural elements in RNA: stacking regions, hairpin loops, bulges on the left and right side, and internal loops. Feasible base pairs are `a - u, g - c, g - u`. The little hairpin denoted by the term



Hairpin in term representation

```
s = sr 'c' (sr 'c' (bl "aaa" (hl 'a' "gugu" 'u')) 'g') 'g'
```

is one of many possible 2D structures of the RNA sequence `ccaaaaguguugg`.

Definition 1 (Evaluation Algebra). *An evaluation algebra is a $\mathcal{T}$-algebra augmented by a choice function h. If l is a list of values of the algebra's value set, then $h(l)$ is a sublist thereof. We require h to be polynomial in $|l|$. Furthermore, h is called reductive if $|h(l)|$ is bounded by a constant.*

The choice function is a standard part of our evaluation algebras because we shall deal with optimization problems. Typically, h will be minimization or maximization, but, as we shall see, it may also be used for counting, estimation, or some other kind of synopsis. Non-reductive choice functions are used e.g. for complete enumeration. The hairpin `s` evaluates to 3 in the *basepair algebra*, and (naturally) to 1 in the *counting algebra*:

```
basepair_alg = (sr,hl,bl,br,il,h)
 where sr _ x _ = x+1
       hl _ x _ = 1
       bl _ x   = x
       br   x _ = x
       il _ x _ = x
       h        = maximum
```

```
counting_alg = (sr,hl,bl,br,il,h)
 where sr _ x _ = x
       hl _ x _ = 1
       bl _ x   = x
       br   x _ = x
       il _ x _ = x
       h        = sum
```

Definition 2 (Yield Grammar). *A yield grammar $(\mathcal{G}, y)$ is given by*

- *an underlying algebraic datatype $\mathcal{T}$, and alphabet $\mathcal{A}$,*
- *a homomorphism $y : \mathcal{T} \to \mathcal{A}^*$ called the yield function,*
- *a regular tree grammar $\mathcal{G}$ over $\mathcal{T}$.*

$\mathcal{L}(\mathcal{G})$ denotes the tree language derived from the axiom, and $\mathcal{Y}(\mathcal{G}) := \{y(t) |\ t \in \mathcal{L}(\mathcal{G})\}$ is the yield language of $\mathcal{G}$.

The homomorphism condition means that $y(Cx_1...x_n) = y(x_1)$`++`$...$`++`$y(x_n)$ for any operator C of $\mathcal{T}$. For the hairpin `s`, we have $y($`s`$) =$ `ccaaaaguguugg`. By virtue of the homomorphism property, we may apply the yield function to the righthand sides of the productions in the tree grammar. In this way, we obtain a context free grammar $y(\mathcal{G})$ such that $\mathcal{Y}(\mathcal{G}) = \mathcal{L}(y(\mathcal{G}))$.

Definition 3 (Yield Parsing). *The yield parsing problem of $(\mathcal{G}, y)$ is to compute for a given $s \in \mathcal{A}^*$ the set of all $t \in \mathcal{T}$ such that*

$$y(t) = s.$$

Definition 4 (Algebraic Dynamic Programming). *Let $\mathcal{I}$ be a $\mathcal{T}$-algebra with a reductive choice function $h_\mathcal{I}$. Algebraic Dynamic Programming is computing for given $s \in \mathcal{A}^*$ the set of solutions*

$$h_\mathcal{I}\{t_\mathcal{I} \mid y(t) = s\} \text{ in polynomial time and space.}$$

This definition precisely describes a class of DP problems over sequences. All biosequence analysis problems we have studied so far fall under the ADP framework. Outside the realm of sequences, DP is also done over trees, dags, and graphs. It is open whether the concept of a yield grammar can be generalized to accommodate these domains. A detailed discussion of the scope of ADP is beyond the space limits of this short review.

2.3 ADP – The Notation

Once a problem has been specified by a yield grammar and an evaluation algebra, the ADP approach provides a systematic transition to an efficient DP algorithm that solves the problem. To achieve this, we introduce a notation for yield grammars that is both human readable and — executable! In ADP notation, yield grammars are written in the form

```
hairpin  = axiom struct where
   struct = open ||| closed
   open   = bl <<< region ~~~ closed |||
            br <<< closed ~~~ region |||
            il <<< region ~~~ closed ~~~ region
```

The grammar `hairpin` has axiom `struct` and further nonterminal symbols `open, closed`. `base` denotes an arbitrary base, and `region` a nonempty sequence of bases from the RNA alphabet. The grammar notation is refined further by allowing predicates and the choice function to be associated with nonterminals symbols and productions:

```
closed =
  ((hl <<< base ~~~ (region `with` minsize 3) ~~~ base          |||
    sr <<< base ~~~ (closed ||| open) ~~~ base) `with` basepair) ... h
```

This production uses two predicates: `minsize k` requires a yield of minimal length `k`, and `basepair` applies to both alternatives, requiring that the bounding bases of either closed structure form a feasible base pair. The choice function `h` is attached via the `...`-combinator, indicating that from several alternative closed structures, a selection according to `h` is imposed.

In the syntactic view of yield grammars, we interpret the operators `hl, sr, ...` in the term algebra. They merely construct terms or trees representing hairpins. In this view, the choice function `h` has little use and should be assumed to be the identity function. However, in a more semantic view, we see `hl, sr, ...` as functions of some evaluation algebra. Then, the "trees" generated by the grammar are actually formulas that can be evaluated. In this view, the grammar is a mechanism to generate a set of values, and it makes sense to apply the algebra's choice function to select (say) a maximal one.

2.4 ADP – The Implementation Level

We now solve the yield parsing problem. A nondeterministic, top-down parser for a context-free grammar is easily obtained by the combinator technique of [13]. This idea is adapted to yield grammars. A yield parser pN for nonterminal N takes a subword (i, j) of x as its argument and returns the set $pN(i, j) = \{t | y(t) = x_{(i,j)}\}$. Technically, it returns a list; when the list is empty, we say that the parser fails. Where the operators of $\mathcal{T}$ take strings from $\mathcal{A}^*$ as their arguments, suitable parsers must be provided.

The grammar itself is turned into a parser by defining the combinators as higher-order functions which compose complex parsers from simpler ones. For

the sake of completeness, definitions are given here, but space does not allow a thorough discussion. We use list comprehension notation borrowed from the functional programming language *Haskell*.

```
(r ||| q) (i,j)    = r(i,j) ++ q(i,j)
(f <<< q) (i,j)    = [f z | z <- q(i,j)]
(r ~~~ q) (i,j)    = [f y | k <- [i+1..j-1], f <- r(i,k), y <- q(k,j)]
(r ... h) (i,j)    = h(p(i,j))
axiom q            = q(0,n)
(r `with` w) (i,j) = if w(i,j) then r(i,j) else []
```

Note that the `axiom`- and the `with`-clause are also defined as functions applied to parsers. With these definitions, a grammar like `hairpin` is now an executable yield parser, albeit of miserable efficiency: There may be an exponential number of parses, and any subparse is constructed many times. This is alleviated by tabulating the parser functions. Let `p` be a table indexing function and `tabulated` be a tabulation function such that

```
p (tabulated f) (i,j) = f(i,j), or equivalently
p (tabulated f)       = f
```

With this convention, a grammar may be annotated for efficiency, replacing parsers by tables. Choosing to tabulate the parser for nonterminal `closed`, grammar `hairpin` now reads

```
hairpin = axiom struct where
 struct =        open                  ||| p closed
 open   = bl <<< region   ~~~ p closed |||
          br <<< p closed ~~~ region   |||
          il <<< region   ~~~ p closed ~~~ region
 closed = tabulated (
  ((hl <<< base ~~~ (region `with` minsize 3) ~~~ base |||
    sr <<< base ~~~ (p closed ||| open) ~~~ base) `with` basepair) ... h)
```

Such annotation does not affect the meaning of the grammar, nor that of the parser. It only affects the parser's efficiency: The parser now uses dynamic programming. In general, the parser consists of a family of recursively defined tables and functions. Substituting the definitions of the combinators and the functions of a specific evaluation algebra, the annotated grammar simplifies to a set of recurrences as we traditionally see it in dynamic programming.

2.5 Two Classical DP Algorithms in ADP Notation

Zuker's Algorithm for RNA Folding. Zuker and Stiegler [25] gave a DP algorithm for determining the minimal free energy structure of an RNA molecule under the nearest neighbour model. The model and the algorithm have been elaborated considerably since then, but for lack of space, we base our discussion on the original description. Evers [5] has recently reformulated Zuker's recurrences as a yield grammar $\mathcal{G}_{\texttt{zuker81}}$[1]:

[1] This example shows actually executable ADP code, and contains a few refinements not explained in Section 2. The variants of the `~~~`-operator are all equivalent in

```
zuker81 algebra inp                  = axiom struct where
  (str,hl,bi,sr,bl,br,il,ol,ox,co,h) = algebra

              -- nonterminals v and w are Zuker's tables V and W.
  struct      = str <<< p w
  v           = tabulated (
      ((hairpin ||| twoedged ||| bifurcation) `with` basepair)    ... h)
  hairpin     = hl <<< base -~~ (region `with` minsize 3) ~~- base
  bifurcation = bi <<< base -~~ p w ~-~ p w ~~- base              ... h
  twoedged    = stack ||| bulgeleft ||| bulgeright ||| interior   ... h
  stack       = sr <<< base -~~             p v            ~~- base
  bulgeleft   = bl <<< base -~~ region ~-~ p v            ~~- base
  bulgeright  = br <<< base -~~             p v ~-~ region ~~- base
  interior    = il <<< base -~~ region ~-~ p v ~-~ region ~~- base

  w = tabulated ( openleft ||| openright ||| p v ||| connected    ... h)
  openleft  = ol <<<   base -~~ p w
  openright = ox <<< p w    ~~-   base
  connected = co <<< p w    ~-~ p w ... h
```

This grammar uses two essential nonterminals, `v` and `w`; the others are introduced to reflect Zuker's case analysis. It is quite instructive to reformulate classical DP algorithms in the uniform ADP framework. Making explicit the grammar behind the algorithm helps to clarify properties relating to ambiguity as well as efficiency.

The Needleman–Wunsch Algorithm of 1970. The Needleman-Wunsch algorithm for pairwise sequence comparison [18] is based on a particularly simple yield grammar with a single nonterminal symbol `alignment`, terminals `xbase, ybase, region, empty`, and the algebra represented the five operators `replace, delete, insert, nil, h`. When sequences x and y are to be aligned, the input to this parser is $x{+}{+}y^{-1}$.

```
nw_alignment algebra x y            =  axiom alignment  where
  (replace, delete, insert, nil, h) =  algebra
  alignment =  tabulated (
               replace <<<   xbase      ~~~ p alignment ~~~ ybase |||
               delete  <<<   region     ~~~ p alignment           |||
               insert  <<< p alignment  ~~~   region              |||
               nil     ><<   empty                                ... h)
```

3 Ambiguity and Canonicity

3.1 Formalizing Ambiguity and Canonicity

Remember that a context-free grammar $\mathcal{G}$ is ambiguous, if there are different leftmost derivations for some $x \in \mathcal{L}(\mathcal{G})$.

the declarative view, but operationally they are special cases with a more efficient implementation. E.g., `~~-` is used when the righthand parser accepts a single base.

Definition 5 (Yield Grammar Ambiguity). *A tree grammar $\mathcal{G}$ is ambiguous if there are different leftmost derivations for some tree $t \in \mathcal{L}(\mathcal{G})$. A yield grammar $(\mathcal{G}, y)$ is ambiguous, if $\mathcal{G}$ is ambiguous, otherwise it is unambiguous. A yield grammar $(\mathcal{G}, y)$ is strictly unambiguous, if it is unambiguous and y is injective.*

Strict unambiguity means that for each $s \in \mathcal{A}^*$, we have at most one $t \in \mathcal{L}(\mathcal{G})$ such that $y(t) = s$. Hence, we do not have an optimization problem at all. Strictly unambiguous yield grammars play no part in dynamic programming.

Canonicity means that all solutions from which we want to choose an optimal one have a *unique* representation in the search space. For example, alignments as shown in Sect. 1.1 could be canonized by requiring that deletions are arranged always before adjacent insertions. To formalize canonicity, we must introduce a canonical model as the point of reference.

Definition 6 (Canonical Models and Canonical Yield Grammars). *Let $\mathcal{K}$ be a set, the canonical model. Let k be a mapping from $\mathcal{L}(\mathcal{G})$ to $\mathcal{K}$. A yield grammar $(\mathcal{G}, y)$ is canonical w.r.t. $\mathcal{K}$ and k if it is unambiguous and the mapping k is bijective. A DP algorithm is canonical w.r.t. $\mathcal{K}$ and k, if the underlying yield grammar is canonical w.r.t. $\mathcal{K}$ and k.*

The canonical model may exist merely in the mind of the algorithm designer, but preferably, it should be formulated explicitly, together with the mapping k.

3.2 Analysing Canonicity

We show that the Zuker algorithm is not canonical. A canonical model for RNA secondary structures would be sets of properly nested base pairs. Such a model is too remote from the tree-like representation of RNA structures. The Vienna notation, encoding a structure as a string of dots and properly nested parentheses, however, proves to be very convenient. It can be formally defined as $\mathcal{L}(\mathcal{V})$, using the string grammar $\mathcal{V} = \{R \to .|..|S, \quad S \to ...|.S|S.|SS|(S)\}$. Our little hairpin `s` would be denoted by the pair `("((...(....)))","ccaaaaguguugg")`. The mapping k from Zuker's underlying data type $\mathcal{Z}$ to $\mathcal{L}(\mathcal{V})$ is defined via

$$
\begin{aligned}
k(bi(a,u,v,b) &= \texttt{"("} \texttt{ ++ } k(u) \texttt{ ++ } k(v) \texttt{ ++ } \texttt{")"} \\
k(ol(a,v) &= \texttt{"."} \texttt{ ++ } k(v) \\
k(co(u,v) &= k(u) \texttt{ ++ } k(v) \\
k(ox(u,b) &= k(u) \texttt{ ++ } \texttt{"."}
\end{aligned}
$$

Further equations are omitted, as these suffice to prove the equalities below.

Theorem 1. *The Zuker DP algorithm for RNA folding is not canonical with respect to feasible RNA structures.*

Proof. We observe the equalities

$$k(ol(a, ox(w, b))) = k(ox(ol(a, w), b) \quad (1)$$
$$k(co(u, co(v, w))) = k(co(co(u, v), w)) \quad (2)$$
$$k(ol(u, co(v, w))) = k(co(ol(u, v), w)) \quad (3)$$
$$k(bi(a, u, co(v, w), b)) = k(bi(a, co(u, v), w, b)) \quad (4)$$
$$k(bi(a, ox(u, b), w, c)) = k(bi(a, u, ol(b, w), c)) \quad (5)$$

Either one of these proves that k is not injective.

While equalities (1) and (2) are quite obvious and easy to avoid, (3) – (5) are more subtle, and there may be more such equalities.

The degree of redundancy incurred by the non-canonical grammar is demonstrated in Section 4.4. Such redundancy is *not* an efficiency problem, as the asymptotic efficiency of a DP algorithm is not affected. However, it makes it impossible to use the same recurrences for other purposes, say for the enumeration of all suboptimal solutions. This explains why Zuker's algorithm employs an incomplete heuristics when enumerating suboptimal foldings.

4 Master Recurrences for RNA Folding

4.1 A Canonical Grammar for Feasible RNA Structures

In [8] a data type $\mathcal{FS}$ is given together with a grammar $\mathcal{G}_f$ of all feasible structures. $\mathcal{FS}$ extends $\mathcal{T}$ as used above by operators `ss` and `ml` representing single stranded and multiloop structures, plus `cons` and `ul` for constructing component lists. It is easy to show by induction that $\mathcal{L}(\mathcal{G}_f) \subset \mathcal{FS}$, and there is a canonical mapping $k : \mathcal{L}(\mathcal{G}_f) \to \mathcal{L}(\mathcal{V})$. Another grammar for feasible structures is implicitly given by the recurrences developed in [22]. These recurrences are designed for canonicity, since the authors seek a complete and non-redundant enumeration of suboptimal structures. We do not show either grammar here as we plan to go one step further, which will provide a significant reduction in the number of structures to be considered.

4.2 A Canonical Grammar for Canonical RNA Secondary Structures

Although the energy model permits structures of minimal free energy with isolated (unstacked) base pairs, there are good biophysical arguments to consider such structures unrealistic. As already noted by Zuker and Sankoff in [24][2], removing such redundant structures from the search space is the key to obtaining more significant near-optimal solutions.

[2] Zuker and Sankoff suggest an even stronger restriction to structures with maximal helices. Such recurrences are within the scope of the ADP approach [5], but well beyond the space limits of this article.

Definition 7. *An RNA structure without isolated base pairs is canonical.*

The canonical model suiting this definition is defined as $\mathcal{L}(\mathcal{W}) \times \mathcal{A}^*$ using the string grammar $\mathcal{W} = \{R \to \epsilon|.|..|S, \quad S \to ...|.S|S.|SS|((P)), \quad P \to S|(P)\}$. $(d, s) \in \mathcal{K}$ is subject to the restriction that bases in s can pair as indicated by matching parentheses in$d$. The following grammar $\mathcal{G}_c$ for canonical RNA structures is based on the data type $\mathcal{FS}$. It uses an algebra with several base sets, and an overloaded choice function `h`.

```
canonicals alg x = axiom struct where
 (str,ss,hl,sr,bl,br,il,ml,nil,cons,ul,h,) = alg
 singlestrand = ss <<< region
 struct = str <<< p comps                              |||
          str <<< (ul  <<< singlestrand)               |||
          str <<< (nil >>>  empty)                                  ... h
 comps  = tabulated (cons  <<< p block ~~~ p comps  |||
                     ul    <<< p block              |||
                     cons  <<< p block ~~~ (ul <<< singlestrand) ... h)
 block  = tabulated (p strong ||| bl <<< region ~~~ p strong     ... h)

 strong = tabulated
           (((sr <<< base -~~ ( p strong ||| p weak)    ~~- base)
                                                 `with` basepair) ... h)
 weak   = tabulated
           (((hairpin ||| leftB ||| rightB ||| iloop ||| multiloop)
                                                 `with` basepair) ... h)
  where
  hairpin   = hl <<< base -~~ (region `with` minsize 3)          ~~- base
  leftB     = sr <<< base -~~ (bl   <<< region    ~~~ p strong) ~~- base
  rightB    = sr <<< base -~~ (br   <<< p strong  ~~~ region)   ~~- base
  multiloop = ml <<< base -~~ (cons <<< p  block  ~~~ p comps)  ~~- base
  iloop     = sr <<< base -~~ (il   <<< region    ~~~ p strong
                                                  ~~~ region)   ~~- base
```

The grammar distinguishes substructures closed by a single base pair (`weak`) from those closed by at least two stacked pairs (`strong`). If we identify these two nonterminals and merge their productions, an ADP version of the Wuchty et al. DP recurrences [22] is obtained. Note how the grammar takes care that single strands and closed components alternate in multiloops, and that multiloops contain at least two branches.

We now specify the canonical mapping $k : \mathcal{L}(\mathcal{G}_c) \to \mathcal{L}(\mathcal{W})$

```
k (str cs)      = k'' cs
k (hl b1 r b2)  = "(" ++ k' r    ++ ")"
k (sr b1 s b2)  = "(" ++ k' s    ++ ")"
k (ml b1 cs b2) = "(" ++ k'' cs  ++ ")"
k (bl r s)      = k' r ++ k  s
k (br s r)      = k s  ++ k' r
k (il r1 s r2)  = k' r1 ++ k s ++ k r2
k (ss r)        = k' r
k' r            = ['.' | b <- r]    -- a sequence of |r| dots
k'' cs          = concat (map k cs) -- concatenating (k c) for all c in cs
```

Theorem 2. *Grammar $\mathcal{G}_c$ is a canonical yield grammar for canonical RNA secondary structures.*

Proof. We have to show that (a) the grammar $\mathcal{G}_c$ is unambiguous, and (b) the mapping k is bijective. (a) is shown by induction on the derivations of the grammar. For (b), injectivity of k is shown by structural recursion, while surjectivity uses a string grammar $k(\mathcal{G}_c)$ (in analogy to $y(\mathcal{G})$ in Sect. 3) to show that $\mathcal{L}(\mathcal{W}) \subset \mathcal{L}(k(\mathcal{G}_c))$. Details are omitted.

4.3 Efficiency

A canonical grammar, whether encoded in ADP or in conventional matrix recurrences, may require some extra tables compared to its non-canonical counterpart, in order to keep more structures distinct. In our RNA example, the non-canonical grammar `zuker81` uses 2 tables, while the canonical grammar `canonicals` uses 4. This is the price for the added versatility.

4.4 Applications

Due to Theorem 2 we know that the DP algorithm $\mathcal{G}_c$ considers each canonical RNA structure exactly once. Hence it can serve as a "master copy" of *all* DP algorithms which can be formulated as a $\mathcal{FS}$-algebra.

Simple Evaluation Algebras. The analyses in Table 1 can be defined each in a few lines: Energy minimization for canonical structures has been designed and

Purpose	Value Domain	Interpretation of Operators	Choice Function
Energy minimization	energy values	energy rules for hairpin loops, bulges, stacked pairs, etc.	minimization
Structure enumeration	trees in data type $\mathcal{T}$	tree constructors `HL, IL, SR,` etc.	identity function
Structure counting	Integers	multiply counts of substructures	summation
Structure count estimation	Reals	multiply counts by pairing prob.	summation

Table 1. Different analyses based on $\mathcal{G}_c$

implemented in [5] and [16]. Structure enumeration has been used to generate visualizations of the folding space via RNA-Movies [6]. Structure counts are correct due to canonicity of the grammar, and canonization of structures proves a dramatic reduction of the folding space. A probabilistic estimate for feasible structures obtained in this manner[3] was already shown in [8] to be remarkably

[3] Equivalently based on the canonical grammar for feasible structures or on the special recurrences given in [24], but modified to reflect base composition.

accurate. Accuracy is confirmed for the estimate of canonical structures supplied here. The Waterman formula [20] for the number of possible structures for all sequences of length n can also be written as a simple yield grammar.

Some Observations. A few of the observations made by applying these evaluation algebras are summarized in Table 2, showing structure statistics for initial segments of an RNA sequence[4] from *neurospora crassi*. n denotes sequence length, and the columns list structures counted, estimated, or evaluated by the various algorithms. These figures also indicate that the majority of structures accounted

n	Waterman formula	Zuker algorithm	Probabilistic estimate feasibles	Feasible structs.	Canonical structs.	Probabilistic estimate canonicals
5	8	0	1.16	1	1	1.00
10	423	12	5.98	9	1	1.34
15	30372	544	100.82	106	7	5.82
20	2516347	38160	510.60	390	7	9.02
25	226460893	2428352	15160.50	16343	72	71.37
30	21511212261	229202163	175550.00	235025	244	233.80

Table 2. Some structure statistics collected via the algebras listed in Table 1

for by Waterman's formula do not exist in the folding space of a *given* sequence, the majority of structures considered by the Zuker algorithm is redundant, and the majority of the feasible structures enumerated by the non-redundant Wuchty algorithm is non-canonical.

Combined Analyses. Since all algebras share the same grammar, a general construction is available [9] that forms the cross product of two algebras. Everything is mechanic except the combined choice function. This means we can, for example, return an optimal solution together with the total number of co-optimal solutions.

Structural Motifs. Retaining the evaluation algebras and the canonical model, but specializing the grammar, we obtain the above analyses for all classes of structural motifs that can be described by a regular tree grammar.

Application to Pairwise Sequence Comparison. In many situations, confidence in the answer computed by a sequence alignment algorithm could be substantiated by reporting the number of (different) co-optimal answers, accompanied by some measure of the diversity within the co-optimal answer space. This, again, requires canonization, which can be achieved by our approach. For lack of space, we refer the reader to the "alignment ambiguity awareness suite" in [9], Chapter 3.

[4] `"gaccauacccacuggaaaacucgggaucccguccgcucuccca..."`.

5 Conclusion

We have provided a framework for reasoning about properties of DP algorithms related to ambiguity. We hope to have shown that canonical yield grammars are a useful concept both in theory – understanding the properties of DP algorithms – and in practice – building reliable and versatile DP algorithms more quickly. We expect to apply this approach to further problem domains, as we are working to explore the full scope of algebraic dynamic programming.

References

1. R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.
2. W.S. Brainerd. Tree generating regular systems. *Information and Control*, 14:217–231, 1969.
3. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.
4. R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis.* Cambridge University Press, 1998.
5. D. Evers. *RNA Folding via Algebraic Dynamic Programming.* Bielefeld University, 2000. Forthcoming Dissertation.
6. D. Evers and R. Giegerich. RNA Movies: Visualizing RNA Secondary Structure Spaces. *Bioinformatics*, 15(1):32–37, 1999.
7. R. Giegerich. Code Selection by Inversion of Order-Sorted Derivors. *Theor. Comput. Sci.*, 73:177–211, 1990.
8. R. Giegerich. A declarative approach to the development of dynamic programming algorithms, applied to RNA folding. Report 98–02, Technische Fakultät, Universität Bielefeld, 1998.
9. R. Giegerich. Towards a discipline of dynamic programming in bioinformatics. Parts 1 and 2: Sequence comparison and RNA folding. Report 99–05, Technische Fakultät, Universität Bielefeld, 1999. (Lecture Notes).
10. D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.
11. D. Gusfield, K. Balasubramanian, and D. Naor. Parametric Optimization of Sequence Alignment. *Algorithmica*, 12:312–326, 1994.
12. I.L. Hofacker, P. Schuster, and P.F. Stadler. Combinatorics of rna secondary structures. *Discr. Appl. Math*, 89:177–207, 1999.
13. G. Hutton. Higher Order Functions for Parsing. *Journal of Functional Programming*, **3**(2):323–343, 1992.
14. S. Kurtz and G. W. Myers. Estimating the Probability of Approximate Matches. In *Proceedings Combinatorial Pattern Matching*, pages 52–64, 1997.
15. H.T. Mevissen and M. Vingron. Quantifying the Local Reliability of a Sequence Alignment. *Prot. Eng.*, 9(2), 1996.
16. C. Meyer. Lazy Auswertung von Rekurrenzen der Dynamischen Programmierung, 1999. Diploma Thesis, Bielefeld University, (in German).
17. D. Naor and D. Brutlag. On Near-Optimal Alignments in Biological Sequences. *J. Comp. Biol.*, 1:349–366, 1994.
18. S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48:443–453, 1970.

19. R. Nussinov, G. Pieczenik, J.R. Griggs, and D.J. Kleitman. Algorithms for loop matchings. *SIAM J. Appl. Math.*, 35:68–82, 1978.
20. M. S. Waterman and T. F. Smith. RNA secondary structure: A complete mathematical analysis. *Math. Biosci.*, 41:257–266, 1978.
21. M.S. Waterman and T.H. Byers. A dynamic programming algorithm to find all solutions in a neighborhood of the optimum. *Math. Biosci.*, 77:179–188, 1985.
22. S. Wuchty, I. Fontana, W.and Hofacker, and P. Schuster. Complete suboptimal folding of RNA and the stability of secondary structures. *Biopolymers*, 49:145–165, 1998.
23. M. Zuker. On Finding all Suboptimal Foldings of an RNA Molecule. *Science*, 244:48–52, 1989.
24. M. Zuker and S. Sankoff. RNA secondary structures and their prediction. *Bull. Math. Biol.*, 46:591–621, 1984.
25. M. Zuker and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Res.*, 9(1):133–148, 1981.

A Dynamic Edit Distance Table*

Sung-Ryul Kim and Kunsoo Park

School of Computer Science and Engineering
Seoul National University
Seoul 151-742, Korea
{kimsr,kpark}@theory.snu.ac.kr

Abstract. In this paper we consider the incremental/decremental version of the edit distance problem: given a solution to the edit distance between two strings A and B, find a solution to the edit distance between A and B' where $B' = aB$ (incremental) or $bB' = B$ (decremental). As a solution for the edit distance between A and B, we define the difference representation of the D-table, which leads to a simple and intuitive algorithm for the incremental/decremental edit distance problem.

1 Introduction

Given two strings $A[1..m]$ and $B[1..n]$ over an alphabet Σ, the *edit distance* between A and B is the minimum number of *edit operations* needed to convert A to B. The edit distance problem is to find the edit distance between A and B. Most common edit operations are the following.

1. *change*: replace one character of A by another single character of B.
2. *deletion*: delete one character from A.
3. *insertion*: insert one character into B.

A well-known method for solving the edit distance problem in $O(mn)$ time uses the *D-table* [1,10]. Let $D(i,j)$, $0 \le i \le m$ and $0 \le j \le n$, be the edit distance between $A[1..i]$ and $B[1..j]$. Initially, $D(i,0) = i$ for $0 \le i \le m$ and $D(0,j) = j$ for $0 \le j \le n$. An entry $D(i,j)$, $1 \le i \le m$ and $1 \le j \le n$, of the D-table is determined by the three entries $D(i-1,j-1)$, $D(i-1,j)$, and $D(i,j-1)$. The recurrence for the D-table is as follows: For all $1 \le i \le m$ and $1 \le j \le n$,

$$D(i,j) = \min\{D(i-1,j-1) + \delta_{ij}, D(i-1,j) + 1, D(i,j-1) + 1\} \quad (1)$$

where $\delta_{ij} = 0$ if $A[i] = B[j]$; $\delta_{ij} = 1$, otherwise.

In this paper we consider the following incremental (resp. decremental) version of the edit distance problem: given a solution for the edit distance between A and B, compute a solution for the edit distance between A and aB (resp. B' where $B = bB'$), where a (resp. b) is a symbol in Σ. By a *solution* we mean some

* This work was supported by the Brain Korea 21 Project.

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 60–68, 2002.

encoding of the D-table computed between A and B. Since essentially the same techniques can be used to solve both incremental and decremental versions of the edit distance problem, we will consider only the decremental version.

The incremental/decremental version of the edit distance problem was first considered by Landau et al. [3]. They used the C-table [2,4,5,7,9] (represented with linked lists) as a solution for the edit distance between A and B. Given a threshold k on the edit distance, their algorithm runs in $O(k)$ time. (If the threshold k is not given, it runs in $O(m+n)$ time.) However, the result in [3] is quite complicated.

As a solution for the edit distance between A and B, we define the difference representation of the D-table (DR-table for short). Each entry $DR(i,j)$ in the DR-table between A and B has two fields defined as follows: For $1 \leq i \leq m$ and $1 \leq j \leq n$,

1. $DR(i,j).U = D(i,j) - D(i-1,j)$
2. $DR(i,j).L = D(i,j) - D(i,j-1)$

A third field $DR(i,j).UL$, which is defined to be $D(i,j) - D(i-1,j-1)$, will be used later, but it need not be stored in $DR(i,j)$ because it can be computed as $DR(i,j).U + DR(i-1,j).L$. Because the possible values that each of $DR(i,j).U$ and $DR(i,j).L$ can have are $-1, 0$, and 1 [8], we need only four bits to store an entry in the DR-table. It is easy to see that the D-table can be converted to the DR-table in $O(mn)$ time, and vice versa. We can also compute one row (resp. column) of the D-table from the DR-table in $O(n)$ (resp. $O(m)$) time.

In this paper we present an $O(m+n)$-time algorithm for the incremental/decremental edit distance problem. Our result is much simpler and more intuitive than that of Landau et al. [3]. A key tool in our algorithm is the *change table* between the two D-tables before and after an increment/decrement. The change table is not actually constructed in our algorithm, but it is central in understanding our algorithm.

Our result finds a variety of applications. To verify whether a string p is an approximate period of another string x where $|x| = n$ and $|p| = m$, one needs to find the edit distance between p and every substring of x [6]. A naive method that computes a D-table of size $O(m^2)$ for each position of x will take $O(m^2 n)$ time, but our algorithm reduces the time complexity to $O(mn)$ [6]. Other applications include the longest prefix match problem, the approximate overlap problem, the cyclic string comparison problem, and the text screen update problem [3].

This paper is organized as follows. In section 2, we describe the important properties of the change table. In section 3, we present our algorithm for the incremental/decremental edit distance problem.

2 Preliminary Properties

Let Σ be a finite *alphabet* of *symbols*. A *string* over Σ is a finite sequence of symbols in Σ. The length of a string A is denoted by $|A|$. The i-th symbol in

A is denoted by $A[i]$ and the substring consisting of the i-th through the j-th symbols of A is denoted by $A[i..j]$.

Let A and B be strings of lengths m and n, respectively, over Σ, and let $B' = B[2..n]$. Let D be the D-table between A and B and let D' be the D-table between A and B'. Also let DR be the DR-table between A and B and let DR' be the DR-table between A and B'. In this section, we prove the key properties between D and D' that enables us to compute efficiently DR' from DR.

D

		b	b	a	b	a	b	b	a	b
	0	1	2	3	4	5	6	7	8	9
a	1	1	2	2	3	4	5	6	7	8
b	2	1	1	2	2	3	4	5	6	7
a	3	2	2	1	2	2	3	4	5	6
b	4	3	2	2	1	2	2	3	4	5
b	5	4	3	3	2	2	2	2	3	4
a	6	5	4	3	3	2	3	3	2	3
b	7	6	5	4	3	3	2	3	3	2
b	8	7	6	5	4	4	3	2	3	3

D'

		b	a	b	a	b	b	a	b
	0	1	2	3	4	5	6	7	8
a	1	1	1	2	3	4	5	6	7
b	2	1	2	1	2	3	4	5	6
a	3	2	1	2	1	2	3	4	5
b	4	3	2	1	2	1	2	3	4
b	5	4	3	2	2	2	1	2	3
a	6	5	4	3	2	3	2	1	2
b	7	6	5	4	3	2	3	2	1
b	8	7	6	5	4	3	2	3	2

Ch

		b	a	b	a	b	b	a	b
	-1	-1	-1	-1	-1	-1	-1	-1	-1
a	0	-1	-1	-1	-1	-1	-1	-1	-1
b	1	0	0	-1	-1	-1	-1	-1	-1
a	1	0	0	0	-1	-1	-1	-1	-1
b	1	1	0	0	0	-1	-1	-1	-1
b	1	1	0	0	0	0	-1	-1	-1
a	1	1	1	0	0	0	-1	-1	-1
b	1	1	1	1	0	0	0	-1	-1
b	1	1	1	1	0	0	0	0	-1

Fig. 1. An example Ch-table

One key tool in understanding our algorithm is the *change table* (Ch-table for short) from D to D'. Later, when we compute DR' from DR, the first column of DR is discarded and each entry $DR(i, j+1)$, $0 \le i \le m$ and $0 \le j < n$, will be converted to $DR'(i, j)$. Thus, each entry in the Ch-table Ch from D to D' is defined as follows:

$$Ch(i, j) = D'(i, j) - D(i, j+1).$$

The Ch-table is not actually constructed in our algorithm because the initialization of the Ch-table will require $\Theta(mn)$ time. It will be used only for the description of the algorithm. See Fig. 1 for an example Ch-table.

Figure 1 suggests a property of the Ch-table: the entries of value -1 (resp. 1) appear contiguously in the upper-right (resp. lower-left) part of the Ch-table in a *staircase-shaped* region. This property is formally proved in the following series of lemmas.

Lemma 1. *In the Ch-table Ch, the following properties hold.*

1. $Ch(0, j) = -1$ *for all* $0 \le j < n$.
2. $Ch(i, 0) = 0$ *for all* $1 \le i < k$, *where* k *is the smallest index in* A *such that* $A[k] = B[1]$.
3. $Ch(i, 0) = 1$ *for all* $k \le i \le m$.

Proof. Immediate from the definition of the D-table.

Lemma 2. *For* $1 \le i \le m$ *and* $1 \le j < n$, *the possible values of* $Ch(i, j)$ *are in the range* $\min\{Ch(i-1, j-1), Ch(i-1, j), Ch(i, j-1)\} .. \max\{Ch(i-1, j-1), Ch(i-1, j), Ch(i, j-1)\}$.

Proof. Recall that $Ch(i,j)$ is defined to be $D'(i,j) - D(i,j+1)$. By recurrence (1), $D(i,j+1)$ is

$$\min\{D(i-1,j) + \delta_{i,j+1}, D(i-1,j+1) + 1, D(i,j) + 1\}. \tag{2}$$

Also, $D'(i,j)$ is $\min\{D'(i-1,j-1) + \delta'_{ij}, D'(i-1,j) + 1, D'(i,j-1) + 1\}$ where $\delta'_{ij} = 0$ if $A[i] = B'[j]$; $\delta'_{ij} = 1$, otherwise. Because $B'[j]$ is the same symbol as $B[j+1]$, $\delta'_{ij} = \delta_{i,j+1}$. Hence,

$$D'(i,j) = \min \begin{cases} D(i-1,j) + Ch(i-1,j-1) + \delta_{i,j+1} \\ D(i-1,j+1) + Ch(i-1,j) + 1 \\ D(i,j) + Ch(i,j-1) + 1. \end{cases} \tag{3}$$

Note that the only differences between (2) and (3) are additional terms $Ch(i-1,j-1), Ch(i-1,j)$, and $Ch(i,j-1)$ in (3). Assume without loss of generality that the second argument is minimum in (2). If the second argument is minimum in (3), the lemma holds because $Ch(i,j) = Ch(i-1,j)$. Otherwise, assume without loss of generality that the third argument is minimum in (3). Then $Ch(i,j) = D(i,j) + Ch(i,j-1) + 1 - (D(i-1,j+1) + 1)) \geq Ch(i,j-1)$ because the second argument is minimum in (2). Also, $Ch(i,j) \leq Ch(i-1,j)$ because the third argument is minimum in (3).

Corollary 1. *The possible values of* $Ch(i,j)$ *are* $-1, 0$, *and* 1.

Proof. It follows from Lemmas 1 and 2.

Lemma 3. *For each* $0 \leq i \leq m$, *let* $f(i)$ *be the smallest integer* j *such that* $Ch(i,j) = -1$. *(*$f(i) = n$ *if* $Ch(i,j') \neq -1$ *for* $0 \leq j' < n$.*) Then,* $Ch(i,j') = -1$ *for all* $f(i) \leq j' < n$. *Furthermore,* $f(i) \geq f(i-1)$ *for* $1 \leq i \leq m$.

Proof. We use induction on i. When $i = 0$, $f(i) = 0$ and the lemma holds by Lemma 1. Assume inductively that the lemma holds for $i = k$. That is, $Ch(k,j') \neq -1$ for $0 \leq j' < f(k)$ and $Ch(k,j') = -1$ for $f(k) \leq j < n$.

Let $Ch(k+1,l)$ be the first entry in row $k+1$ that is -1. For $Ch(k+1,l)$ to be -1, at least one of $Ch(k,l-1)$ and $Ch(k,l)$ must be -1 by Lemma 2. Thus, we have shown that $l = f(k+1) \geq f(k)$. It is easy to see that $Ch(k+1,l') = -1$ for $f(k+1) < l' < n$ by the inductive assumption, the condition that $f(k+1) \geq f(k)$, and Lemma 2.

The following lemma is symmetric to Lemma 3 and it can be similarly proved.

Lemma 4. *For each* $0 \leq j < n$, *let* $g(j)$ *be the smallest integer* i *such that* $Ch(i,j) = 1$. *(*$g(j) = m+1$ *if* $Ch(i',j) \neq 1$ *for* $0 \leq i' \leq m$.*) Then,* $Ch(i',j) = 1$ *for all* $g(j) \leq i' \leq m$. *Furthermore,* $g(j) \geq g(j-1)$ *for* $1 \leq j < n$.

We say that an entry $Ch(i,j)$ is *affected* if the values of $Ch(i-1,j-1), Ch(i-1,j)$, and $Ch(i,j-1)$ are not the same. We also say that $DR'(i,j)$ is affected if $Ch(i,j)$ is affected.

Lemma 5. *If $DR'(i,j)$ is not affected, then $DR'(i,j)$ equals $DR(i,j+1)$.*

Proof. If $DR'(i,j)$ is not affected, then the value of $Ch(i,j)$ is the same as the common value of $Ch(i-1,j-1), Ch(i-1,j)$, and $Ch(i,j-1)$ by Lemma 2. Then $DR'(i,j).U = D'(i,j) - D'(i-1,j) = D(i,j+1) + Ch(i,j) - (D(i-1,j+1) + Ch(i-1,j)) = DR(i,j+1).U$. Similarly, $DR'(i,j).L = DR(i,j+1).L$.

We say that an entry $Ch(i,j)$ is a (-1)-*boundary* (resp. 1-*boundary*) *entry* if $Ch(i,j)$ is of value -1 (resp. 1) and at least one of $Ch(i,j-1), Ch(i+1,j)$, and $Ch(i+1,j-1)$ (resp. $Ch(i,j+1), Ch(i-1,j)$, and $Ch(i-1,j+1)$) is not of value -1 (resp. 1).

By Lemma 5 we can conclude that in computing DR' from DR, only the affected entries need be changed. See Fig. 1 again. Because the entries whose values are -1 (or 1) appear contiguously in the Ch-table, the affected entries are either (-1)- or 1-boundary entries themselves or appear adjacent to (-1)- or 1-boundary entries. The key idea of our algorithm is to scan the (-1)- and 1-boundary entries starting from the upper-left corner of the DR-table when we compute the affected entries. Lemmas 3 and 4 imply that the number of (-1)- and 1-boundary entries in the DR-table is $O(m+n)$.

3 Boundary Scan Algorithm

In this section we show how to compute DR' from DR. First, we describe how we scan the boundary entries starting from the upper-left corner of the DR'-table within the proposed time complexity. Then, we will mention the modifications to the boundary-scan algorithm which leads to an algorithm that converts DR to DR'.

For simplicity we will use the Ch-table in the description of our algorithm. However, the Ch-table is not explicitly constructed but accessed through the one-dimensional tables $f()$ and $g()$. The details will be given later.

Lemma 6.

$$Ch(i,j) = \min \begin{cases} -DR(i,j+1).UL + Ch(i-1,j-1) + \delta_{i,j+1} \\ -DR(i,j+1).U + Ch(i-1,j) + 1 \\ -DR(i,j+1).L + Ch(i,j-1) + 1 \end{cases}$$

(i.e., $Ch(i-1,j-1), Ch(i-1,j), Ch(i,j-1)$, and $DR(i,j+1)$ are needed to compute $Ch(i,j)$).

Proof. Recall that $Ch(i,j) = D'(i,j) - D(i,j+1)$. Substituting recurrence (1) for $D'(i,j)$ and distributing $D(i,j+1)$ into the min function, we have $Ch(i,j) = \min\{\ldots, D'(i-1,j) - D(i,j+1) + 1, \ldots\}$ (only the second argument is shown). Substituting $D(i-1,j+1) + Ch(i-1,j)$ for $D'(i-1,j)$, the second argument becomes $D(i-1,j+1) - D(i,j+1) + Ch(i-1,j) + 1 = -DR(i,j+1).U + Ch(i-1,j) + 1$. The lemma follows from similar calculations for the first and the third arguments.

Algorithm 1

Let k be the smallest index in A such that $A[k] = B[1]$.
$(i_{-1}, j_{-1}) \leftarrow (0, 1)$; $(i_1, j_1) \leftarrow (k, 0)$; $f(0) \leftarrow 0$; $g(0) \leftarrow k$
$finished_{-1} \leftarrow$ **false**
$finished_1 \leftarrow$ **false**
while not $finished_{-1}$ **or not** $finished_1$ **do**
  if $i_{-1} < i_1 - 1$ **then** {Case 1}
    Compute $Ch(i_{-1} + 1, j_{-1})$. {See Fig. 4.}
    if $Ch(i_{-1} + 1, j_{-1}) = -1$ **then**
      $i_{-1} \leftarrow i_{-1} + 1$; $f(i_{-1}) \leftarrow j_{-1}$
    else
      $j_{-1} \leftarrow j_{-1} + 1$
    fi
  else if $j_1 < j_{-1} - 1$ **then** {Case 2}
    Symmetric to Case 1.
  else {Case 3, $i_1 = i_{-1} + 1$ and $j_1 = j_{-1} - 1$ }
    Compute $Ch(i_{-1} + 1, j_{-1})$. {See Fig. 5.}
    if $Ch(i_{-1} + 1, j_{-1}) = -1$ **then**
      $i_{-1} \leftarrow i_{-1} + 1$; $i_1 \leftarrow i_1 + 1$; $f(i_{-1}) \leftarrow j_{-1}$
    else if $Ch(i_{-1} + 1, j_{-1}) = 1$ **then**
      $j_{-1} \leftarrow j_{-1} + 1$; $j_1 \leftarrow j_1 + 1$; $g(j_1) \leftarrow i_1$
    else
      $j_{-1} \leftarrow j_{-1} + 1$; $i_1 \leftarrow i_1 + 1$
    fi
  fi
  if $i_{-1} = m$ or $j_{-1} = n$ **then** $finished_{-1} \leftarrow$ **true fi**
  if $i_1 = m + 1$ or $j_1 = n - 1$ **then** $finished_1 \leftarrow$ **true fi**
od

Fig. 2. Algorithm 1

Algorithm 1 is the boundary-scan algorithm. In the algorithm, the pair (i_{-1}, j_{-1}) (resp. (i_1, j_1)) indicates that $Ch(i_{-1}, j_{-1})$ (resp. $Ch(i_1, j_1)$) is the current (-1)-boundary (resp. 1-boundary) entry that is being scanned. The following property holds for $Ch(i_{-1}, j_{-1})$ and $Ch(i_1, j_1)$ by Lemmas 3 and 4. See Fig. 3 for an illustration.

Property 1.

1. $Ch(i, j) \neq -1$ if $i > i_{-1}$ and $j < j_{-1}$.
2. $Ch(i, j) \neq 1$ if $i < i_1$ and $j > j_1$.

In one iteration of the loop in Algorithm 1, one or both of the current boundary entries are moved to the next boundary entries. For example, the current (-1)-boundary entry is moved to the next (-1)-boundary entry which can be down or to the right of the current (-1)-boundary entry. We maintain the following invariants in each iteration of Algorithm 1.

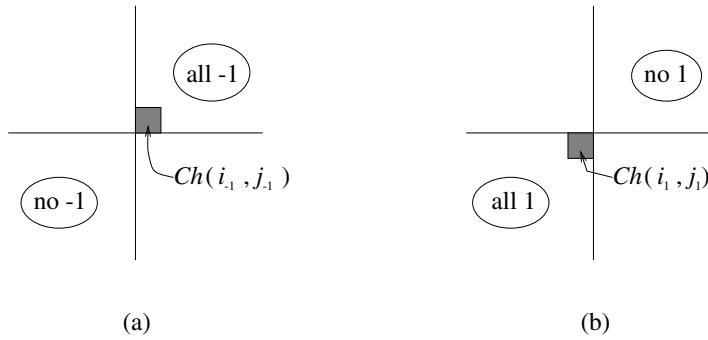


Fig. 3. Boundary entry conditions

Invariant 1

1. $i_{-1} < i_1$ *and* $j_{-1} > j_1$.
2. *All values of* $f(0), \ldots, f(i_{-1})$ *are known.*
3. *All values of* $g(0), \ldots, g(j_1)$ *are known.*

One iteration of Algorithm 1 has three cases. Case 1 applies when the current (-1)-boundary can be moved by one entry (down or to the right) without violating Invariant 1.1. Case 2 applies when the current 1-boundary can be moved by one entry (down or to the right) without violating Invariant 1.1. Case 3 applies when moving the (-1)-boundary entry down by one entry or moving the 1-boundary entry to the right by one entry will violate Invariant 1.1, and thus both boundary entries have to be moved simultaneously. What Algorithm 1 does in each case is described in Fig. 2.

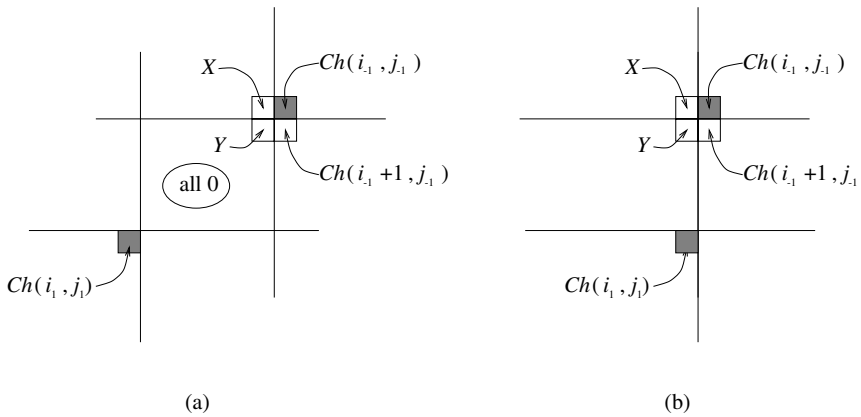


Fig. 4. Case 1

What remains to show is the methods to obtain the values of the Ch-table entries that are used to compute a new Ch-table entry, e.g., $Ch(i_{-1}+1, j_{-1})$ in Case 1. The two subcases for Case 1 are depicted in Fig. 4. The first subcase is when $j_{-1} > j_1 + 1$. See Fig. 4 (a). The unknown values of the Ch-table entries are X and Y. By Invariant 1.2 the value of $f(i_{-1})$ is known. If $f(i_{-1}) < j_{-1}$, then $X = -1$. Otherwise ($f(i_{-1}) = j_{-1}$), $X = 0$ because X is not 1 by Property 1.1. It is easy to see that $Y = 0$ because Y is inside the region in which there are no (-1)'s (by Property 1.1) and no 1's (by Property 1.2). The second subcase is when $j_{-1} = j_1 + 1$. See Fig. 4 (b). We can compute the value of X as -1 if $f(i_{-1}) < j_{-1}$; 1 if $g(j_1) \leq i_{-1}$; 0, otherwise. We know that $Y \neq -1$ by Property 1.1. Thus, $Y = 1$ if $g(j_1) \leq i_{-1} + 1$; $Y = 0$, otherwise. Case 3 is depicted in Fig. 5. The value of X can be computed as we computed the value of X in the second subcase of Case 1.

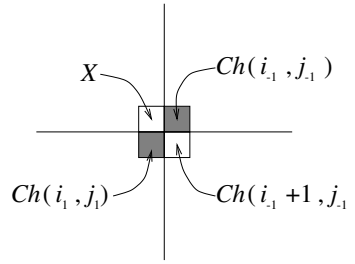


Fig. 5. Case 3

We now show that all affected Ch-table entries are computed by Algorithm 1. It is easy to see that each affected entry $Ch(i, j)$, $1 \leq i \leq m$ and $1 \leq j < n$, falls into one of the following types by Lemmas 3 and 4. For each of the types we can easily check which cases in our algorithm compute $Ch(i, j)$.

1. $Ch(i, j)$ is a (-1)-boundary entry such that $Ch(i, j-1) \neq -1$: $Ch(i, j)$ is computed by Case 1 if $Ch(i, j-1) = 0$; by Case 3, otherwise.
2. $Ch(i, j)$ is an 1-boundary entry such that $Ch(i-1, j) \neq 1$: $Ch(i, j)$ is computed by Case 2 if $Ch(i-1, j) = 0$; by Case 3, otherwise.
3. $Ch(i, j) = 0$ and either $Ch(i-1, j) = -1$ or $Ch(i, j-1) = 1$: $Ch(i, j)$ is computed by Case 1 if $Ch(i, j-1) = 0$; by Case 2 if $Ch(i-1, j) = 0$; by Case 3, otherwise.

To compute DR' from DR, we first discard the first column from DR. Then, we run a modified version of Algorithm 1. The modifications to Algorithm 1 is to compute $DR'(i, j)$ whenever we compute the value of $Ch(i, j)$. Once $Ch(i, j)$ is computed using Lemma 6, the fields in $DR'(i, j)$ can be easily computed. That is, $DR'(i, j).L = DR(i, j+1).L + Ch(i, j) - Ch(i, j-1)$ and $DR'(i, j).U = DR(i, j+1).U + Ch(i, j) - Ch(i-1, j)$.

We can easily check that one iteration of the loop takes only constant time and that it increases at least one of i_{-1}, j_{-1}, i_1, j_1 by one. Hence, the time complexity of our algorithm is $O(m+n)$.

Theorem 1. *Let A and B be two strings of lengths m and n, respectively, and $B' = B[2..n]$. Given the difference representation DR between A and B, the difference representation DR' between A and B' can be computed in $O(m+n)$ time.*

References

1. Galil, Z. and Giancarlo, R.: Data Structures and Algorithms for Approximate String Matching. J. Complexity 4 (1988) 33–72
2. Galil, Z. and Park, K.: An Improved Algorithm for Approximate String Matching. SIAM J. Computing, Vol. 19, No. 6 (1990) 989–999
3. Landau, G. M., Myers, E. W., and Schmidt, J. P.: Incremental String Comparison. SIAM J. Computing, Vol. 27, No. 2 (1998) 557–582
4. Landau, G. M. and Vishkin, U.: Fast String Matching with k Differences. J. Comput. System Sci., 37 (1998) 63–78
5. Landau, G. M. and Vishkin, U.: Fast Parallel and Serial Approximate String Matching. J. Algorithms, 10 (1989) 157–169
6. Sim, J. S., Iliopoulos, C. S., Park, K., and Smyth, W. F.: Approximate Periods of Strings. In 10th Annual Symposium, Combinatorial Pattern Matching '99 (1999) 123–133
7. Ukkonen, E.: Algorithms for Approximate String Matching. Inform. and Control, 64 (1985) 100–118
8. Ukkonen, E.: Finding Approximate Patterns in Strings. J. of Algorithms, 6 (1985) 132–137
9. Ukkonen, E. and Wood, D.: Approximate String Matching with Suffix Automata. Algorithmica, 10 (1993) 353–364
10. Wagner, R. A. and Fisher, M. J.: The String-to-string Correction Problem. J. Assoc. Comput. Mach., 21 (1974) 168–173

Parametric Multiple Sequence Alignment and Phylogeny Construction

David Fernández-Baca[1]*, Timo Seppäläinen[2]**, and Giora Slutzki[1]

[1] Department of Computer Science, Iowa State University, Ames, IA 50011
{fernande,slutzki}@cs.iastate.edu.
[2] Department of Mathematics, Iowa State University, Ames, IA 50011
seppalai@iastate.edu

Abstract. Bounds are given on the size of the parameter-space decomposition induced by multiple sequence alignment problems where phylogenetic information may be given or inferred. It is shown that many of the usual formulations of these problems fall within the same integer parametric framework, implying that the number of distinct optima obtained as the parameters are varied across their ranges is polynomially bounded in the length and number of sequences.

1 Introduction

Multiple sequence comparison is among the most basic computational problems in biology, serving as an aid to identifying common structure and function. The problem is, in a way, simply a generalization of pairwise sequence comparison, a question that has been studied extensively (see [19]). However, when three or more sequences are involved, new issues arise, making definitions more complex and the associated problems harder to solve efficiently [33]. One of these new facets is the role of evolutionary relationships between sequences. One possibility is to disregard these relationships, at least to a certain extent, by comparing all sequences against each other — the *sum-of-pairs* approach [5] is one example. Using or attempting to infer evolutionary relationships leads to a host of new problems. In one problem, called *phylogenetic alignment*, the input is a tree whose leaves are labeled by sequences and the objective is to find a labeling of the internal nodes that minimizes the total length of the tree, which is the sum of the (evolutionary) distances between adjacent sequences in the tree [30]. In another problem, *generalized phylogenetic alignment*, the input is a set $\mathcal{S}$ of sequences and one must find a sequence-labeled tree of minimum length wherein each element of $\mathcal{S}$ labels a distinct leaf in the tree [26]. Sum-of-pairs multiple alignment, phylogenetic alignment, generalized phylogenetic alignment, and some of their variants are known to be NP-hard [26,25]. However, the first two

* Part of this work was conducted while the author was on leave at the University of California, Davis. Supported in part by the National Science Foundation under grant CCR-9520946.
** Supported in part by the National Science Foundation under grant DMS-9801085.

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 69–83, 2000.

can be solved in time polynomial in the lengths of the sequences and exponential in their number. Implementations of several of these methods, often relying on heuristics, are available [27,15,11].

The optimum solution to a multiple alignment problem depends on the various parameters used to compute inter-sequence distance or similarity — e.g., the weights of mismatches and spaces. These are not always easy to choose [36]. One approach is *parametric analysis* [16]; i.e., to examine the space of all parameter choices to determine the set of all optimum solutions. This question has been studied for pairwise sequence comparison [13,20,22,24,34,35], to a lesser extent for sum-of-pairs multiple alignment [13,32], and hardly at all for phylogenetic alignment — see, however, [36]. Here we explore parametric multiple alignment and phylogenetic alignment.

One objective of parametric analysis is to establish upper bounds on the number of distinct *optimality regions*, i.e., maximal connected regions of the parameter space such that, within each region a single solution is optimal. Gusfield et at. [20], obtained the first results, proving, among other things, a $O(n^{2/3})$ upper bound on the number of regions for two-sequence global alignment, the shorter of which has length n. This was extended to obtain a $O(n^{2/3}k^{4/3})$ upper bound for the parametric sum-of-pairs alignment of k sequences of length n [13]. In the present paper, we argue that many multiple alignment schemes fall within the same "integer parametric" framework. This leads to the somewhat surprising result that the number of optimality regions for these problems is polynomial in both the lengths of the sequences and their number, even if the scoring is alphabet-dependent. Our bounds are consequences of the following observation: While the number of potential phylogenies and sequences labeling them is exponentially large, any scoring system based on linear functions whose coefficients are themselves functions of discrete features of alignments (e.g, number of mismatches, spaces, etc.), only allows a polynomially-bounded number of distinct cost functions to be optimal. The techniques used are uniform and straightforward; once the problems are formulated properly, the common structure becomes evident. Better bounds might be obtainable by a tighter analysis of our framework; however, we suspect that significant advances will require deeper understanding of the combinatorial structure of the individual problems.

Main Results. The alignment problems studied here are classified according to whether the scoring is (i) local or global, (ii) distance-based or similarity-based, (iii) alphabet-dependent or -independent, (iv) gap-length dependent or not. The input consists of k sequences, which, for simplicity, are assumed to have the same length n. Our results include the following bounds on the number of optimality regions.

1. A $O(n^{2/3}p^{2/3})$ bound for global multiple alignment under sum-of-pairs alphabet-independent similarity and distance scoring, with zero gap penalty, when the induced alignment of only p of the $\binom{k}{2}$ possible pairs is considered for computing the total score.
2. A $O(n^{5/3}p^{5/3})$ bound for the previous problem when the gap penalty varies. For the pairwise case ($p = 1$), this improves on earlier $O(n^2)$ bound by

Gusfield et al. A $O(n^{5/3}p^{5/3})$ bound also holds for the local case when the gap penalty is zero. For the pairwise case, this improves on a $O(n^2)$ bound by Gusfield et al.

3. A $O(n^{2/3}k^{4/3})$ bound for phylogenetic and generalized phylogenetic alignment under distance-based global alphabet-independent scoring with zero gap penalty. The bound goes up to $O(n^{5/3}k^{10/3})$ when the gap penalty is variable.
4. A $O(n^{2/3}k^{2/3})$ bound for star alignment (a special case of tree alignment) under global alphabet-independent distance-based scoring with fixed zero gap penalty. This increases to $O(n^{5/3}k^{5/3})$ when the gap penalty is allowed to vary.
5. Polynomial bounds for sum-of-pairs, tree alignment, and generalized tree alignment problems under alphabet-dependent, global or local, similarity or distance scoring.

Organization of the Paper. The main problems studied here, as well as some of their properties, are defined in Section 2. Section 3 discusses parametric analysis in a general context and consists of two parts. First, we obtain an upper bound on the number of optimality regions for parametric problems satisfying certain integrality conditions. Second, we describe a general approach to generating parameter-space decompositions. Parametric multiple alignments are discussed in Section 4. Section 5 presents some conclusions and open problems.

2 Preliminaries

We now give formal definitions and prove some of the basic properties of the problems whose parametric versions we shall study. The first part of this section introduces distance and similarity measures based on pairwise alignments. These notions are the basis for the scoring schemes used in multiple sequence comparison, which are discussed in the second part. In what follows, Σ will denote an alphabet that includes a special *space* character "-". All input strings are assumed to be over $\Sigma \setminus \{-\}$.

2.1 Pairwise Alignments

An *alignment* between two strings S_1 and S_2 is a pair of equal-length strings $\mathcal{A} = (S_1', S_2')$ where S_1' and S_2' are obtained by inserting space characters into S_1 and S_2 respectively, so that that there is no character position in which both S_1' and S_2' have spaces. A *match* is a position in which S_1' and S_2' have the same character. A *mismatch* is a position in which S_1' and S_2' have different characters, neither of which is a "-". An *indel* is a position in which one of S_1' and S_2' has a "-". A *gap* is a sequence of one or more consecutive spaces in S_1' or S_2'. Collectively, we call the matches, mismatches, indels, and gaps the *features* of $\mathcal{A}$. These features are used to compute the *value* of $\mathcal{A}$ according to a certain *scoring scheme*. In *local* scoring schemes, the goal is to locate highly similar substrings. In *global* schemes, the entire input strings are taken into account.

Global Alignment. We first consider alphabet-independent scoring schemes. Let $w_{\mathcal{A}}$, $x_{\mathcal{A}}$, $y_{\mathcal{A}}$, and $z_{\mathcal{A}}$ denote, respectively, the number of matches, mismatches, indels, and gaps in an alignment $\mathcal{A}$, and let α, β, and γ be the mismatch, indel, and gap penalties, respectively. Penalties are assumed to be nonnegative.

The similarity value of an alignment $\mathcal{A}$ is given by

$$\boldsymbol{\sigma}_{\mathcal{A}} = w_{\mathcal{A}} - \alpha x_{\mathcal{A}} - \beta y_{\mathcal{A}} - \gamma z_{\mathcal{A}}. \tag{1}$$

The global *similarity* between sequences S_1 and S_2 is defined as

$$sim(S_1, S_2) = \max\{\boldsymbol{\sigma}_{\mathcal{A}} : \mathcal{A} \text{ is an alignment of } S_1 \text{ and } S_2\}. \tag{2}$$

The distance value of an alignment $\mathcal{A}$ is

$$\boldsymbol{\delta}_{\mathcal{A}} = \alpha x_{\mathcal{A}} + \beta y_{\mathcal{A}} + \gamma z_{\mathcal{A}}. \tag{3}$$

The *distance* between S_1 and S_2 is

$$dist(S_1, S_2) = \min\{\boldsymbol{\delta}_{\mathcal{A}} : \mathcal{A} \text{ is an alignment of } S_1 \text{ and } S_2\}. \tag{4}$$

Schemes (1) and (3) are related by the lemma below, in which the mismatch, indel, and gap penalties are given by triples (α, β, γ).

Lemma 1. *Under global alphabet-independent scoring,*

$$\boldsymbol{\sigma}_{\mathcal{A}}(\alpha, \beta, \gamma) = \frac{n+m}{2} - \boldsymbol{\delta}_{\mathcal{A}}(\alpha + 1, \beta + 1/2, \gamma).$$

Therefore, a pairwise alignment has maximum similarity score at (α, β, γ) if and only if it has minimum distance score at $(\alpha + 1, \beta + 1/2, \gamma)$.

Proof. $\mathcal{A}$ satisfies $2w_{\mathcal{A}} + 2x_{\mathcal{A}} + y_{\mathcal{A}} = n + m$ [20]. Thus, the similarity score of $\mathcal{A}$ (1) can be re-expressed as

$$\begin{aligned}\boldsymbol{\sigma}_{\mathcal{A}}(\alpha, \beta, \gamma) &= \frac{n+m}{2} - (\alpha + 1)x_{\mathcal{A}} - \left(\beta + \frac{1}{2}\right) y - \gamma z \\ &= \frac{n+m}{2} - \boldsymbol{\delta}_{\mathcal{A}}(\alpha + 1, \beta + 1/2, \gamma),\end{aligned}$$

where the second line follows from the definition of distance score (3). The rest of the lemma follows immediately. □

Note that we can assume without loss of generality that the mismatch penalty α in (3) is one, since changing its value only affects the magnitude, but not the relative values, of the alignments. Thus, there are effectively only two parameters to be chosen for distance scoring. By Lemma 1, this is also true for similarity scoring.

Alphabet-dependent scoring schemes depend on a symmetric $|\Sigma| \times |\Sigma|$ *substitution matrix* α, where $\alpha(s, t)$ is the cost of lining up character s with character

t. Widely-used families of matrices for protein alignment are PAM [6] and BLOSUM [23]. The similarity score of an alignment $\mathcal{A}$ is now given by

$$\boldsymbol{\sigma}_{\mathcal{A}} = -\gamma z + \sum_{\{s,t\} \subseteq \Sigma} \alpha(s,t) \cdot x(s,t) \tag{5}$$

where $x(s,t)$ is the number of times character s is lined up with character t in $\mathcal{A}$ and z is the number of gaps in $\mathcal{A}$. The similarity between two sequences is obtained by applying this scoring scheme in (1).

The alphabet-dependent distance score of $\mathcal{A}$, $\boldsymbol{\delta}_{\mathcal{A}}$, can be defined in the same way as (5), except that the "$-\gamma z$" term is replaced by "$+\gamma z$." For both similarity and distance, the total number of parameters is $(|\Sigma|^2 + |\Sigma|)/2 + 1$: the number of entries in the substitution matrix plus the gap penalty.

Local Alignment. For two strings S and R, we write $S \sqsubseteq R$ if S is a substring of R. The *local* similarity between S_1 and S_2, denoted $sim_L(S_1, S_2)$, is

$$sim_L(S_1, S_2) = \max\{\boldsymbol{\sigma}_A : \mathcal{A} \text{ is an alignment of } S_1' \sqsubseteq S_1 \text{ with } S_2' \sqsubseteq S_2\}. \tag{6}$$

The scoring scheme used in the definition above may be alphabet-dependent or -independent. Note that the global similarity between two strings is a lower bound on their local similarity. Note also that while it is straightforward to define local distance measures, using minimization and a scoring scheme, say, like (3), the alphabet-independent versions of distance and similarity are *not* related by Lemma 4. Thus, even though (alphabet-independent) local distance depends on two parameters, local similarity depends on three. For the alphabet-dependent case, we still have $(|\Sigma|^2 + |\Sigma|)/2 + 1$ parameters.

Bounds on the Features of Alignments. We will need the following facts about pairwise alignment, which were proved in [20]. As before $w_{\mathcal{A}}$, $x_{\mathcal{A}}$, $y_{\mathcal{A}}$, and $z_{\mathcal{A}}$ denote the number of matches, mismatches, indels, and gaps in an alignment $\mathcal{A}$. Let n and m denote the lengths of the input strings, where $n \leq m$.

Lemma 2. *For any pairwise global or local alignment $\mathcal{A}$, $w_{\mathcal{A}} + x_{\mathcal{A}} \leq n$.*

Lemma 3. *For any global and local alignment $\mathcal{A}$, $y_{\mathcal{A}}, z_{\mathcal{A}} \leq m + n$. Moreover, if $\mathcal{A}$ is global, $y_{\mathcal{A}} \geq m - n$.*

2.2 Multiple Alignments

A *multiple alignment* $\mathcal{A}$ of strings $S_1, \ldots, S_k$, where S_i has length n_i, is obtained by inserting spaces in each string to obtain strings of the same length l. The result is a matrix with k rows and l columns, such that each character and space of each string appears in exactly one column. $\mathcal{A}$ induces a pairwise alignment of S_i and S_j in a natural way: remove all rows of $\mathcal{A}$ except those corresponding to S_i and S_j and strike out any columns containing two spaces. This will be called the *induced pairwise alignment* of S_i and S_j.

The following generalization of two-sequence alignment was considered in [4,3]; it is used in the `MSA` package for multiple sequence alignment [27,15].

Weighted Sum-of-Pairs Alignment (Similarity Version)
Input: A set of sequences $\mathcal{S} = \{S_1, \ldots, S_k\}$ and a $k \times k$ matrix $B = [b_{ij}]$.
Question: Find a multiple alignment $\mathcal{A}$ for $\mathcal{S}$ maximizing $\sum_{i<j} b_{ij}\boldsymbol{\sigma}_{\mathcal{A}}(S_i, S_j)$, where $\boldsymbol{\sigma}_{\mathcal{A}}(S_i, S_j)$ is the similarity score of the pairwise alignment between S_i and S_j induced by $\mathcal{A}$.

The scoring scheme can be global or local, alphabet-dependent or independent. Distance versions of this problem can be defined in the obvious way, using minimization instead of maximization and appropriate scoring schemes. The following is a direct consequence of Lemma 1.

Lemma 4. *Under global alphabet-independent scoring, a multiple alignment has maximum similarity score at* (α, β, γ) *if and only if it has minimum distance score at* $(\alpha + 1, \beta + 1/2, \gamma)$.

Thus, for the global alphabet-independent case, the score is a function of only two parameters, the indel and gap penalties. For the local alphabet-independent case, the score is a function of two parameters for distance measures and three for similarity measures (since the mismatch penalty must be considered, in addition to the indel and gap penalties). For the alphabet-dependent case, the score is still a function of $(|\Sigma|^2 + |\Sigma|)/2 + 1$ parameters.

Gaps raise a problematic issue in multiple alignments. A natural approach is to consider as gaps only those that arise in the induced pairwise alignments. However, computing optimum alignments in this way would be too time-consuming [2,15]. As an alternative, Altschul proposed "quasi-natural" gap penalties [3], a scheme later integrated into `MSA` [15]. We will not elaborate on this issue, except to state that it will not impact the parametric analysis of Section 4 significantly.

In the next two families of problems, evolutionary history is used and/or inferred. We define them for distance measures; similarity versions can be defined in the obvious way. Alphabet-dependent or -independent scoring can be used.

We need some definitions. A *phylogeny* for a set of sequences $\mathcal{S}$ is a tree T where every internal node has degree at least three, each element of $\mathcal{S}$ labels a distinct leaf of T, and no leaf of T is unlabeled. An *internal labeling* for T is an assignment of sequences over $\Sigma \setminus \{\texttt{-}\}$ to the internal nodes of T. The *length* of a tree T whose vertices are labeled by sequences is the sum of the pairwise distances between the labels of adjacent nodes.

Phylogenetic Alignment
Input: A phylogeny T for a set of sequences $\mathcal{S}$.
Question: Find an internal labeling for T that minimizes the total length of the resulting tree.

While this problem is NP-complete [25], it can be solved in time that is exponential in the number of sequences [30,29,31]. An important special case is *star alignment*, where the tree T has only one internal node [5].

Given a solution to the phylogenetic alignment problem, one can derive a multiple alignment $\mathcal{A}$ for the sequences labeling the phylogeny that is *consistent*

with it in the following sense: The value of the induced pairwise alignment for any two sequences equals the distance between them [19]. One can obtain a multiple alignment for $\mathcal{S}$ by striking out the rows of $\mathcal{A}$ that do not correspond to elements of $\mathcal{S}$. However, the labels on the internal nodes can be valuable, as they represent hypothetical ancestors to the elements of $\mathcal{S}$.

The next problem, which is MAX SNP-hard [26], is related to phylogenetic alignment, but the tree itself is not given.

Generalized Phylogenetic Alignment
Input: A set of sequences $\mathcal{S}$.
Question: Find a minimum-length internally-labeled phylogeny for $\mathcal{S}$.

Phylogenetic alignment and its generalized version have similarity variants where the goal is to find a solution that maximizes the similarity score.

To prevent spurious matches between internal nodes, we make the following assumption about internally-labeled phylogenies T: Let $\mathcal{A}$ be a multiple alignment consistent with the labels of T. Then, each column in $\mathcal{A}$ contains at least one character from one of the sequences in $\mathcal{S}$.

One distinction between phylogenetic and SP alignment is that similarity and distance measures are *not* equivalent in either the local or global cases. This equivalence for the SP problem is in part a consequence of knowing in advance the lengths of the strings involved in the pairwise comparisons. This is not true for phylogeny problems. Thus, for global and local alphabet-independent phylogenetic alignment, the score is a function of two parameters for distance measures and three for similarity measures. For the alphabet-dependent case, the score is a function of $(|\Sigma|^2 + |\Sigma|)/2 + 1$ parameters. We can, however, still establish useful bounds on the number of features.

Lemma 5. *For any alignment $\mathcal{A}$ of a set $\mathcal{S}$ of k sequences of length n to a phylogeny with r internal nodes, $w_{\mathcal{A}} + x_{\mathcal{A}} \leq nkr$ and $z_{\mathcal{A}} \leq y_{\mathcal{A}} \leq nk(k + 2r - 1)$.*

Proof. Let T be the input phylogeny for $\mathcal{S}$. Each unlabeled node in T is assigned a sequence of length at most nk, since each of its characters must line up with a character in some sequence in $\mathcal{S}$. By Lemma 2, the total number of matches and mismatches in the induced pairwise alignment between any two adjacent sequences in T is at most equal to the length of the shorter sequence. Thus, the the contribution of an edge in T to the total number of matches and mismatches is n if one endpoint is an element of $\mathcal{S}$ and at most nk if neither endpoint is in $\mathcal{S}$. The total number of edges in the latter category is at most $r - 1$, while the number of edges in the former category is at most k. This establishes the bound for $w_{\mathcal{A}} + x_{\mathcal{A}}$.

To bound $y_{\mathcal{A}}$ and $z_{\mathcal{A}}$, we use Lemma 3, noting that edges where one endpoint is in $\mathcal{S}$ contribute at most $n + nk$ indels, while those where neither endpoint is in $\mathcal{S}$ contribute at most $2nk$ to the total. □

For star alignments, we have a better bound, whose proof is omitted.

Lemma 6. *Let $\mathcal{A}$ be an optimal star alignment under alphabet-independent distance-based global scoring. Then $y_{\mathcal{A}} \leq kn$ and $z_{\mathcal{A}} \leq k$.*

3 Parametric Analysis

In this section, we consider two issues that arise in parametric analysis: finding the number of distinct optimal solutions attained as the parameters are varied across their range and generating all of these solutions. We study these questions within a framework that encompasses a broad class of problems that include various alignment problems. We first need some definitions.

In an *affine parametric combinatorial optimization problem*, the cost of each element $\mathbf{x}$ of the set $X \subseteq \mathbb{R}^{d+1}$ of feasible solutions is an affine function $f_{\mathbf{x}}$ of a parameter vector $\lambda = (\lambda_1, \ldots, \lambda_d)$. The *fixed parameter problem* is to compute

$$F(\lambda) = \min_{\mathbf{x} \in X} f_{\mathbf{x}}(\lambda). \tag{7}$$

We write "min" in the definition above for concreteness; the concepts and results to follow have analogs for maximization problems.

Since it is the lower envelope of a set of affine functions, F is piecewise affine. F induces a partition of $\mathbb{R}^d$ into d-dimensional convex polyhedral *optimality regions*, such that $F(\lambda)$ is attained by a single function $f_{\mathbf{x}}$ for all λ in the interior of each such region [1]. This subdivision of $\mathbb{R}^d$ is known as the *minimization diagram* of F. Note that, while a single *cost function* attains the optimal value for each region, there might be several feasible solutions $\mathbf{x}$ with the same cost function that are co-optimal within the region.

3.1 The Number of Optimality Regions

We now prove upper bounds on the number of optimality regions for parametric problems of the form (7) where the cost of a feasible solution $\mathbf{x} = (x_0, \ldots, x_d) \in X$ is given by $f_{\mathbf{x}}(\lambda) = x_0 + \sum_{i=1}^{d} \lambda_i x_i$. Note that the alignment scoring schemes described in Section 2 are of this form.

The following result is implicit in the work of Gusfield et al. [21,20].

Lemma 7. *If $X \subseteq \{0, \ldots, N\}^2$ for some nonnegative integer N, then $F(\lambda)$ induces $O(N^{2/3})$ optimality regions in $\mathbb{R}$.*

Proof. We rely on the following fact, which is shown in [21]:

(*) Let $\{a_i/b_i\}_{1 \le i \le k}$ be a set of (distinct) irreducible fractions with positive numerators and denominators such that $\sum_{i=1}^{k} a_i, \sum_{i=1}^{k} b_i \le N$. Then $k = O(N^{2/3})$.

Let us denote a feasible solution $\mathbf{x}$ by (x, y) and its cost by $f_{\mathbf{x}}(\lambda) = x + \lambda y$. $F(\lambda)$ is a non-decreasing piecewise affine function consisting of a sequence of line segments. Hence, if (x_i, y_i) and (x_{i+1}, y_{i+1}) denote the intercept and slope of the ith and $(i+1)$st segments of F, $x_i < x_{i+1}$ and $y_i > y_{i+1}$.

The λ-value of the meeting point between the ith and $(i+1)$st segments of F is $\Delta x_i / \Delta y_i$, where $\Delta x_i = x_{i+1} - x_i$ and $\Delta y_i = y_i - y_{i+1}$. Thus, $\Delta x_1/\Delta y_1 < \Delta x_2/\Delta y_2 < \cdots < \Delta x_s/\Delta y_s$. Since the numerators and denominators of these fractions are nonnegative integers and $\sum_{i=1}^{s} \Delta x_i, \sum_{i=1}^{s} \Delta y_i \le N$, (*) applies, implying that F has $O(N^{2/3})$ optimality regions. □

Lemma 8. *If X is a subset of $\{0,\ldots,N\}^{d+1}$, where N is a positive integer, then $F(\lambda)$ induces $O(N^{d-1/3})$ optimality regions in $\mathbb{R}^d$.*

Proof. By induction on d. The basis, $d = 1$, follows from Lemma 7. For $d > 1$, define $X_j = \{\mathbf{x} \in X : x_d = j\}$ and $h_{\mathbf{x}}(\lambda) = x_0 + \sum_{i=1}^{d-1} \lambda_i x_i$. Then, we can express F as

$$F(\lambda) = \min_{j=0,\ldots,N} \left(\left(\min_{\mathbf{x}\in X_j} h_{\mathbf{x}}(\lambda) \right) + \lambda_d j \right)$$

By hypothesis, $F'_j(\lambda) = \min_{\mathbf{x}\in X_j} h_{\mathbf{x}}(\lambda)$ induces $O(N^{(d-1)-1/3})$ regions. It can be verified that $g_j(\lambda) = F'_j(\lambda) + \lambda_d j$ induces a subdivision of $\mathbb{R}^d$ into $O(N^{(d-1)-1/3})$ cylinders whose boundary lines are parallel to the λ_d-axis. Since F is the lower envelope of $N+1$ such g_i's, it induces $O(N \cdot N^{(d-1)-1/3}) = O(N^{d-1/3})$ optimality regions in $\mathbb{R}^d$. □

The following observation generalizes a result in [20].

Lemma 9. *Suppose that $X \subset A_0 \times \cdots \times A_d$, where each A_i is a set of N_i distinct real values. Then, $F(\lambda)$ induces at most $\left(\prod_{i=0}^{d} N_i\right) / \max_{0\le i\le d} N_i$ optimality regions in $\mathbb{R}^d$.*

Proof. Assume without loss of generality that $N_0 = \max_{0\le i\le d} N_i$. Consider any $\mathbf{x}, \mathbf{y} \in X$ such that $F(\lambda') = f_{\mathbf{x}}(\lambda')$ and $F(\lambda'') = f_{\mathbf{y}}(\lambda'')$ for some λ', λ''. If $x_0 < y_0$, then we must have $x_i \neq y_i$ for some $i \in \{1,\ldots,d\}$, for otherwise $F(\lambda) < f_{\mathbf{y}}(\lambda)$ for all λ. Thus, out of all $(d+1)$-tuples $(x_0,\ldots,x_d)$ whose last d entries are equal, at most one is associated with a feasible solution that is optimal at some point. Hence, there are at most $\prod_{i=1}^{d} N_i$ functions that are optimal at some point, which also bounds the number of regions of F. □

3.2 Constructing the Minimization Diagram

Algorithms for constructing the minimization diagram of parametric problems have been proposed before (see, e.g., [10,17,14]), mostly for the one- and two-parameter cases. Here we sketch an approach that appears to be part of the folklore[1], but deserves to be more widely known.

An *evaluation* of F at λ consists of finding the solution $\mathbf{x} \in X$ such that $F(\lambda) = f_{\mathbf{x}}(\lambda)$. Evaluating $F(\lambda)$ is equivalent to computing the equation of the supporting hyper-plane of the set $B_F = \{(\lambda_1,\ldots,\lambda_d,z) : z \le F(\lambda)\}$ at the point $(\lambda_1,\ldots,\lambda_d,F(\lambda))$. This operation has been called a *hyper-plane probe* by Dobkin et al. [7,8], who studied the problem of reconstructing a convex object from a sequence of such probes. Constructing B_F (or, equivalently, F) from repeated evaluations of F is one instance of this problem.

Theorem 1 (Dobkin et al. [7,8]). *F can be computed with $O(m + dv)$ evaluations, where m and v are, respectively, the number of optimality regions and vertices of the minimization diagram.*

[1] Naoki Katoh, personal communication

For brevity, we omit the details of the probing algorithm, which is explained fully in [7,8]. The 1- and 2-parameter algorithms of [10] and [14] can be viewed as special cases. The probing process returns successive elements of a set H of half-spaces in $\mathbb{R}^d$ whose intersection equals B_F. Actually generating F requires computing this intersection. A good practical algorithm to do so is the beneath-beyond method [28], whose run time is $O(s|H|)$, where s is the size of the output. It is a consequence of the Upper Bound Theorem [9] that if there are m optimality regions, $s = O(m^{\lceil d/2 \rceil})$, leading to a worst-case bound of $O(m^{\lceil d/2 \rceil + 1})$.

4 Parametric Multiple Alignments

We now present upper bounds on the number of optimality regions for the multiple alignment problems of Section 2. To a certain degree, the results are independent of the kind of alignment problem we are dealing with, as long as we have bounds on the number of distinct values for the coefficients of the objective functions. The arguments are similar: We first show how the problem falls within the scope of Lemmas 7, 8 or 9 and then invoke the appropriate bound.

4.1 SP Alignments

Our first results concern sum-of-pairs (SP) alignments. We assume that the weight matrix is fixed. We first consider 0-1 weight matrices, a problem we refer to as *0-1 SP alignment*. For the next three results, p denotes the number of non-zero entries in the weight matrix. Given an multiple alignment $\mathcal{A}$, w_{ij}, x_{ij}, y_{ij}, and z_{ij} denote the number of matches, mismatches, indels, and gaps in the induced pairwise alignment for sequences i and j. By Lemmas 2 and 3, each of these values is at most $2n$. Thus,

$$0 \leq \sum_{1 \leq i < j \leq k} b_{ij} w_{ij}, \sum_{1 \leq i < j \leq k} b_{ij} x_{ij}, \sum_{1 \leq i < j \leq k} b_{ij} y_{ij}, \sum_{1 \leq i < j \leq k} b_{ij} z_{ij} \leq 2np \qquad (8)$$

Theorem 2. *The number of optimality regions for* alphabet-independent *parametric 0-1 SP alignment under* global *similarity or distance measures is*

(a) $O(n^{2/3}p^{2/3})$ *if the gap penalty is zero and*
(b) $O(n^{5/3}p^{5/3})$ *if the gap penalty is variable.*

Proof. By Lemma 4 it suffices to consider distance measures. In this case, the distance value of a multiple alignment $\mathcal{A}$ is

$$\boldsymbol{\delta}_{\mathcal{A}} = \sum_{1 \leq i < j \leq k} b_{ij} x_{ij} + \beta \sum_{1 \leq i < j \leq k} b_{ij} y_{ij} + \gamma \sum_{1 \leq i < j \leq k} b_{ij} z_{ij}, \qquad (9)$$

Now, parts (a) and (b) follow by applying Lemma 8 for $d = 1$ and $d = 2$, respectively, with $N = 2np$, where the latter is valid by (8). □

Theorem 3. *The number of optimality regions for* alphabet-independent *parametric 0-1 SP alignment under* local *similarity measures is:*

(a) $O(n^{5/3}p^{5/3})$ *if the gap penalty is zero and*
(b) $O(n^{8/3}p^{8/3})$ *if the gap penalty varies.*

Proof. The total similarity score of a multiple alignment $\mathcal{A}$ is given by

$$\boldsymbol{\sigma}_{\mathcal{A}} = \sum_{1\le i<j\le k} b_{ij}w_{ij} - \alpha \sum_{1\le i<j\le k} b_{ij}x_{ij} - \beta \sum_{1\le i<j\le k} b_{ij}y_{ij} - \gamma \sum_{1\le i<j\le k} b_{ij}z_{ij}. \tag{10}$$

Now, parts (a) and (b) follow from applying Lemma 8 for $d = 2$, and $d = 3$ and $N = np$. □

Theorem 4. *The number of optimality regions for* alphabet-dependent *parametric global or local 0-1 SP alignment under distance and similarity measures is* $(np)^{O(|\Sigma|^2)}$.

Proof. By equation (5), the value of an alignment $\mathcal{A}$ is:

$$\boldsymbol{\delta}_{\mathcal{A}} = \gamma \sum_{1\le i<j\le k} b_{ij}z_{ij} + \sum_{\{s,t\}\subseteq\Sigma} \alpha(s,t) \sum_{1\le i<j\le k} b_{ij}x_{ij}(s,t) \tag{11}$$

where z_{ij} and $x_{ij}(s,t)$ are, respectively, the number of gaps and the number of times character s is lined up with character t in the pairwise alignment between strings i and j induced by $\mathcal{A}$. The claim follows from Lemma 9, since $\sum_{1\le i<j\le k} b_{ij}x_{ij}(s,t)$ can take on $O(np)$ distinct values. □

We now consider a two-parameter problem whose pairwise version was studied earlier by Gusfield et al. [20] in an attempt to analyze the trade-offs between match, mismatch, and indel penalties under alphabet-dependent scoring, when the substitution matrix is fixed. Given a multiple alignment $\mathcal{A}$, define the score of the induced pairwise alignment for sequences i and j as

$$M_{ij}(\mathcal{A}) = \sum_{t\in\Sigma\setminus\{\text{-}\}} \alpha(t,t)x_{ij}(t,t), \qquad MS_{ij}(\mathcal{A}) = \sum_{s,t\in\Sigma\setminus\{\text{-}\},s\neq t} \alpha(s,t)x_{ij}(s,t),$$

$$S_{ij}(\mathcal{A}) = \sum_{t\in\Sigma\setminus\{\text{-}\}} \alpha(t,\text{-})x_{ij}(t,\text{-}).$$

The total score of $\mathcal{A}$ is the sum of the pairwise scores:

$$\boldsymbol{\sigma}_{\mathcal{A}}(\lambda,\mu) = \sum_{i<j} M_{ij} - \lambda \sum_{i<j} MS_{ij} - \mu \sum_{ij} S_{ij}. \tag{12}$$

We refer to the problem of finding a maximum-score alignment under the above scoring scheme as the *SP trade-off problem*. Gusfield et al. [20] proved a subexponential bound on the number of optimality regions encountered in traversing the (λ,μ)-plane along any line. For the case where the entries of the substitution matrix are small integers, as is often true for PAM and BLOSUM matrices used in practice, we can prove a better bound.

Theorem 5. *Suppose that for $s, t \in \Sigma$, $\alpha(s,t) \in \mathbb{Z}$ and $|\alpha(s,t)| \leq U$, $U \in \mathbb{Z}$. Then, the total number of optimality regions induced by the SP trade-off problem on the (λ, μ) plane is $O(n^{5/3}U^{5/3}k^{10/3})$.*

Proof. Follows from Lemma 8 since $\sum_{i<j} M_{ij}$, $\sum_{i<j} MS_{ij}$, and $\sum_{i<j} S_{ij}$ are $O(nUk^2)$. □

Finally, consider the situation where the weights are arbitrary real values.

Theorem 6. *The number of optimality regions for global or local parametric SP alignment is $n^{O(k^2)}$ for the alphabet-independent case and $n^{O(k^2|\Sigma|^2)}$ for the alphabet-dependent case under similarity and distance measures.*

Proof. We consider only the alphabet-dependent case; the other cases are similar. The cost of a feasible solution is given by (11). The claim now follows from Lemma 9, since $\sum_{i<j} b_{ij}x_{ij}(s,t)$ can take on $n^{O(k^2)}$ distinct values and there are $O(|\Sigma|^2)$ parameters. □

4.2 Phylogenetic Alignments

Abusing terminology, we shall call a feasible solution to the phylogenetic alignment problem a *phylogenetic alignment* (or, simply, an alignment). An alignment will be viewed as consisting of both an internal labeling for the input phylogeny T and, for each edge T a pairwise alignment between the sequences labeling its endpoints. For the generalized case, in addition to the above, a feasible solution (also called an *alignment*) will also consist of a phylogeny.

Theorem 7. *The number of optimality regions for parametric phylogenetic and generalized phylogenetic alignment under* alphabet-independent *scoring is*

(a) $O(n^{2/3}k^{4/3})$ under the distance measure if the gap penalty is zero,
(b) $O(n^{5/3}k^{10/3})$ under the distance measure if the gap penalty is allowed to vary.
(c) $O(n^{5/3}k^{10/3})$ under the similarity measure if the gap penalty is held at zero, and
(d) $O(n^{8/3}k^{16/3})$ under the similarity measure if the gap penalty is allowed to vary.

Proof. By Lemma 5, and the fact that the number of internal nodes of any phylogeny for $\mathcal{S}$ is at most k, $0 \leq w_{\mathcal{A}}, x_{\mathcal{A}}, y_{\mathcal{A}}, z_{\mathcal{A}} \leq N = O(nk^2)$. Under distance measures, the total score of an alignment is $x_{\mathcal{A}} + \beta y_{\mathcal{A}} + \gamma z_{\mathcal{A}}$. Now, (a) and (b) follow from Lemma 8 with $d = 1$ and $d = 2$, respectively. Under similarity measures, the score is $w_{\mathcal{A}} - \alpha x_{\mathcal{A}} - \beta y_{\mathcal{A}} - \gamma z_{\mathcal{A}}$. Thus, parts (c) and (d) follow from Lemma 8 with $d = 2$ and $d = 3$, respectively. □

Theorem 8. *The number of optimality regions for star alignment under global alphabet independent scoring is $O(n^{2/3}k^{2/3})$ when the gap penalty is fixed and $O(n^{5/3}k^{5/3})$ when the gap penalty varies.*

Proof. Use Lemmas 6 and 8 with $d = 2$ and $d = 3$, respectively. □

Theorem 9. *The number of optimality regions for optimality regions for alphabet-dependent parametric phylogenetic and generalized phylogenetic alignment under distance and similarity measures is* $(nk^2)^{O(|\Sigma|^2)}$.

Proof. By equation (5), the value of an alignment $\mathcal{A}$ is:

$$val_{\mathcal{A}} = \gamma \sum_{(u,v)\in T} z_{uv} + \sum_{\{s,t\}\subseteq\Sigma} \alpha(s,t) \sum_{(u,v)\in T} x_{uv}(s,t) \tag{13}$$

where T is the phylogeny, z_{uv} and $x_{uv}(s,t)$ are, respectively, the number of gaps and the number of times character s is lined up with character t in the pairwise alignment between the strings labeling nodes u and v of T. The value of $\mathcal{A}$ is thus a function of $O(|\Sigma|^2)$ parameters. Moreover, $\sum x_{uv}(s,t)$ can take on $O(nk^2)$ distinct values. The claim now follows from Lemma 9. □

5 Discussion

Since the number of optimality regions for all problems considered here is polynomial in the length and number of sequences (assuming bounded alphabet in the alphabet-dependent case). Theorem 1 implies that the corresponding minimization diagrams can be computed with a polynomial number of calls to algorithms for the respective fixed-parameter problems. However, the fact that an exact solution to these problems is needed is a big limitation, since the cost of carrying out even a single multiple or phylogenetic alignment is prohibitive, except for short sequences.

In practice, the fixed-parameter problems are often solved heuristically, and each such scheme raises its own parameter-sensitivity issues. The approach presented here can be used to analyze any procedure that minimizes or maximizes a function that has a discrete dependence on the features of pairwise alignments. Examples of such approaches are given in [18]. Different techniques seem necessary to analyze heuristics that do not fall in this category; e.g., progressive alignment [12] and some of the methods outlined in [33].

Acknowledgments

Thanks to Gavin Naylor for discussions on sensitivity analysis in phylogeny construction and for pointing out [36]. Thanks also to Steven Altschul for clarifying the meaning of gaps in multiple alignments and to Dan Gusfield for hospitality and useful discussions while the first author visited Davis.

References

1. Pankaj K. Agarwal and Micha Sharir. *Davenport-Schinzel Sequences and their Geometric Applications*. Cambridge University Press, Cambridge–New York–Melbourne, 1995.
2. Stephen F. Altschul. Gap costs for multiple sequence alignment. *J. Theor. Biol.*, 138:297–309, 1989.
3. Stephen F. Altschul. Leaf pairs and tree dissections. *SIAM J. Disc. Math.*, 2(3):293–299, 1989.
4. Stephen F. Altschul, Raymond J. Carrol, and David J. Lipman. Weights for data related by a tree. *J. Mol. Biol.*, 207:647–653, 1989.
5. Stephen F. Altschul and David J. Lipman. Trees, stars, and multiple biological sequence alignment. *SIAM J. Appl. Math.*, 49(1):197–209, 1989.
6. M.O. Dayhoff, R.M. Schwartz, and B.C. Orcutt. A model of evolutionary change in proteins. *Atlas of Protein Sequence and Structure*, 5:345–352, 1978.
7. David Dobkin, Herbert Edelsbrunner, and C. K. Yap. Probing convex polytopes. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 424–432, 1986.
8. David Dobkin, Herbert Edelsbrunner, and C. K. Yap. Probing convex polytopes. In Cox and Wilfong, editors, *Autonomous Robot Vehicles*, pages 328–341. Springer-Verlag, 1990.
9. Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, Heidelberg, 1987.
10. M.J. Eisner and D.G. Severance. Mathematical techniques for efficient record segmentation in large shared databases. *J. Assoc. Comput. Mach.*, 23:619 –635, 1976.
11. J. Felsenstein. Phylogeny programs. `http://evolution.genetics.washington.edu/phylip/software.html`.
12. D. Feng and R.F. Doolittle. Progressive alignment as a prerequisite to correct phylogenetic trees. *J. Mol. Evol.*, 25:351–360, 1987.
13. David Fernández-Baca, Timo Seppäläinen, and Giora Slutzki. Bounds for parametric sequence comparison. In *Proc. Symp. on String Processing and Information Retrieval*, pages 55–62. IEEE Computer Society, 1999.
14. David Fernández-Baca and S. Srinivasan. Constructing the minimization diagram of a two-parameter problem. *Operations Research Letters*, 10:87–93, 1991.
15. S. K. Gupta, J. Kececioglu, and A. A. Schäffer. Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment. *Journal of Computational Biology*, 2:459–472, 1995.
16. Dan Gusfield. Sensitivity analysis for combinatorial optimization. Technical Report UCB/ERL M80/22, University of California, Berkeley, May 1980.
17. Dan Gusfield. Parametric combinatorial computing and a problem in program module allocation. *J. Assoc. Comput. Mach.*, 30(3):551– 563, 1983.
18. Dan Gusfield. Efficient algorithms for multiple sequence alignment with guaranteed error bounds. *Bull. Math. Biol.*, 55:141–154, 1993.
19. Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge–New York–Melbourne, 1997.
20. Dan Gusfield, K. Balasubramanian, and Dalit Naor. Parametric optimization of sequence alignment. *Algorithmica*, 12:312–326, 1994.

21. Dan Gusfield and Robert W. Irving. Parametric stable marriage and minimum cuts. *Information Processing Letters*, 30:255–259, 1989.
22. Dan Gusfield and Paul Stelling. Parametric and inverse-parametric sequence alignment with XPARAL. In Russell F. Doolittle, editor, *Computer methods for macromolecular sequence analysis*, volume 266 of *Methods in Enzymology*, pages 481–494. Academic Press, 1996.
23. S. Henikoff and J.G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc. Natl. Acad. Sci. USA*, 89:10915–10919, 1992.
24. Xiaoqiu Huang, Pavel A. Pevzner, and Webb Miller. Parametric recomputing in alignment graphs. In M. Crochemore and D. Gusfield, editors, *Combinatorial Pattern Matching*, volume 807 of *Lecture Notes in Computer Science*, pages 87–101. Springer-Verlag, 1994.
25. T. Jiang and L. Wang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1:337–348, 1994.
26. Tao Jiang, Eugene L. Lawler, and Lusheng Wang. Aligning sequences via an evolutionary tree: Complexity and approximation (extended abstract). In *Proc. 26th Ann. Symp. Theory of Computing*, pages 760–769, 1994.
27. D. J. Lipman, S. F. Altschul, and J. D. Kececioglu. A tool for multiple sequence alignment. *Proc. Natl. Acad. Sci. USA*, 86:4412–4415, 1989.
28. Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.
29. D. Sankoff and R. J. Cedergren. Simultaneous comparison of three or more sequences related by a tree. In Sankoff and Kruskal [31], pages 253–263.
30. David Sankoff. Minimal mutation trees of sequences. *SIAM J. Appl. Math.*, 28(1):35–42, January 1975.
31. David Sankoff and Joseph B. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, Reading, MA, 1983.
32. Tetsuo Shibuya and Hiroshi Imai. New flexible approaches for multiple sequence alignment. *Journal of Computational Biology*, 4(3):385–413, 1997.
33. Martin Vingron. Sequence alignment and phylogeny construction. In M. Farach-Colton et al., editor, *Mathematical Support for Molecular Biology*, volume 47 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1998.
34. Martin Vingron and Michael S. Waterman. Sequence alignment and penalty choice: Review of concepts, case studies, and implications. *J. Mol. Biol.*, 235:1–12, 1994.
35. Michael S. Waterman, M. Eggert, and Eric Lander. Parametric sequence comparisons. *Proc. Natl. Acad. Sci. USA*, 89:6090–6093, 1992.
36. Ward C. Wheeler. Sequence alignment, parameter sensitivity, and the phylogenetic analysis of molecular data. *Syst. Bio.*, 44(3):321–331, 1995.

Tsukuba BB: A Branch and Bound Algorithm for Local Multiple Sequence Alignment

Paul Horton

Real World Computing Partnership
Tsukuba Mitsui Bldg., Ibaraki 305-0032, Japan
horton@rwcp.or.jp

Abstract. In this paper we present a branch and bound algorithm for local gapless multiple sequence alignment (motif alignment) and its implementation. This is the first program to exploit the fact that the motif alignment problem is easier for short motifs. Indeed for a fixed motif width the running time of the algorithm is asymptotically linear in the size of the input. We tested the performance of the program on a dataset of 300 E.*coli* promoter sequences. For a motif width of 4 the optimal alignment of the entire set of sequences can be found. For the more natural motif width of 6 the program can align 19 sequences of length 100; more than twice the number of sequences which can be aligned by the best previous exact algorithm. The algorithm can relax the constraint of requiring each sequence to be aligned, and align 100 of the 300 promoter sequences with a motif width of 6. We also compare the effectiveness of the Gibbs sampling and beam search heuristics on this problem and show that in some cases our branch and bound algorithm can find the optimal solution, with proof of optimality, when those heuristics fail to find the optimal solution.

1 Introduction

The function of DNA and protein sequences can often be characterized by the presence of important substrings or "motifs", in the case of DNA sequences often corresponding to protein binding sites. This observation has led to the field of local gapless multiple sequence alignment (hereafter motif alignment) which attempts to find meaningful substrings from a collection of sequences which have a common biological function, typically by learning a function which scores substrings based on how "good" they are as instances of the motif in question. Consensus based methods score commonly appearing substrings higher than other substrings, sometimes allowing a small number of mismatches. These methods have the advantage that it is easy to devise exact algorithms for small motif lengths. [14] is a recent example of this approach. However for poorly conserved motifs the number of common substrings grows exponentially with the motif length, and since the available data is always limited it is problematic to accurately estimate the score of each individual substring. Thus it is common to use matrix based or "profile" methods which decompose the scoring function into a sum over each position in the substring to be scored. Unfortunately for

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 84–98, 2000.

these matrix based methods there have been no known algorithms which do not depend exponentially on the number of sequences n. Indeed the specific problem formulation adopted in this work has been found to be NP-hard [1], [10] for a general motif width. Furthermore [2] show that the problem is APX-hard, which implies that, if $P \neq NP$, a polynomial time approximation scheme for this problem does not exist. The problem is also NP-hard when the constraint of finding an optimal discrete alignment is relaxed to that of finding an optimal "fuzzy" alignment [7]). Thus most work in this field has concentrated on finding good heuristic algorithms. Beam search [12], and Gibbs Sampling [8] have been used for the specific problem formulation that this paper adopts, and expectation maximization (EM)[9] has been used for a related problem. In earlier work, an exact branch and bound algorithm has been developed which can solve some problem instances with more that 10 DNA sequences [6]. However, the worst case running time of the algorithm is still exponential in n and can be very long in practice.

In this paper we describe a new branch and bound algorithm which is the first matrix based exact method to actively exploit the fact that the alignment problem is easier for smaller motif widths. Indeed for a fixed width motif the running time of the algorithm has an upper bound, which although quite large, is only linear in the number of sequences to be aligned. We refer to our new algorithm as the "Tsukuba BB Algorithm" and the older branch and bound algorithm [6] as the "Berkeley BB Algorithm".

This paper is organized as follows: we define a specific optimization problem, then describe a search tree for finding optimal alignments and introduce a score based bound for nodes in that search tree. Following that, we introduce a technique for avoiding redundant search of equivalent paths of the search tree and prove its correctness. We then define the concept of a *consistent* alignment and show how that concept leads to a complementary method for pruning branches in the search tree. After that we describe our implementation of the algorithm and give empirical results comparing its performance to a previous branch and bound algorithm, Gibbs sampling, and beam search, on a dataset of E.*coli* promoter sequences. Finally, a generalization of the original optimization problem in which some sequences are left out of the alignment is introduced and its use is demonstrated on the promoter dataset. We close with a short discussion of the results presented.

2 Problem Definition

We adopt the following problem formulation: to choose one substring of length w from each sequence in a set of sequences such that the score of the chosen substrings is maximal. For a scoring function we use the common maximum likelihood ratio score.

2.1 Maximum Likelihood Ratio Score

In this section we formally define the maximum likelihood ratio (MLR) score of an alignment. For our purposes an alignment A is a collection of length w substrings from the input sequences, where A_j is the substring taken from the jth input sequence. First we define a special case known as the entropy score. In words it is simply -1 times the sum of the information theoretic entropy of the distribution of characters in each column in the given alignment. Let σ denote the size of the alphabet, n be the number of input sequences, and $F(c,i)$ the frequency of character c in column i of the alignment. (The matrix F/n is sometimes called a profile.) We slightly abuse notation by letting σ represent the set of characters in the alphabet in the phrase $c \in \sigma$. The entropy score can then be written as:

$$\sum_{i=1}^{w} \sum_{c \in \sigma} \frac{F(c,i)}{n} \log \frac{F(c,i)}{n}.$$

The MLR Score adds a probability vector $\boldsymbol{B}$ of length σ as a background model. Also it is common to add a vector of "pseudocounts" $P(1 \ldots \sigma)$. The MLR Score, R(A) of an alignment A is defined as:

$$\sum_{i=1}^{w} \sum_{c \in \sigma} \frac{F'(c,i)}{n'} (\log \frac{F'(c,i)}{n'} - \log B(c)),$$

$$\text{where } n' = n + \sum_{c \in \sigma} P(c),\ F'(c,i) = F(c,i) + P(c)$$

The MLR score is equivalent to the entropy score when a uniform distribution is used for the background model and all pseudocounts are set to zero, as discussed in [13] and [3].

The standard definition of the MLR score, given above, sums over columns in an alignment and characters in the alphabet. However it will suit our purposes to rewrite this score as an equivalent formula that sums over substrings included in the alignment. To describe that score we use A_{ji} to denote the ith character in the substring A_j, i.e. $A_j = A_{j1} \ldots A_{jw}$. We define the *score* of a substring $S = s_1 \ldots s_w$ relative to a frequency matrix F' as:

$$t_{F'}(S) = \frac{1}{n'} \sum_{i=1}^{w} (\log \frac{F'(s_i,i)}{n'} - \log B(s_i))$$

The MLR score can now be written as:

$$R(A) = \sum_{j=1}^{n} t_{F'}(A_j) + \frac{w}{n'} \sum_{c \in \sigma} P(c) (\log \frac{F'(c,i)}{n'} - \log B(c))$$

Note that the second term does not include an A_j term. Therefore we can define a *prior included score* of a substring S as:

$$t'_{F'}(S) = t_{F'}(S) + \frac{w}{nn'} \sum_{c \in \sigma} P(c) (\log \frac{F'(c,i)}{n'} - \log B(c))$$

and then rewrite that MLR score as the sum of the score of each substring in an alignment.

$$R(A) = \sum_{j=1}^{n} t'_{F'}(A_j).$$

Choice of Background Probabilities and Priors. The choice of background model will depend on the intended use. In this paper we adopt two common choices: a uniform background model and a model based on the frequencies of bases in the input. For priors, we used the add one Laplace prior i.e. we set all the elements of $\boldsymbol{P}$ to one.

3 Condensed Search Tree

The search tree used by the Tsukuba BB algorithm is constructed from a root node and input set of sequences as follows: for any substring s of length w found in the input sequences, construct a branch from the root to a new node labeled s, then recursively construct a search tree from s using s as the root node and the same set of input sequences except for the removal of *all* sequences containing s. An example of a search tree is shown in figure 1. Note that this search tree is *condensed* in the sense that a single edge can represent multiple occurrences of a substring. Thus some alignments cannot be expressed as paths through this tree. However we prove that a path to an optimal alignment exists in this tree.

Theorem 1 *A path to any optimal alignment exists in the condensed search tree.*

We use some properties of an optimal alignment A^* to prove this theorem. We denote the MLR score obtained by this alignment as R^* and the prior included frequency matrix of the alignment by $F^{*\prime}$. We define the *optimal model score* $M(A)$ of an alignment as the prior included score of the substring relative to $F^{*\prime}$ summed over the substrings in the alignment. i.e.

$$M(A) = \sum_{j=1}^{n} t'_{F^{*\prime}}(A_j)$$

(One interpretation of this is that it is the log likelihood ratio of A being generated by the model $F^{*\prime}$ versus the background model $\boldsymbol{B}$.) We use this definition to prove two lemmas.

Lemma 1. *There exists a path in the condensed tree which corresponds to an alignment whose optimal model score is at least* R^*.

Proof: The proof is by construction. Consider the distinct length w substrings contained in the input sequences. Order all the substrings by their prior included score relative to $F^{*\prime}$. Align all occurrences of the highest scoring substring and remove the aligned sequences from consideration. Repeat this procedure with the remaining sequences until a full alignment is obtained. This procedure gives

the alignment with the maximum optimal model score, which must be at least R^* since A^{opt} is a feasible alignment whose optimal model score is R^*.

Lemma 2. *The optimal model score of an alignment is never greater than its MLR score*

Proof: The optimal model score of an alignment A with a prior included frequency matrix F' can be written as:

$$M(A) = \sum_{j=1}^{n} t'_{F^{*\prime}}(A_j) = \sum_{i=1}^{w} \sum_{c \in \sigma} \frac{F'(c,i)}{n'} (\log \frac{F^{*\prime}(c,i)}{n'} - \log B(c)). \tag{1}$$

The MLR score of A is:

$$R(A) = \sum_{i=1}^{w} \sum_{c \in \sigma} \frac{F'(c,i)}{n'} (\log \frac{F'(c,i)}{n'} - \log B(c)). \tag{2}$$

Here we state a well known fact, which can be proved with the method of Lagrange multipliers. For any given fixed probability vector $\boldsymbol{u}$ of length σ, the maximum of the sum $u_i \log v_i$ maximized over any length σ probability vector $\boldsymbol{v}$ occurs when $\boldsymbol{v} = \boldsymbol{u}$. i.e.

$$\max_{\boldsymbol{v}} \sum_{i=1}^{\sigma} u_i \log v_i = \sum_{i=1}^{\sigma} u_i \log u_i \tag{3}$$

Combining equations 1, 2, 3 gives:

$$M(A) \leq R(A) \leq R^*$$

Thus any alignment which has an optimal model score of at least R^* must in fact have an optimal model score of exactly R^* and be an optimal alignment. The alignment guaranteed to exist by lemma 1. is indeed an optimal alignment, which proves the theorem. By condensing nodes, the size of the search tree becomes bounded by a function of the alphabet and the motif width. As a substring is never repeated in a path in the search tree, each leaf of the search tree represents an ordering of some of the length w substrings. Since there are only σ^w possible substrings of length w, the size of the tree is $O((\sigma^w)!)$. Although this is a large number, note that it is *independent* of the number of sequences n. Thus in the limit as w is fixed and n goes to infinity, any algorithm which efficiently searches the condensed search tree, e.g. depth first search, requires only constant time plus the time needed to read the input, which is linear in n. Of course the size of the search tree is also bounded by some functions of n. Namely it is $O(d^n)$, where d is the number of distinct substrings of length w occurring in the input sequences; d itself is bounded by the number of bases in the input as well as by σ^w.

4 Tsukuba BB Algorithm

The Tsukuba BB algorithm is simply depth first search on the condensed search tree with pruning. The pruning criteria guarantees that a path through the tree

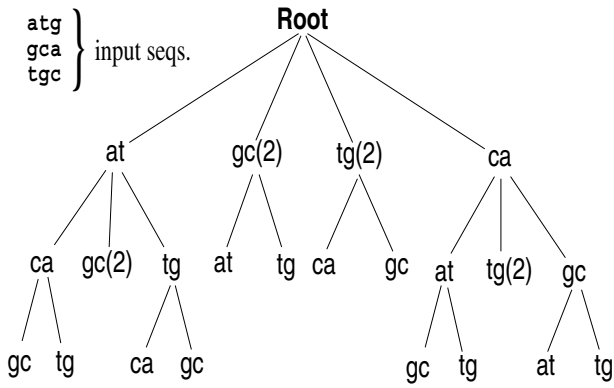


Fig. 1. A diagram of the search tree for three sequences with a motif width of two is shown. Multiple occurrences of a substring represented by a node are indicated by the number in parenthesis, as in "tg(2)".

which builds an optimal alignment by adding substrings in the best first scoring order, e.g. with non-increasing values of $t'_{F^{*\prime}}$, will never be pruned. Note that the construction of Lemma 1. in the previous section guarantees that such a path to an optimal alignment exists.

4.1 Pruning Criteria

Let L be a lower bound on R^*, for example the MLR score of some feasible solution. Let a *subalignment* be defined as an alignment of some of the input sequences. Let S_q be the set of substrings contained in a subalignment $A^{(q,m)}$ of $m < n$ sequences at some node q of the search tree. Let an *extension* from q be any alignment of n substrings $A^{(q,n)}$ produced by adding substrings from S_q to $A^{(q,m)}$. Note that while $A^{(q,m)}$ is a feasible subalignment of the input sequences, an extension $A^{(q,n)}$ is not a feasible alignment of the input sequences. Thus extensions are not of direct interest. However the following pruning criteria uses extensions to advantage.

Pruning Criteria: Prune a node q if any extension $A^{(q,n)}$ can be found such that:

$$R(A^{(q,n)}) < L$$

4.2 Correctness of the Pruning Criteria

We must show that no node which represents a subalignment of an optimal alignment and built in best scoring first order will ever be pruned.

Theorem 2 *Let A^* be an optimal alignment having a prior included frequency matrix $F^{*\prime}$. Let q be a node in the search tree representing a subalignment of A^* built by adding substrings in order of non-decreasing $t'_{F^{*\prime}}$ values. For any extension $A^{(q,n)}$ of q*

$$R(A^{(q,n)}) \geq R(A^*).$$

Proof: From lemma 2. of the previous section we know that the MLR score of $A^{(q,n)}$, $R(A^{(q,n)})$, is at least as great as its optimal model score, $M(A^{(q,n)})$. Thus it is sufficient to show that $M(A^{(q,n)}) \geq R(A^*)$. Note that:

$$M(A^{(q,n)}) = \sum_{j=1}^{m} t'_{F^{*\prime}}(A_j^{(q,n)}) + \sum_{j=m+1}^{n} t'_{F^{*\prime}}(A_j^{(q,n)})$$

Where we have divided the sum into two sums by numbering the sequences so that the first m sequences are the ones included in the subalignment represented by q.
Likewise $R(A^*)$ can be expanded as:

$$R(A^*) = \sum_{j=1}^{m} t'_{F^{*\prime}}(A_j^*) + \sum_{j=m+1}^{n} t'_{F^{*\prime}}(A_j^*)$$

The first summation term for $M(A^{(q,n)})$ and $R(A^*)$ is the same by the requirement that the subalignment represented by q is a subalignment of A^*. Thus the difference is:

$$M(A^{(q,n)}) - R(A^*) = \sum_{j=m+1}^{n} \{t'_{F^{*\prime}}(A_j^{(q,n)}) - t'_{F^{*\prime}}(A_j^*)\}$$

Since the substrings added at q are taken from the substrings in A^* in order of non-decreasing $t'_{F^{*\prime}}$ values, each term of this summation is at least zero. Thus:

$$R(A^{(q,n)}) \geq M(A^{(q,n)}) \geq R(A^*)$$

When priors are not used a more powerful pruning criterion can be used:
Pruning Criteria (No Priors): Prune a node q, representing a subalignment with m total substrings, if any extension $A^{(q,i)}$, $m \leq i \leq n$ can be found such that:

$$R(A^{(q,i)}) < L$$

We omit the proof of correctness, which is similar to the proof of the pruning criteria used with priors given in the previous section.
Choosing Extensions. As stated above, the score of *any* extension, $R(A^{(q,n)})$, is an upper bound on the score of alignments which may be obtained by expanding node q. We use a greedy strategy to try to find an extension which gives a low upper bound. We consider all extensions obtained by adding one substring from S_q and pick the lowest scoring one. i.e. we pick an extension that minimizes $R(A^{(q,m+1)})$. We then consider all possible ways to add one substring to that extension to produce an extension with $m+2$ total substrings, and so on.

5 Using Canonical Representative Nodes to Reduce Redundant Work

General Idea. The condensed search tree shown in figure 1 includes some redundancy. This redundancy comes from adding the same set of distinct substrings

in different orders. For example node ("at", "ca") is *equivalent* (represents the same subalignment) to node ("ca", "at"). Note that this is not always the case for two nodes whose subalignments contain the same distinct substrings. For example node ("gc", "tg") is not equivalent to node ("tg", "gc"). This is because there is a sequence in the input which contains both "gc" and "tg" and therefore the alignment of that sequence differs for the two nodes. Still, in the worst case many groups of up to $n!$ equivalent nodes may be found in the tree. We have developed an algorithm which effectively reduces redundant calculations without significantly increasing memory requirements.

Even if a node survives the pruning criteria of the previous section it would be correct for us to prune that node if we could guarantee that we would *not* prune some other equivalent node. The general idea of our algorithm is to try to expand only a single canonical representative node from any group of equivalent nodes. In the following section we describe the algorithm we implemented which largely achieves this goal.

Generation Canonical Nodes. We assume that an ordering has been assigned to the set of possible substrings of length w. Before describing the algorithm we introduce some additional notation: let I be the set of input sequences and $I - \{U_1, \ldots, U_p\}$ denote the input sequences that do not contain any substrings from the set $\{U_1, \ldots, U_p\}$. The algorithm for computing canonical nodes, shown in figure 2, takes as input a node U and either outputs U itself or an equivalent node with a longer tail of ascending substrings. Where

Definition 1 *The tail of a node $U = U_1 \cdots U_k$ is a series $U_p \cdots U_k$, where p is the smallest integer such that:*

$$\forall i \; p \leq i < k, \;\; U_i < U_{i+1}$$

The purpose of this section is to show that:

Theorem 3 *The algorithm shown in Figure 2 either outputs U or a node which is equivalent to U but has a longer tail than U.*

Proof: The program exits from one of four return statements. The first two cannot violate the terms of the theorem because they return U unchanged. The third returns V where,

$$U = U_1 \cdots U_r \cdots U_i U_{i+1} \cdots U_k U_{k+1}$$
$$V = U_1 \cdots U_r \cdots U_i U_{k+1} U_{i+1} \cdots U_k$$

Note that the only difference between U and V is the placement of U_{k+1}. The **For** loop condition ensures that no substring from $\{U_{i+1}, \ldots, U_k\}$ occurs together with U_{k+1} in any sequence in $I - U_1, \ldots U_r$. Thus U and V are equivalent. The fourth returns V where,

$$U = U_1 \cdots U_r U_{r+1} \cdots U_k U_{k+1}$$
$$V = U_1 \cdots U_r U_{k+1} U_{r+1} \cdots U_k$$

In this case also, U and V are equivalent for the same reason. It remains to prove that V returned by the third or fourth return statement has a longer tail than U. If the algorithm executes past the first **If** statement the tail of U must be just U_{k+1}. However if the algorithm returns from the third or fourth return statements it returns a node V whose tail includes U_k and U_{k-1} (and possibly other nodes as well). This proves the theorem. We close this section

Canonical(U) // Output the canonical representative node of $U = U_1 \cdots U_{k+1}$

If $U_{k+1} > U_k$ Return(U) // #1.

If $U_{k-1} > U_k$ Return(U) // #2.

Assign π such that U_π is the last substring in U, other than U_k, which is greater than its successor. // The series $U_{\pi+1} \cdots U_k$ is in ascending order.

If $U_{k+1} < U_{\pi+1}$
    $r \leftarrow \pi$
Else
    Assign r such that:
    U_r is the last substring in $U_{\pi+1}, \ldots, U_k$ which is smaller than U_{k+1}

For($i = k$; $i > r$; $i = i - 1$) // Note that i is *decremented*.
    If any sequence in $I - \{U_1, \ldots, U_r\}$ contains both U_i and U_{k+1}
        Let V be the same as U except with U_{k+1} moved directly after U_i
        Return(V) // #3. Note that if $i = k$ here, $V = U$.

Let V be the same as U except with U_{k+1} moved directly after U_r
Return(V) // #4. The series $V_{\pi+1} \cdots V_{k+1}$ is in ascending order.

Fig. 2. Pseudocode for computing the canonical representative of a node

by noting that in practice the second return statement, returning U, is called infrequently. This is because, as will be seen in the next section, the parent node of U, $U_1 \cdots U_k$, itself is a node returned from **Canonical** and therefore U_{k-1} tends to be less than U_k.

The Tsukuba BB Algorithm Using Canonical Nodes. Here we define the *canonical* node V of a given node U as the node returned from the algorithm shown in figure 2. Note that theorem 3 ensures that either $V = U$ or the tail of V is longer than the tail of U. Thus at least one node (any one with a maximal length tail) in a group of equivalent nodes must be its own canonical node.

The steps taken to decide whether to expand a node U, whose canonical node is V, during the depth first search of the Tsukuba BB algorithm are:

- 1. If U can be pruned by the score based pruning criteria don't expand
- 2. Otherwise, if V is the same as U, expand.
- 3. Otherwise, if all of the ancestors of V in the search tree survive the score based criteria, don't expand.
- 4. Otherwise, Expand.

Note that for step 3. common ancestors of both U and V do not need to be checked (we know the ancestors of U pass the pruning criteria). This algorithm guarantees that if any node U is reached that passes the score based pruning criteria, either U will be expanded, or an equivalent node with a longer tail will be expanded.

6 Pruning Inconsistent Alignments

```
AT gac
AC atg
```

Fig. 3. The aligned substrings shown in upper case.

In this section we describe a pruning strategy which complements the score based bound. The key idea is that *inconsistent* subalignments can be pruned. An example of a inconsistent alignment is shown in figure 3. This alignment is inconsistent in the sense that it favors 'at' over 'ac' in the first sequence, but the opposite is true in the second sequence. We believe optimal alignments for reasonable scoring functions will not behave in this way. In particular for the MLR scoring function used in this work, optimal alignments can be produced by independently optimizing the score of the substring picked in each sequence relative to the frequency matrix $F^{*\prime}$. Of course we do not know $F^{*\prime}$ before we solve the problem. However, if we assume that a subalignment is included in an optimal alignment, that subalignment gives us some information about $F^{*\prime}$, which in turn restricts the number of ways in which sequences can be added to that subalignment while maintaining consistency. For example if a subalignment aligns 'at' in a sequence containing 'ac' we can conclude that $F^{*\prime}(t,2) \geq F^{*\prime}(c,2)$, and therefore, for any unaligned sequence which contains a pair ('xt','xc'), where 'x' is any base, the score of 'xc' cannot exceed the score of 'xt'. We describe this relationship between 'at' and 'ac' as 'at' *dominates* 'ac'.

We exploit this by keeping a $\sigma^w \times \sigma^w$ binary matrix **D_table**, which allows pruning based on inconsistency. This type of pruning is easily added to the Tsukuba BB algorithm described so far. First, all of the entries of **D_table** are initialized to *false*. Then the condensed search tree is descended using **D_table** to prune any inconsistent subalignments which may have survived the score based bound. When a subalignment cannot be pruned, relationships of the form ('xt','xc') for the newly aligned substring paired with all other substrings occurring in the newly aligned sequences are added and the transitive closure is taken.

7 Berkeley BB

For comparison purposes we report the running times of aligning sequences with a previous branch and bound algorithm [6], which we refer to as the Berkeley BB algorithm. That algorithm uses a non-condensed search tree, which is dependent on the order of the input sequences. The bound used by the algorithm is different than the score based pruning criteria described here. The worst case time of the algorithm is $O(l^n)$, but in practice the algorithm runs much faster than a naïve

enumerative algorithm. We slightly modified the implementation to allow priors to be used by adding "dummy sequences" of length w to the input sequences. The modified program can add dummy sequences to the beginning or end of the input, but adding to the beginning was faster for the cases we compared so we report those times.

8 Results

Dataset. We used a dataset of 300 E.*coli* promoter sequences of length 100 [11]. The {a, c, g, t} content of the dataset was 28.2%, 21.2%, 21.2%, and 29.8% respectively. [5] and [11] give analysis and pointers to the extensive biological literature concerning *E.coli* promoters.

Exact Methods. This section reports results from a 450 MHz Pentium II machine running Linux; Tsukuba BB is a C++ program, while Berkeley BB is a C program. For motif widths of up to four we were able to compute an optimal alignment of all 300 sequences using Tsukuba BB. Results for a motif width of four with the Laplace prior are shown in table 1. Results for the Tsukuba BB algorithm for a motif width of five are shown in table 2. In contrast, with a motif width of six and the Laplace prior, Berkeley BB requires 19 hours to align 10 sequences. The running times for the two algorithms for motif widths of six and seven are shown in figure 4. This figure plots times with the Laplace prior.

# seqs	algorithm	uniform	input comp.
7	Berkeley	22.4 *min*	35.3 *min*
7	Tsukuba	0.67 *sec*	1.07 *sec*
300	Tsukuba	44.2 *sec*	28.8 *min*

Table 1. Running times of Tsukuba BB and Berkeley BB are shown for w=4. The background model was either a uniform distribution or the composition of the input sequences.

# seqs	priors	no priors
40	4.45	0.96
45	5.50	1.24
50	30.4	4.65
55	50.0	19.5

Table 2. Running times in hours are shown for w=5, with and without priors. Tsukuba BB was used with a uniform background model.

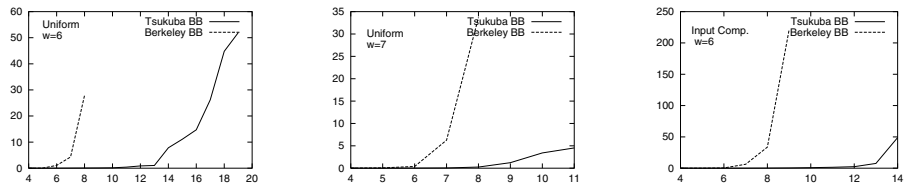


Fig. 4. The running times of the Tsukuba BB and Berkeley BB algorithms are shown for promoter sequences. The x-axis is the number of sequences. The y-axis is the running time in hours. (From left to right) the first two plots show times with a uniform background and motif widths of 6 and 7 respectively. The last shows times for an input composition background model with a motif width of 6.

Heuristics. This section reports the results of running two heuristic algorithms on the input sequences with a motif width of four, a uniform background, and the add-one Laplace prior. The optimal alignment has a score of 6.258, and can be computed in 44 seconds with Tsukuba BB. Two heuristics where used, 1. a beam search algorithm similar to the one used by [12] but with a beam width of 5000, and 2. the Gibbs sampling heuristic [8]. The beam search heuristic was coded in "C" and the Gibbs sampling program "C++". The Gibbs sampling program uses a round robin scheduling pattern for the choice of sequence to realign at each step. It has three modes: No Shifting, Plain Shifting, and Forced Shifting. No shifting is the straight Gibbs sampling algorithm. Shifting is the "phase shifting" described by [8] of up to $w-1$ positions to the left or right. Plain shifting and forced shifting differ only in how they treat boundary conditions. Plain shifting does not consider shifts in which some position(s) cannot be shifted without going past the edge of a sequence, while forced shifting considers those shifts by simply leaving such positions unchanged while shifting the others. Plain shifting becomes less meaningful as the number of sequences grows. This can be seen by considering the proportion of alignments which can be shifted one to the left, which is $(l-w)^n/(l-w+1)^n$. For $l=100$, $w=4$, $n=300$ this proportion only is 4.47%. Table 3 shows the results of tests run with the different heuristics. The beam search is deterministic, but depends on the order of the input sequences, so we used 10 different randomly generated orderings of the input sequences for the beam search trials. Gibbs sampling is inherently stochastic so we simply ran it (for $70,000,000$ potential sequence realignments) in each mode ten times (in each case starting with a randomly chosen alignment). For the two shifting modes, shifting was considered once every 1000 iterations of the main Gibbs sampling loop.

Heuristic	mean score	best score	mean time
Beam	6.083	6.242	57.3
No	5.000	5.077	49.9
Plain	5.014	5.087	50.1
Forced	5.048	5.112	58.7

Table 3. The average score, best score, and average running times in minutes are shown for different heuristics. The averages are over 10 trials with the full promoter data set and a motif width of four. "No", "Plain", and "Forced" refer to the modes of the Gibbs sampling program.

Tsukuba BB for the ZOOPS Problem. [4] proposed a variant of the EM problem formulation in which it is assumed that the motif occurs once in some sequences, and not at all in the other sequences. In this section we describe a generalization of the Tsukuba BB algorithm which uses that general idea and their name, ZOOPS, for Zero or One Occurrences Per Sequence. The corresponding name for requiring one occurrence in each sequence is OOPS.

The modified problem is to find a subalignment of size $k \leq n$, present in the condensed search tree, that has a maximal MLR score, where k is a user given parameter. Two minor changes to the Tsukuba algorithm allow this generalization. First, nodes whose subalignments contain k or more total substrings are not further expanded, and second, the role of n in the pruning criteria is replaced by k.

Pruning Criteria (ZOOPS, No Priors): For the ZOOPS problem, aligning k sequences, Prune a node q, representing a subalignment with m total substrings, if any extension $A^{(q,i)}$, $m \leq i \leq k$ can be found such that:

$$R(A^{(q,i)}) < L$$

Where L is defined as a lower bound for the optimal score of aligning k sequences. When priors are used i must be constrained to be exactly k.

We investigated the use of this generalization on a truncated version of the promoter dataset, in which only the 55 bases from position -50 to position +5 were used. All of the results given in this section were obtained using these shorter sequences. We used Tsukuba BB to align 100 of the 300 truncated promoter sequences with a motif width of six, requiring 54 hours. Figure 5 shows the substrings chosen in the alignment. Figure 6 shows a histogram of the starting positions of the substrings found in the alignment. The occurrences cluster nicely around the -10 position which is consistent with the currently accepted model of promoter structure.

substring	# occurrences
taaaat	27
tagaat	18
tataat	14
taaaaa	14
tacaat	12
tataaa	9
tagaaa	6

Fig. 5. Substrings aligned with the ZOOPS model and a requirement of aligning 100 sequences from the shortened promoter dataset.

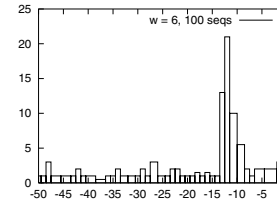


Fig. 6. A histogram of the starting position of the substrings aligned when at least 100 sequences out of 300 were required to be aligned. The substrings were of width six.

9 Discussion

Tsukuba BB is dramatically faster than Berkeley BB for motif widths of up to five. For these motif widths pruning based on inconsistency very effectively complements the score based pruning. For example, the 29 minutes required to align a motif of width four for 300 sequences with an input composition based

background model increases to 61 *hours* when inconsistency based pruning is disabled. Unfortunately this pruning technique is much less effective as the motif width grows (and the number of sequences to be aligned decreases). For example, the 52 hours required to align a motif of width six for 19 sequences with a uniform background only increased to 58 hours when inconsistency based pruning was disabled. On the other hand, canonical based pruning is relatively more effective for longer motif widths, which is expected since the chance that two substrings occur in the same sequence decreases. For example, for a motif width of six and a uniform background model the time required to align 14 sequences increased by a factor of more than 10 when canonical based pruning was disabled.

For a motif width of four Tsukuba BB is able to find the optimal alignment of 300 sequences in 44 seconds, along with a guarantee of its optimality. The two common heuristic algorithms tested here could not find the optimum even *once* after 10 trials of close to one hour each. We do not claim that these are necessarily the best heuristics and parameter settings possible, for example a sequence order independent version of beam search has been developed recently [5]. However we believe these results raise the issue of how well heuristic algorithms will scale up to larger data sets. We note that both heuristics could often find optimal or near optimal alignments in just a few minutes, when the number of sequences was on the order of 20 (results not shown). We do however acknowledge the possibility that the heuristics may perform better with longer motif widths. We did not evaluate EM because it is a heuristic for a different problem formulation. The EM formulation considers the likelihood of the motif model summed over all possible alignments, while the formulation adopted in this work considers only the likelihood of the most likely alignment (i.e. Viterbi path). We note that the significant difference between these two formulations has been widely overlooked in the biological literature.

The results given with ZOOPS were primarily intended to demonstrate how Tsukuba BB can be used for the ZOOPS problem formulation. We note that in general the algorithm can align many more sequences with the ZOOPS than with the basic OOPS problem formulation. Indeed from the results in this paper we see that aligning 100 sequences out of 300 with a motif width of six requires about the same amount of time as aligning 19 sequences with OOPS. [14] analyzes a dataset in which only approximately one third of the sequences are thought to contain the motif (ribosome binding site) of interest; a case in which the ZOOPS problem formulation is appropriate.

Tsukuba BB can currently align most conceivable input datasets for a motif width of four and is not too far from being able to do so for a motif width of five. Since this approach is new it is likely that some further speed-ups will be discovered. Furthermore the algorithm, even when using canonical nodes, is just depth first search with essentially no global state (although the lower bound L is updated infrequently when better feasible solutions are found, using a stale value of L does not effect correctness). Thus the algorithm is easy to parallelize and will benefit fully from hardware advances. This is the best exact method. However

at present the number of sequences which can be handled for a motif width of six or greater is still limited, especially for the OOPS problem formulation.

Conclusion. We have presented the first exact algorithm for a matrix based scoring function which actively exploits the fact that this generally hard problem is easier for a fixed length short motif. In fact the algorithm is asymptotically linear in the number of sequences. In practice the algorithm can align more sequences that the best previous exact method and in some cases can find guaranteed optimal solutions where reasonable heuristics fail.

Acknowledgments. Dr. Yutaka Akiyama for careful reading of this manuscript.

References

1. Tatsuya Akutsu. Hardness results on gapless local multiple sequence alignment. Technical Report 98-MPS-24-2, Information Processing Society of Japan, 1998.
2. Tatsuya Akutsu, Hiroki Arimura, and Shinichi Shimozono. On approximation algorithms for local multiple alignment. *RECOMB2000*, 2000. in press.
3. Timothy L. Bailey. Likelihood vs. information in aligning biopolymer sequences. Technical Report CS93-318, UCSD, February 1993.
4. Timothy L. Bailey and Charles Elkan. The value of prior knowledge in discovering motifs with meme. In *Proceeding of the Third International Conference on Intelligent Systems for Molecular Biology*, pages 21–38. AAAI Press, 1995.
5. G. Z. Hertz and G. D. Stormo. Identifying dna and protein patterns with statistically significant alignments of multiple sequences. *Bioinformatics*, 1999. in press.
6. Paul Horton. A branch and bound algorithm for local multiple alignment. In *Pacific Symposium on Biocomputing '96*, pages 368–383, 1996.
7. Paul Horton. On the complexity of some local multiple sequence alignment problems. Technical Report TR-990001, Real World Computing Partnership, 1999.
8. C. E. Lawrence, S. F. Altschul, M. B. Boguski, J. S. Liu, A. F. Neuwald, and J. C. Wootton. Detecting subtle sequence signals: A gibbs sampling strategy for multiple alignment. *Science*, 262:208–214, 1993.
9. Charles E. Lawrence and Andrew A. Reilly. An expectation maximization (em) algorithm for the identification and characterization of common sites in unaligned biopolymer sequences. *PROTEINS*, 7:41–51, 1990.
10. Ming Li, Bin Ma, and Lusheng Wang. Finding similar regions in many strings. In *STOC*, 1999.
11. Shlomit Lisser and Hanah Margalit. Compilation of *e.coli* mrna promoter sequences. *Nucleic Acids Research*, 21(7):1507–1516, 1993.
12. Gary Stormo and George W. Hartzell. Identifying protein-binding sites from unaligned dna fragments. *Proc. Natl. Acad. Sci., USA*, 86:1183–1187, 1989.
13. Gary D. Stormo. Consensus patterns in dna. *Methods in Enzymology*, 183:211–221, 1990.
14. Martin Tompa. An exact method for finding short motifs in sequences with application to the ribosome binding site problem. In *Proceeding of the Seventh International Conference on Intelligent Systems for Molecular Biology*, pages 262–271, Menlo Park, 1999. AAAI Press.

A Polynomial Time Approximation Scheme for the Closest Substring Problem

Bin Ma*

Department of Computer Science, University of Waterloo
Waterloo, ON, N2L 3G1, Canada
b3ma@wh.math.uwaterloo.ca.

Abstract. In this paper we study the following problem: Given n strings $s_1, s_2, \ldots, s_n$, each of length m, find a substring t_i of length L for each s_i, and a string s of length L, such that $\max_{i=1}^{n} d(s, t_i)$ is minimized, where $d(\cdot, \cdot)$ is the Hamming distance. The problem was raised in [6] in an application of genetic drug target search and is a key open problem in many applications [7]. The authors of [6] showed that it is NP-hard and can be trivially approximated within ratio 2. A non-trivial approximation algorithm with ratio better than 2 was found in [7]. A major open question in this area is whether there exists a polynomial time approximation scheme (PTAS) for this problem. In this paper, we answer this question positively. We also apply our method to two related problems.

1 Introduction

Let s be a string, without further specification, we suppose it is a string over alphabet $\Sigma = \{1, 2, \ldots, A\}$. Denote $l(s)$ as the length of s. Let s and s' be two strings of same length. Then $d(s, s')$ denotes the Hamming distance between s and s'. Let s and s' be two strings that $l(s) \geq l(s')$. s is said to be *d-close* to s' if it contains a substring t of length $l(s')$ such that $d(t, s') \leq d$. The closest substring problem is defined as:

Closest Substring Problem. Given a set $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ of strings each of length m, and an integer L, find a "center" string s and a substring t_i of length L for each s_i, minimizing d such that for each $1 \leq i \leq n$, $d(s, t_i) \leq d$.

The closest substring problem was introduced in [6] and is a key theoretical open problem in applications such as antisense drug design [2,6], creating diagonal probes [1,9,6], creating universal PCR primers [5,6]. In these applications, one wants to distinguish two sets of strings by one short string, which is close to a substring of each string in one set and far from any string in another set. For example, in the applications of drug design, one wants to design a drug that would kill several closely related pathogenic bacteria while it would be relatively harmless to humans. In order to do this, one might look for a short strand of nucleic acid sequence which can bind to part of a vital gene of each bacteria

* Supported in part by NSERC Research Grant OGP0046506.

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 99–107, 2000.

yet cannot bind to any part of the genes of humans. This naturally raises the following question:

Distinguishing String Problem. Given a set $\mathcal{S}_b \subseteq \Sigma^m$ of (bad) strings, a set $\mathcal{S}_g \subseteq \Sigma^L$ of (good) strings and two thresholds d_b and d_g, find a string x such that s_b is d_b-close to x for every $s_b \in \mathcal{S}_b$, and $d(x, s_g) \geq d_g$ for every $s_g \in \mathcal{S}_g$.

The reason to call $\mathcal{S}_b$ "bad" is that it represents the sequences of harmful bacteria. Note that when $\mathcal{S}_g$ is empty, the distinguishing string problem is actually the decision version of the closest substring problem. In contrast, when $\mathcal{S}_b$ is empty, the problem is the decision version of the *farthest string problem* studied in [6], which seems easier than the closest substring (string) problem and has a PTAS, as proved in [6] by a standard technique. In practice, d_b is usually small and d_g is usually large. Therefore, to find a string that satisfies the conditions for $\mathcal{S}_b$ is more decisive than for $\mathcal{S}_g$. We will see this more clearly in Section 4. Therefore, the closest substring problem is more crucial in these applications. However, this problem behaved so elusive that [6,7,8] had to study an easier version of its:

Closest String Problem. Given a set $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ of strings each of length m, find a "center" string s of length m minimizing d such that for each $1 \leq i \leq n$, $d(s_i, s) \leq d$.

Analogous to the closest substring problem, the authors of [6] also introduced another related problem, the max close string problem, which tends to find a string close to as many "bad" strings as possible:

Max Close String Problem. Given a set $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ of strings of length at least L, and a threshold $d \geq 0$, find a string s of length L maximizing the number of strings s_i in $\mathcal{S}$ that is d-close to s.

The closest string problem has been studied widely and independently in different contexts. In the context of coding theory it was shown to be NP-hard [3]. In DNA sequence related topics, the authors of [1] used a standard linear programming and random rounding technique and gave a near-optimal algorithm only for large d(super-logarithmic in number of sequences). Then in [6], the authors gave a $\frac{4}{3}$ approximation algorithm. They use the linear programming and random rounding technique only at $O(d_{opt})$ positions of the solution, while use a certain string in $\mathcal{S}$ to approximate the solution at the other $L - O(d_{opt})$ positions. [4] also presented the $\frac{4}{3}$ approximation with a similar idea. Later, the authors of [7] used a more sophisticated method to find the $O(d_{opt})$ positions and achieved a PTAS for the problem.

However, the closest substring problem seemed much more elusive. It is certainly NP hard as the closest string problem is. It admits a trivial ratio 2 approximation as shown in [6]. In [7], the authors found a nontrivial approximation algorithm, with approximation ratio $2 - \frac{2}{2|\Sigma|+1}$. At $O(d_{opt})$ positions of the solution, they used a random string to approximate the optimal solution for sometime. This gains about $\frac{1}{|\Sigma|}$ reduction from the trivial ratio 2. However, when $|\Sigma|$ is large, e.g. 20 for protein sequences, this improvement is very small.

The methods that solved the closest string problem cannot be extended straightforward to the closest substring problem. This is because that we do not know how to generate the linear programming on the $O(d_{opt})$ positions. Only when the optimal solution such that $d_{opt} = O(\log(nm))$, one can find the best solution at the $O(d_{opt})$ positions by trying all possibilities, instead of using linear programming approach. By doing this, [11] gave a PTAS for the closest substring problem when $d_{opt} = O(\log(nm))$.

In this paper, combining the methods in [7] and a *random sampling* technique, we present a PTAS for the closest substring problem. We will also prove that the max close string problem cannot be approximated within ratio n^{ϵ} for any $\epsilon \in (0, \frac{1}{4})$ unless P = NP. At last, as two applications of the above PTAS, we give suboptimal solutions to the max close string problem and the distinguishing string problem.

2 A PTAS for the Closest Substring Problem

Let t be a string of length m. Denote the j-th letter t by $t[j]$. Let $R = \{j_1, j_2, \ldots, j_k\}$ be a multiset of positions, where $1 \leq j_i \leq m$. Then $t|_R$ is defined by a length-k string $t[j_1]t[j_2]\cdots t[j_k]$.

First, let us recall the PTAS for the closest string problem in [7]. Let $\mathcal{S} = \{t_1, t_2, \ldots, t_n\}$ be a set of n strings of length m. Suppose s is the closest string of $\mathcal{S}$, i.e., s such that $\max_{i=1}^{n} d(s, t_i)$ is minimized. Denote $d_{opt} = \max_{i=1}^{n} d(s, t_i)$. For any given $r \geq 2$, let $1 \leq i_1, i_2, \ldots, i_r \leq n$ be r distinct numbers. Let $Q_{i_1,i_2,\ldots,i_r}$ be the set of positions where $t_{i_1}, t_{i_2}, \ldots, t_{i_r}$ agree. The following lemma is proved in [7] (Claim 16 and 17):

Lemma 1. [7] *Let $\rho_0 = \max_{1\leq i,j\leq n} d_H(t_i, t_j)/d_{opt}$. For any constant r, if $\rho_0 > 1+\frac{1}{2r-1}$, then there are indices $1 \leq i_1, i_2, \ldots, i_r \leq n$ such that for any $1 \leq l \leq n$,*

$$d(t_l|_{Q_{i_1,i_2,\ldots,i_r}}, t_{i_1}|_{Q_{i_1,i_2,\ldots,i_r}}) - d(t_l|_{Q_{i_1,i_2,\ldots,i_r}}, s|_{Q_{i_1,i_2,\ldots,i_r}}) \leq \frac{1}{2r-1} d_{opt}.$$

This lemma ensures the authors of [7] to use s_{i_1} to approximate s at the positions in $Q_{i_1,i_2,\ldots,i_r}$. Let $Q = Q_{i_1,i_2,\ldots,i_r}$ and $P = \{1, 2, \ldots, m\} - Q$. It is easy to verify that $|P| \leq r\, d_{opt} = O(d_{opt})$. At the positions in P, s can be approximated by solving the following optimization problem:

$$\begin{cases} \min \ \ d; \\ d(t_i|_P, x) \leq d - d(t_i|_Q, t_{i_1}|_Q), \ \ i = 1, \cdots, n; |x| = |P|. \end{cases} \tag{1}$$

Since it can be proved that the optimal solution of the above problem such that $d = \Theta(|P|)$, it can be approximately solved by i) rewriting it to be a zero-one optimization problem; ii) solving the relaxed linear program; and iii) random rounding. At last, combining the approximate solution x of (1) at P and s_{i_1} at Q gives a PTAS.

The algorithm for the closest string problem cannot be extended to the closest substring problem straightforward, since we do not know which substring t_i

$(i = 1, 2, \ldots, n)$ to use in the optimization problem (1). It is easy to see that the choice of good t_i's is the only obstacle on the way to the solution. Our strategy is *random sampling*.

Now let us outline the main ideas. Let $\langle \mathcal{S} = \{s_1, s_2, \ldots, s_n\}, L\rangle$ be an instance of Closest Substring, where s_i is of length m. Suppose that s is its optimal center string and t_i is a length L substring of s_i which is the closest to s $(i = 1, 2, \ldots, n)$. Let $d_{opt} = \max_{i=1}^{n} d(s, t_i)$. By trying all possibilities, we assume that $t_{i_1}, t_{i_2}, \ldots, t_{i_r}$ are the r substrings t_{i_j} that satisfy Lemma 1. Let Q be the set of positions where $t_{i_1}, t_{i_2}, \ldots, t_{i_r}$ agree and $P = \{1, 2, \ldots, L\} - Q$. By Lemma 1, $t_{i_1}|_Q$ is a good approximation to $s|_Q$. We want to approximate $s|_P$ by the solution x of the following optimization problem (2), where t'_i is a substring of s_i and is up to us to choose.

$$\begin{cases} \min \ \ d; \\ d(t'_i|_P, x) \le d - d(t'_i|_Q, t_{i_1}|_Q), \ \ i = 1, \cdots, n; |x| = |P|. \end{cases} \tag{2}$$

The ideal choice is $t'_i = t_i$, *i.e.*, t'_i is the closest to s among all substrings of s_i. However, we only approximately know s in Q and know nothing about s in P so far. So, we randomly pick $O(\log(mn))$ positions from P. Suppose the multiset of these random positions is R. Since $|R| = O(\log(mn))$, by trying all possibilities, we can assume that $s|_R$ is known. We then find the substring t'_i from s such that $d(s|_R, t'_i|_R) \times \frac{|P|}{|R|} + d(t_{i_1}|_Q, t'_i|_Q)$ is minimized. Then t'_i potentially belongs to the substrings of s_i that are the closest to s.

Then we solve (2) approximately by the method provided in [7] and combine the solution x at P and t_{i_1} at Q, the resulting string should be a good approximation to s.

The more detailed algorithm (Algorithm closestSubstring) is given in Figure 1. We prove Theorem 1 in the rest of the section. Before the proof, we need a lemma which is commonly known as Chernoff's bounds ([10], Theorem 4.2 and 4.3):

Lemma 2. [10] *Let $X_1, X_2, \ldots, X_n$ be n independent random 0-1 variables, where X_i takes 1 with probability p_i, $0 < p_i < 1$. Let $X = \sum_{i=1}^{n} X_i$, and $\mu = E[X]$. Then for any $\delta > 0$,*

(1) $\mathbf{Pr}(X > (1+\delta)\mu) < \left[\frac{\mathbf{e}^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\mu}$,
(2) $\mathbf{Pr}(X < (1-\delta)\mu) \le \exp\left(-\frac{1}{2}\mu\delta^2\right)$.

From Lemma 2, we can prove the following lemma:

Lemma 3. *Let X_i, X and μ be defined as in Lemma 2. Then for any $0 < \epsilon \le 1$,*

(1) $\mathbf{Pr}(X > \mu + \epsilon\, n) < \exp\left(-\frac{1}{3}n\epsilon^2\right)$,
(2) $\mathbf{Pr}(X < \mu - \epsilon\, n) \le \exp\left(-\frac{1}{2}n\epsilon^2\right)$.

Proof. (1) Let $\delta = \frac{\epsilon n}{\mu}$. By Lemma 2,

$$\mathbf{Pr}(X > \mu + \epsilon n) < \left[\frac{\mathbf{e}^{\frac{\epsilon n}{\mu}}}{(1+\frac{\epsilon n}{\mu})^{(1+\frac{\epsilon n}{\mu})}}\right]^{\mu} = \left[\frac{\mathbf{e}}{(1+\frac{\epsilon n}{\mu})^{(1+\frac{\mu}{\epsilon n})}}\right]^{\epsilon n} \le \left[\frac{\mathbf{e}}{(1+\epsilon)^{1+\frac{1}{\epsilon}}}\right]^{\epsilon n},$$

Algorithm closestSubstring

Input n sequences $\{s_1, s_2, \ldots, s_n\} \subseteq \Sigma^m$, integer L.
Output the center string s.

1. **for** every r length-L substrings $t_{i_1}, t_{i_2}, \ldots, t_{i_r}$ (allowing repeats, but if t_{i_j} and t_{i_k} are both chosen from the same s_i then $t_{i_j} = t_{i_k}$) of $s_1, \ldots, s_n$ **do**
   (a) $Q = \{1 \le j \le L \mid t_{i_1}[j] = t_{i_2}[j] = \ldots = t_{i_r}[j]\}$, $P = \{1, 2, \ldots, L\} - Q$.
   (b) Let R be a multiset containing $\lceil \frac{4}{\epsilon^2} \log(nm) \rceil$ uniformly random positions from P.
   (c) **for** every string y of length $|R|$ **do**
      (i) **for** i from 1 to n **do**
         Let t'_i be a length L substring of s_i minimizing $d(y, t'_i|_R) \times \frac{|P|}{|R|} + d(t_{i_1}|_Q, t'_i|_Q)$.
      (ii) Using the method in [7], solve the optimization problem defined by Formula (2) approximately. Let x be the approximate solution within error $\epsilon |P|$.
      (iii) Let s' be the string such that $s'|_P = x$ and $s'|_Q = t_{i_1}|_Q$. Let $c = \max_{i=1}^{n} \min_{\{t_i \text{ is a substring of } s_i\}} d_H(s', t_i)$.
2. **for** every length-L substring s' of s_1 **do**
      Let $c = \max_{i=1}^{n} \min_{\{t_i \text{ is a substring of } s_i\}} d_H(s', t_i)$.
3. Output the s' with minimum c in step 1(c)(iii) and step 2.

Fig. 1. The PTAS for the closest substring problem.

where the last inequality is because $\mu \le n$ and that $(1+x)^{(1+\frac{1}{x})}$ is increasing for $x \ge 0$. It is easy to verify that for $0 < \epsilon \le 1$, $\frac{\mathbf{e}}{(1+\epsilon)^{1+\frac{1}{\epsilon}}} \le \exp\left(-\frac{\epsilon}{3}\right)$. Therefore, (1) is proved.

(2) Let $\delta = \frac{\epsilon n}{\mu}$. By Lemma 2, (2) is proved. □

Theorem 1. *Algorithm closestSubstring is a PTAS for the closest substring problem.*

Proof. Let s be an optimal center string and t_i be the length-L substring of s_i that is the closest to s. Let $d_{opt} = \max d(s, t_i)$. Let ϵ be any small positive number and $r \ge 2$ be any fixed integer. Let $\rho_0 = \max_{1 \le i,j \le n} d(s_i, s_j)/d_{opt}$. If $\rho_0 \le 1 + \frac{1}{2r-1}$, then clearly we can find a solution s' within ratio ρ_0 in step 2. So, we suppose $\rho_0 \ge 1 + \frac{1}{2r-1}$ from now on.

For convenience, for any position multiset T and two strings t_1 and t_2. we denote $d^T(t_1, t_2) = d(t_1|_T, t_2|_T)$. By Lemma 1, Algorithm closestSubstring picks a group of $t_{i_1}, t_{i_2}, \ldots, t_{i_r}$ in step 1 at a certain time such that

Fact 1. For any $1 \le l \le n$, $d^Q(t_{i_1}, t_l) - d^Q(t_{i_1}, s) \le \frac{1}{2r-1} d_{opt}$.

Obviously, the algorithm takes y as $s|_R$ for a certain time in step 1(c). Let $y = s|_R$ and $t_{i_1}, t_{i_2}, \ldots, t_{i_r}$ satisfy Fact 1. Let t'_i be defined as in step 1(c)(i). Let s^* be a string such that $s^*|_P = s|_P$ and $s^*|_Q = t_{i_1}|_Q$. Then we claim:

Fact 2. With high probability, $d(s^*, t'_i) \le d(s^*, t_i) + 2\epsilon|P|$ for all $1 \le i \le n$.

Proof. Let $\rho = \frac{|P|}{|R|}$. For any substring t' of length L of s_i such that

$$d(s^*, t') \geq d(s^*, t_i) + 2\epsilon|P|, \tag{3}$$

from the definition of s^*, the following inequality is obviously,

$$\begin{aligned} &\mathbf{Pr}\left(\rho\, d(y, t'|_R) + d^Q(t_{i_1}, t') \leq \rho\, d(y, t_i) + d^Q(t_{i_1}, t_i)\right) \\ \leq\ &\mathbf{Pr}\left(\rho\, d^R(s^*, t') + d^Q(s^*, t') \leq d(s^*, t') - \epsilon|P|\right) + \\ &\mathbf{Pr}\left(\rho\, d^R(s^*, t_i) + d^Q(s^*, t_i) \geq d(s^*, t_i) + \epsilon|P|\right). \end{aligned} \tag{4}$$

It is easy to see that $d^R(s^*, t')$ is a sum of $|R|$ independent random 0-1 variables X_i, $i = 1, 2, \ldots, |R|$, where X_i indicates if s^* mismatches t' at the i-th position in R. Let $\mu = E[d^R(s^*, t')]$. Then obviously, $\mu = d^P(s^*, t')/\rho$. Therefore, by Lemma 3,

$$\begin{aligned} &\mathbf{Pr}\left(\rho\, d^R(s^*, t') + d^Q(s^*, t') \leq d(s^*, t') - \epsilon|P|\right) \\ =\ &\mathbf{Pr}\left(d^R(s^*, t') \leq (d(s^*, t') - d^Q(s^*, t'))/\rho - \epsilon|R|\right) \\ =\ &\mathbf{Pr}\left(d^R(s^*, t') \leq \mu - \epsilon|R|\right) \leq \exp\left(-\frac{1}{2}\epsilon^2|R|\right) \leq (nm)^{-2}, \end{aligned} \tag{5}$$

where the last inequality is because of $|R| = \lceil \frac{4}{\epsilon^2}\log(nm) \rceil$ by step 1(b) of the algorithm. For the same reason, we have

$$\mathbf{Pr}\left(\rho\, d^R(s^*, t_i) + d^Q(s^*, t_i) \geq d(s^*, t_i) + \epsilon|P|\right) \leq (nm)^{-\frac{4}{3}}. \tag{6}$$

Combining Formula (4)(5)(6), we know that for any t' that satisfies Formula (3),

$$\mathbf{Pr}\left(\rho\, d(y, t'|_R) + d^Q(t_{i_1}, t') \leq \rho\, d(y, t_i) + d^Q(t_{i_1}, t_i)\right) \leq 2\,(nm)^{-\frac{4}{3}}. \tag{7}$$

For any fixed $1 \leq i \leq n$, there are less than m substrings t' that satisfies Formula (3). Thus, from Formula (7) and the definition of t'_i,

$$\mathbf{Pr}\left(d(s^*, t'_i) \geq d(s^*, t_i) + 2\epsilon|P|\right) \leq 2\, n^{-\frac{4}{3}} m^{-\frac{1}{3}}. \tag{8}$$

Summing up all $i \in [1, n]$, we know that with probability at least $1 - 2\,(nm)^{-\frac{1}{3}}$, $d(s^*, t'_i) \leq d(s^*, t_i) + 2\epsilon|P|$ for all i. □

From Fact 1, $d(s^*, t_i) = d^P(s, t_i) + d^Q(t_{i_1}, t_i) \leq d(s, t_i) + \frac{1}{2r-1}\, d_{opt}$. Combining with Fact 2 and $|P| \leq r\, d_{opt}$, we get

$$d(s^*, t'_i) \leq (1 + \frac{1}{2r-1} + 2\epsilon\, r) d_{opt}. \tag{9}$$

By the definition of s^*, the optimization problem defined by Formula (2) has a solution $s|_P$ such that $d \leq (1 + \frac{1}{2r-1} + 2\epsilon\, r) d_{opt}$. Solving it within error $\epsilon|P|$ by the method in [7], suppose x is the solution. Then by Formula (2), for any $1 \leq i \leq n$,

$$d(t'_i|_P, x) \leq (1 + \frac{1}{2r-1} + 2\epsilon\, r) d_{opt} - d(t'_i|_Q, t_{i_1}|_Q) + \epsilon|P|. \tag{10}$$

Let s' be defined in step 1(c)(iii), then by Formula (10),

$$\begin{aligned} d(s', t'_i) &= d(x, t'_i|_P) + d(t_{i_1}|_Q, t'_i|_Q) \\ &\leq (1 + \frac{1}{2r-1} + 2\epsilon r)d_{opt} + \epsilon|P| \\ &\leq (1 + \frac{1}{2r-1} + 3\epsilon r)d_{opt}. \end{aligned}$$

It is easy to see that the algorithm runs in polynomial time for any fixed positive r and ϵ. For any $\delta > 0$, by properly setting r and ϵ such that $\frac{1}{2r-1}+3\epsilon r \leq \delta$, with high probability, the algorithm outputs in polynomial time a solution s' such that s_i is $(1 + \delta)d_{opt}$-close to s' for every $1 \leq i \leq n$. The algorithm can be derandomized by standard methods such as the *method of conditional probabilities* [10]. Thus, Theorem 1 is correct. □

3 Approximating the Max Close String Problem

If we could solve the max close string problem $\langle \mathcal{S}, L, d \rangle$ (in polynomial time), then we might try all $d = 0, 1, 2, \ldots, L$ and find the least d such that all the strings in $\mathcal{S}$ are d-close to a string s. It is easy to verify that this d is the optimal value of the closest substring problem $\langle \mathcal{S}, L \rangle$. Since the closest substring problem is NP-hard, we know that the max close string problem is NP-hard too. Actually, we can show a much stronger hardness result:

Theorem 2. *For any $0 < \epsilon < \frac{1}{4}$, the max close string problem cannot be approximated within ratio n^ϵ in polynomial time, unless $P = NP$.*

Proof. First, let us prove the theorem for the case $\Sigma = \{0, 1\}$. We reduce the far from most string problem to it.

Far from Most String Problem. Given a set $\mathcal{S}$ of strings of length m and a threshold $d \geq 0$. Find a string x of length m maximizing the number of strings s in $\mathcal{S}$ that satisfies $d(x, s) \geq d$.

In [6], it is proved that for any finite size alphabet Σ, unless P = NP, the far from most string problem does not admit an approximation algorithm within ratio n^ϵ for any $0 < \epsilon < \frac{1}{4}$. Suppose $I =< \mathcal{S}, d >$ is one of its instance such that $\Sigma = \{0, 1\}$. Let $\overline{\mathcal{S}} = \{\overline{s} \mid s \in \mathcal{S}\}$, where $\overline{s}$ is the string gotten by flipping every bit of s. We construct an instance of the max close string problem as $I' =< \overline{\mathcal{S}}, m, m - d >$.

It is easy to see that the following properties are identical:

1. I has a solution s which is far from k strings in $\mathcal{S}$ with distance at least d.
2. s is close to k strings in $\overline{\mathcal{S}}$ with distance at most $m - d$.
3. I' has a solution s which is $m - d$ close to k strings in $\mathcal{S}$.

Therefore, I is identical to I'. Since the hardness of approximating for the far from most string problem, the theorem is correct for $\Sigma = \{0, 1\}$.

Now let $\{0, 1\} \subset \Sigma$. Suppose we have an approximation algorithm with ratio $\rho > 1$ for the max close string problem over Σ. Let I be an instance of the

max close string problem over $\{0,1\}$. We solve it with the ratio-ρ approximation algorithm over Σ. Suppose s is the solution. We modify s by replacing every character not in $\{0,1\}$ by 0 or 1 arbitrarily. Obviously, this will not make the solution worse. However, s is a solution over $\{0,1\}$ now. It is easy to verify that this solution has a ratio at most ρ. Thus, as the theorem is correct for the alphabet $\{0,1\}$, it is correct for any alphabet Σ. □

Although the max close string problem is hard to approximate according to Theorem 2, the following theorem shows that we can approximate it in another sense.

Theorem 3. *Given an instance of the max close string problem $\langle \mathcal{S}, L, d\rangle$. Suppose an optimal solution s such that there are k strings in $\mathcal{S}$ which contain length-L substrings within Hamming distance d to s. For any $\epsilon' > 0$, we can find in polynomial time an s' such that there are at least k strings in $\mathcal{S}$ which is $(1+\epsilon')d$-close to s'.*

Proof. The algorithm is quite similar to Algorithm closestSubstring, except for that in step 1(c)(i) we only keep the t'_i's for those i's satisfying $d(y, t'_i|_R) \times \frac{|P|}{|R|} + d(t_{i_1}|_Q, t'_i|_Q) \leq (1 + \frac{1}{2r-1} + 2\epsilon r)d$. Then by the same arguments of Theorem 1, we know that with high probability, we have at least k length-L substrings t'_i from at least k distinct strings in $\mathcal{S}$ and a length-L string s' such that $d(s', t'_i) \leq (1 + \frac{1}{2r-1} + 3\epsilon r)d$. Set r and ϵ so that $\epsilon' = \frac{1}{2r-1} + 3\epsilon r$. The theorem is proved. □

4 Approximating the Distinguishing String Problem

In this section, we give a suboptimal solution to the distinguishing string problem, using the algorithm for the closest substring problem. The idea is as follows: Since d_b is significantly less than d_g, whenever we obtain a good solution at the "bad" string side, by Triangular Inequality, it is not too bad at the "good" string side. In this sense, the closest substring problem is more crucial to the distinguishing string problem than the farthest string problem is.

Theorem 4. *For any $\epsilon' > 0$, there is an algorithm such that if an instance of the distinguishing string problem $I =< \mathcal{S}_b, \mathcal{S}_g, d_b, d_g, L >$ has a solution s, then the algorithm outputs a string s' in polynomial time such that y is a solution of $I' =< \mathcal{S}_b, \mathcal{S}_g, (1+\epsilon')d_b, d_g - (2+\epsilon')d_b, L >$.*

Proof. We invoke Algorithm closestSubstring to solve the closest substring problem $\langle \mathcal{S}_b, L\rangle$. However, at step 1(c)(i), whenever $i = i_j$, we let $t'_i = t_{i_j}$; at step 3, we output the s' satisfying $c \leq (1+\epsilon')d_b$ and $d(s', s_g) \geq d_g - (2+\epsilon')d_b$ for any $s_g \in \mathcal{S}_g$. We need to show that this s' exists in step 1 or step 2.

Let s be a solution of I and t_i be a substring of $s_i \in \mathcal{S}_b$ such that $d(s, t_i) \leq d_b$. Following the proof of Theorem 1, in the modified algorithm closestSubstring, there are t'_i $(i = 1, 2, \ldots, n)$ such that $t'_{i_j} = t_{i_j}$ and the s' defined in step 1(c)(iii) satisfies that

$$d(s', t'_i) \leq (1+\epsilon')d_b. \tag{11}$$

Particularly,

$$d(s', t_{i_j}) = d(s', t'_{i_j}) \leq (1 + \epsilon')d_b. \tag{12}$$

Since $d(s, t_{i_j}) \leq d_b$ and $d(s, s_g) \geq d_g$ for any $s_g \in \mathcal{S}_g$, by Triangular Inequality, we have

$$d(s_g, t_{i_j}) \geq d_g - d_b. \tag{13}$$

Further combining with Formula (12) and Triangular Inequality, we know that $d(s_g, s') \geq (d_g - d_b) - (1+\epsilon')d_b = d_g - (2+\epsilon')d_b$. Combining with Formula (11), the theorem is proved. □

Remark. By combining the conditions for $\mathcal{S}_g$ into the optimization problem (2), we in fact can get an algorithm that y is a solution of $I' =< \mathcal{S}_b, \mathcal{S}_g, (1+\epsilon')d_b, d_g - (1+\epsilon')d_b, L >$ in Theorem 4. Since the proof is tedious and has no new idea, we omit it.

Acknowledgment

The author thanks Dr. M. Li and L. Wang for their valuable discussions on the topic.

References

1. A. Ben-Dor, G. Lancia, J. Perone, and R. Ravi, Banishing Bias from Consensus Sequences, *Combinatorial Pattern Matching, 8th Annual Symposium*, Springer-Verlag, Berlin, 1997.
2. S. Crooke and B. Lebleu (editors), *Antisense Research and Applications*, CRC Press, 1993.
3. M. Frances and A. Litman, On covering problems of codes, *Theory of Computing Systems*, vol. 30, pp. 113-119, 1997.
4. L. Gąsieniec, J. Jansson, and A. Lingas, Efficient approximation algorithms for the Hamming center problem, *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 905-906, San Francisco, 1999.
5. E. M. Hillis, C. Moritz, and B. K. Mable, *Molecular Systematics*, 2nd ed., Sinauer Associates Inc., Sunderland, 1996.
6. K. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang, Distinguish string search problems, *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 633-642, San Francisco, 1999.
7. M. Li, B. Ma, and L. Wang, Finding similar regions in many strings, *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing*, pp. 473-482, Atlanta, 1999.
8. B. Ma, Some approximation algorithms on strings and trees, *Ph.D. Thesis*, Peking University, 1999.
9. A. Macario and E. Macario, *Gene Probes for Bacteria*, Academic Press, 1990.
10. R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.
11. L. Wang, *Personal Communication*, 1999.

Approximation Algorithms for Hamming Clustering Problems

Leszek Gąsieniec[1], Jesper Jansson[2], and Andrzej Lingas[2]

[1] Dept. of Computer Science, University of Liverpool
Peach Street, L69 7ZF, UK
leszek@csc.liv.ac.uk
[2] Dept. of Computer Science, Lund University Box 118, 221 00 Lund, Sweden
{Jesper.Jansson,Andrzej.Lingas}@cs.lth.se

Abstract. We study Hamming versions of two classical clustering problems. The *Hamming radius p-clustering* problem (HRC) for a set S of k binary strings, each of length n, is to find p binary strings of length n that minimize the maximum Hamming distance between a string in S and the closest of the p strings; this minimum value is termed the *p-radius of S* and is denoted by ϱ. The related *Hamming diameter p-clustering* problem (HDC) is to split S into p groups so that the maximum of the Hamming group diameters is minimized; this latter value is called the *p-diameter of S*.

First, we provide an integer programming formulation of HRC which yields exact solutions in polynomial time whenever k and p are constant. We also observe that HDC admits straightforward polynomial-time solutions when $k = O(\log n)$ or $p = 2$. Next, by reduction from the corresponding geometric p-clustering problems in the plane under the L_1 metric, we show that neither HRC nor HDC can be approximated within any constant factor smaller than two unless P=NP. We also prove that for any $\epsilon > 0$ it is NP-hard to split S into at most $pk^{1/7-\epsilon}$ clusters whose Hamming diameter doesn't exceed the p-diameter. Furthermore, we note that by adapting Gonzalez' farthest-point clustering algorithm [6], HRC and HDC can be approximated within a factor of two in time $O(pkn)$. Next, we describe a $2^{O(p\varrho/\varepsilon)}k^{O(p/\varepsilon)}n^2$-time $(1+\varepsilon)$-approximation algorithm for HRC. In particular, it runs in polynomial time when $p = O(1)$ and $\varrho = O(\log(k+n))$. Finally, we show how to find in $O((\frac{n}{\varepsilon} + kn\log n + k^2\log n)(2^\varrho k)^{2/\varepsilon})$ time a set L of $O(p\log k)$ strings of length n such that for each string in S there is at least one string in L within distance $(1+\varepsilon)\varrho$, for any constant $0 < \varepsilon < 1$.

1 Introduction

Let $\mathbb{Z}_2^n$ be the set of all strings of length n over the alphabet $\{0,1\}$. For any $\alpha \in \mathbb{Z}_2^n$, we use the notation $\alpha[i]$ to refer to the symbol placed at the ith position of α, where $i \in \{1,..,n\}$. The *Hamming distance* between $\alpha_1, \alpha_2 \in \mathbb{Z}_2^n$ is defined as the number of positions in which the strings differ, and is denoted by $d(\alpha_1, \alpha_2)$.

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 108–118, 2000.

The *Hamming radius p-clustering* problem[1] (HRC) is stated as follows: Given a set S of k binary strings $\alpha_i \in \mathbb{Z}_2^n$, where $i = 1, .., k$, and a positive integer p, find p strings $\beta_j \in \mathbb{Z}_2^n$, where $j = 1, .., p$, minimizing the value $\varrho = \max_{1 \le i \le k} \min_{1 \le j \le p} d(\alpha_i, \beta_j)$. Such a set of β_j's is called a *p-center set of S*, and the corresponding value of ϱ is called the *p-radius of S*. Note that an instance of HRC can have several p-center sets.

The *Hamming diameter p-clustering* problem (HDC) is defined on the same set of instances as HRC, and is stated as follows: Partition S into p disjoint subsets $S_1, .., S_p$ (called *p-clusters of S*) so that the value of $\max_{1 \le q \le p} \max_{\alpha_i, \alpha_j \in S_q} d(\alpha_i, \alpha_j)$ is minimized. This value is called the *p-diameter of S*.

One can immediately generalize HRC and HDC by considering a larger finite size alphabet instead of $\{0, 1\}$, making the problem more amenable to biological applications. However, as long as the distance between two different characters is measured as one, such a generalization involves only trivial generalizations of our approximation methods. Therefore, we only consider the original binary versions of HRC and HDC throughout this paper.

In [4], Frances and Litman showed that the decision version of the Hamming radius 1-clustering problem (1-HRC) is NP-complete. Motivated by the intractability of 1-HRC and its applications in computational biology, coding theory, and data compression, two groups of authors recently provided several close approximation algorithms [5,12]. This was followed by a polynomial-time approximation scheme (PTAS) for 1-HRC [13]. As for the more general HRC and HDC, one can merely find work on the related graph or geometric p-center, p-supplier, and p-clustering problems in the literature [3,8,9,10,15]. In the undirected complete graph case, with edge weights satisfying the triangle inequality, all of the three aforementioned problems are known to admit 2-approximation or 3-approximation polynomial-time algorithms, but none of them are approximable within $2-\varepsilon$ for any $\varepsilon > 0$ in polynomial-time unless P=NP [8,9,10]. This contrasts with the $p = O(1)$ case when, e.g., the graph p-center and p-supplier problems can be trivially and exactly solved in $n^{O(p)}$ time. HRC doesn't seem easier than these graph problems. Optimal or nearly optimal center solutions to it have to be searched in $\mathbb{Z}_2^n$ whose size might be exponential in the input size. For this reason, HRC is NP-complete already for $p = 1$. Our results indicate that in the general case HRC as well as HDC are equally hard to approximate in polynomial time as the p-center or p-clustering graph problems are.

1.1 Motivation

Clustering is used to solve classification problems in which the elements of a specified set have to be divided into classes so that all members of a class are similar to each other in some sense. HRC and HDC are equally fundamental problems within strings algorithms as the corresponding graph and geometric

[1] The corresponding graph problem is often termed the *p-center* problem in the literature [8].

center and clustering problems are within graph algorithms or computational geometry respectively [3,8,9,10,15]. They have potential applications in computational biology and pattern matching.

For example, when classifying biomolecular sequences, consensus representatives are useful. The around 100000 different proteins in humans can be divided into 1000 (or less) protein families, which makes it easier for researchers to understand their structures and biological functions [7]. A lot of information about a newly discovered protein may be deduced by establishing which family it belongs to. During identification, it is more efficient to try to align the new protein to representatives for various families than to individual family members. Conversely, given a set S of k related sequences, one way to find other similar sequences is by computing p representatives (where $p << k$) for S and then using the representatives to probe a genome database. The representatives should resemble all sequences in S, and must be chosen carefully. For instance, when $p = 1$, the sequence s that minimizes the sum of all pairwise distances between s and elements in S is biased towards sequences that occur frequently, but using a 1-center as representative will avoid this problem[2]. For $p > 1$, the representatives can be the members in the p-center set or simply p sequences, each from a different p-cluster.

In pattern matching applications, the number of classes p can be large; a system for Chinese character recognition, for example, would need to be able to discriminate between thousands of characters.

1.2 Organization of the Paper

Section 2 provides polynomial-time solutions for restricted cases of HRC and HDC based on integer programming, exhaustive search, and breadth-first search. In Section 3, we prove the NP-hardness of approximating HRC and HDC within any constant factor smaller than two. In the same section, we also prove that another type of approximation for HDC in terms of the number of clusters is NP-hard. Section 4 presents three approximations algorithms for HRC and HDR: a two-approximation algorithm for HRC and HDC based on Gonzalez' furthest-point clustering method [6], an approximation scheme, i.e., a $(1 + \varepsilon)$-approximation algorithm for HRC, and a $(1 + \varepsilon)$-approximation algorithm for HRC using a moderately larger number of approximative centers.

2 Polynomial-Time Solutions for Restricted Cases

The Hamming radius p-clustering problem is equivalent to a special case of the integer programming problem. A given instance $(\alpha_1, .., \alpha_k, p, \varrho)$ of the decision version of HRC, where $\alpha_i \in \mathbb{Z}_2^n$ for $1 \leq i \leq k$, and $p, \varrho \in \mathbb{N}$, can be expressed as a system of $k \cdot p$ linear inequalities.

[2] Depending on the application, the difference between strings is sometimes measured in terms of edit distance, which also takes insertions and deletions into account, rather than Hamming distance, which just considers substitutions.

We use two matrices X and Y of 0-1-variables. The rows of X correspond directly to the p strings that constitute a p-center for the supplied instance, and Y is used to make sure that each α_i is within distance ϱ of at least one of the centers.

Let X be a $p \times n$-matrix of variables $x_{jm} \in \mathbb{Z}_2$, where $1 \leq j \leq p$ and $1 \leq m \leq n$. The value of x_{jm} determines the value of the m-th position of the j-th center. Let Y be a $k \times p$-matrix of variables $y_{ij} \in \mathbb{Z}_2$, where $1 \leq i \leq k$ and $1 \leq j \leq p$. $y_{ij} = 1$ only if row j of X is a center string that is closest to α_i, so that for each $i = 1, .., k$, we have $\sum_{j=1}^{p} y_{ij} = 1$. Next, for each $i = 1, .., k$ and $j = 1, .., p$, we have the inequality

$$\sum_{\substack{\alpha_i[m]=0 \\ 1 \leq m \leq n}} x_{jm} + \sum_{\substack{\alpha_i[m]=1 \\ 1 \leq m \leq n}} (1 - x_{jm}) \leq \varrho + (1 - y_{ij}) \cdot D$$

where $D = \max_{1 \leq j \leq k} (\max_{1 \leq i \leq k} d(\alpha_i, \alpha_j))$.

The above system of inequalities can be transformed to the form $Ax \leq b$, where A is a $(kp) \times (np + kp)$ integer matrix, x is a variable vector over $\mathbb{Z}_2^{np+kp}$, and b is a vector in $\mathbb{Z}_2^{kp}$. Note that the scalar product of any prefix of any row of A with a $0-1$-vector of the same length is neither less than $-n$ nor greater than $n + D$. In particular, when $p = 1$, such a product has its absolute value simply bounded by n. Now, we can solve the transformed system of kp inequalities by a well-known dynamic programming procedure [14], proceeding in stages. At the jth stage, we compute the set S_j of all vectors that can be expressed as $\sum_{l=1}^{j} c_l z_l$, where c_l is the lth column of A and $z_l \in \mathbb{Z}_2$. Since the S_j cannot be larger than $(2n + D + 1)^{kp}$ (or $(2n + 1)^k$ if $p = 1$), the whole procedure for a fixed ϱ takes $O((2n + D)^{kp} 2^{kp}(np + kp))$ time (or $O(n^k 2^{2k}(n + k))$ if $p = 1$). Hence, by using binary search to find the smallest possible ϱ, we conclude that HRC for $k = O(1)$ and $p = O(1)$ can be solved in polynomial time.

Theorem 1. *HRC for instances with k strings of length n is solvable in $n^{O(kp)}$ time.*

On the other hand, if $n = O(\log k)$, exhaustive search yields a $k^{O(p)}$-time solution.

Theorem 2. *HRC restricted to instances with k strings of length $O(\log k)$ is solvable in $k^{O(p)}$ time.*

One of the main differences between HDC and HRC is that the former doesn't involve strings outside the input set S. For this reason it seems simpler to solve exactly than HRC does [3]. For example, it has a simpler integer programming formulation involving only a single matrix of indicator variables. Furthermore, it can be solved by exhaustive search in $O(k^2 n + k^2 p^k)$ time, which immediately yields the following result.

[3] Paradoxically, as for approximation in terms of the number of clusters it might be more difficult, as is observed in the next sections.

Theorem 3. *HDC restricted to instances with $O(\log n)$ strings of length n is solvable in $n^{O(\log p)}$ time.*

More interestingly, the Hamming diameter 2-clustering problem admits the following, rather straightforward polynomial-time solution. Let d be a candidate value for the maximum Hamming cluster diameter in an optimal 2-clustering of the k input strings of length n. Form a graph G with vertices in one-to-one correspondence with the input strings, and connect a pair of vertices by an edge whenever the Hamming distance between the corresponding strings is less than or equal to d. Now, the problem of Hamming diameter 2-clustering for the input strings becomes equivalent to that of partitioning the vertices of G into two cliques. The latter problem in turn reduces to 2-coloring the complement graph. By breadth-first search, we can find a 2-coloring of the complement graph, if one exists, in $O(k^2)$ time. To find the smallest possible d, we use the procedure just described to test different values of d, generated by a binary search. Calculating all pairwise Hamming distances requires $O(k^2 n)$ time, but this can be done before starting the search for d. Hence, we obtain the following result.

Theorem 4. *For $p = 2$, HDC is solvable in $O(k^2 n)$ time.*

Note that Theorem 4 can be generalized to any metric.

3 NP-Hardness of Approximating HRC and HDC

By approximating HRC or HDC, we mean providing a polynomial-time algorithm yielding a p-center set or a p-clustering approximating the p-radius or the p-diameter, respectively. Our results from the first subsection prove the NP-hardness of this type of approximation of HRC and HDC. In the second subsection, we consider another kind of approximation of HDC relaxing the requirement on the number of produced clusters under the condition that their diameter doesn't exceed the p-diameter; we show that it is NP-hard to approximate the number of clusters within any reasonable factor.

3.1 NP-Hardness of Approximating the p-Radius and p-Diameter

To prove the hardness results in this subsection, we use the reduction described in [3] from vertex cover for planar graphs of degree at most three to the corresponding p-clustering problem in the plane under the L_1 metric. (The *radius p-clustering problem in the plane under the L_1 metric* is the following: For a finite set S of points in the plane, find a set P of p points in the plane that minimizes $\max_{s \in S} \min_{u \in P} d_1(s, u)$, where d_1 is the L_1 distance. The *diameter p-clustering problem in the plane under the L_1 metric* is defined correspondingly.)

By straightforward inspection of the aforementioned reduction from vertex cover for planar graphs [3] and using, e.g., the planar graph drawing algorithm from [2] in order to embed the input planar graph in the plane, we can ensure

that the points in the resulting instance of the p-clustering problem in the plane as well as p points in an optimal solution lie on an integer grid of size polynomial in the size of the input planar graph and $(\alpha - 1)^{-1}$. This yields the following technical strengthening of Theorem 2.1 from [3].

Lemma 5. *Let α be a positive constant not less than 1. The radius p-clustering and diameter p-clustering problems for a finite set S of points in the plane with the L_1 metric, where the points in S lie on an integer grid of size polynomial in the cardinality of S and $(\alpha - 1)^{-1}$, and where the approximative solution to the radius version is required to lie on the grid, are NP-hard to approximate within α.*

By using the idea of embedding the L_1-metric on a integer square grid into the Hamming one, we obtain our main result in this section.

Theorem 6. *HRC and HDC are NP-hard to approximate within any constant factor smaller than two.*

Proof. Let S be a set of points on integer square grid of size $q(|S|)$ where $q()$ is a polynomial. Encode each grid point s of coordinates s_x and s_y respectively by the $0-1$ string $e(s)$ of length $2q(|S|)$ composed of s_x consecutive 1's followed by $q(|S|) - s_x$ consecutive 0's, next s_y consecutive 1's, and finally, $q(|S|) - s_y$ consecutive 0's. Note that for any two grid points s' and s'' their L_1 distance is equal to the Hamming distance between their encodings $e(s')$ and $e(s'')$. This observation yields immediately the theorem thesis for HDC by Lemma 5.

Consider an approximative solution $a_1, a_2, .., a_p$ to HRC problem for the strings $e(s)$, $s \in S$. For $i = 1, .., p$, we can transform a_i to a'_i having the form of $1^l 0^{q(|S|)-l} 1^m 0^{q(|S|)-m}$ for some $l, m \leq q$ by moving all the 1's in the first half of a_i to the appropriate prefix of a_i and similarly moving the remaining 1's to the appropriate prefix of the second half of a_i and filling the left positions with the left 0's. Observe that the resulting string sequence $a'_1, a'_2, .., a'_p$ yields at least as good solution as $a_1, a_2, .., a_p$ for the strings $e(s)$, $s \in S$ by the special form of the $e(s)$'s. Also, it can be immediately decoded into a sequence of grid points $g_1, g_2, .., g_p$ such that $a'_i = e(g_i)$ for $i = 1, .., p$. Putting everything together, we obtain the theorem thesis for HRC by Lemma 5. □

3.2 NP-Hardness of Approximating HDC in Terms of the Number of Clusters

Consider the following *clique partition problem*: Given an undirected graph G and a natural number p, partition the set of vertices of G into pairwise disjoint subsets $V_1, .., V_p$ such that for $j = 1, .., p$, the subgraph of G induced by V_j is a clique. Clearly, this problem is equivalent to coloring the complement graph with p colors. It follows from known inapproximability results for graph coloring [1] that for any $\epsilon > 0$, the problem of finding an approximative solution to the clique partition problem consisting of $pn^{1/7-\epsilon}$ cliques, where n is the number of vertices in the instance graph G, is NP-hard. For our purposes, it will be

convenient to assume that the instance graph is *quasi-regular*, by which we mean that it satisfies the following two properties:

1. It contains two distinguished cliques.
2. All vertices outside the two cliques have the same degree, which is not less than that of any vertex in the cliques.

To achieve this, we can augment G with two cliques on n auxiliary vertices each. Next, we connect each original vertex in G of degree q to we equally distribute the new connections in a cyclic fashion so that each vertex of the two cliques receives at most $\lceil (2n^2 - 2m)/2n \rceil$ connections to the original vertices. Let G^* be the resulting graph on $3n$ vertices. Note that all original vertices have degree $2n$ and all vertices in the n-cliques have degree at most $2n$ in G^*. It is clear that if the vertices of G can be partitioned into l cliques then the vertices of G^* can be partitioned into at most $l + 2$ cliques. Conversely, if the vertices of G^* can be partitioned into l cliques then the vertices of G can also be trivially partitioned into at most l cliques. Putting everything together, we obtain the following technical lemma.

Lemma 7. *For any $\epsilon > 0$, the clique partition problem restricted to quasi-regular graphs cannot be approximated (in terms of the number of cliques) within $n^{1/7-\epsilon}$ unless P=NP.*

By a reduction from the clique partition problem for quasi-regular graphs to HDC, we obtain the following result.

Theorem 8. *For any $\epsilon > 0$, the problem of finding a partition of a set of k binary strings of length $O(k^2)$ into at most $pk^{1/7-\epsilon}$ disjoint clusters such that each cluster has Hamming diameter not exceeding the p-diameter is NP-hard.*

Proof. Consider an instance of the restricted clique partition problem consisting of a quasi-regular graph G on k vertices and m edges, and a natural number p. Enumerate the edges of G. For each vertex v of G, form a string $s(v)$ of length m such that there is a 1 on the ith position in $s(v)$ iff the ith edge of G is incident to v. Let d be the maximum vertex degree of G. It follows that each vertex in G outside the two distinguished cliques has degree d. Note that for any pair of vertices v_1, v_2 in G of degree d, the Hamming distance between $s(v_1)$ and $s(v_2)$ is $2d - 2$ if they are adjacent, otherwise it is $2d$. Also, for any pair of vertices v_1, v_2 in the same distinguished clique of G, the Hamming distance between $s(v_1)$ and $s(v_2)$ is at most $2d - 2$. Therefore, any clique p-partition of G yields a p-clustering of the resulting strings of maximum Hamming diameter less than or equal to $2d - 2$. Conversely, any q-clustering of the resulting strings of maximum Hamming diameter less than or equal to $2d - 2$ trivially yields a clique $(q + 2)$-partition of G. Hence, by Lemma 7 we obtain our result. □

As for the corresponding problem for HRC (i.e., producing a larger set of approximative centers such that each input string is within the p-radius from at least one of the centers), we doubt whether it is equally hard to approximate.

At least, if we weaken the requirement of being within the p-radius by a multiplicative factor of $1 + \epsilon$, then this problem admits a logarithmic approximation in polynomial time, as it is shown at the end of the next section.

4 Approximation Algorithms for HRC and HDC

In this section, we first observe how an approximation factor of two for HRC and HDC can be achieved. Next, we provide an approximation scheme for HRC running in polynomial time when $p = O(1)$ and $\varrho = O(\log(k + n))$. Finally, we give a relaxed type of arbitrarily close approximation of ϱ due to a moderate increase in the number of clusters which runs in polynomial time whenever $\varrho = O(\log(k + n))$.

4.1 A 2-Approximation Algorithm for HRC and HDC

To obtain an approximation factor of two, we adapt Gonzalez' farthest-point clustering algorithm [6] to HRC and HDC respectively as follows:

Algorithm A

STEP 1: Set P^* to $\{\alpha_i\}$, where α_i is an arbitrary string in S.

STEP 2: For $l = 2, .., p$: augment P^* by a string in S that maximizes the minimum distance to P^*, i.e., that is as far away as possible from the strings already in P^*.

STEP 3 (HRC): Return P^*.

STEP 3 (HDC): Assign each string in S to a closest member in P^* and return the resulting clusters. □

The Hamming distance obeys the triangle inequality ([11], p. 424). Therefore, by the proof of Theorem 8.14 in [8], Algorithm A yields an approximative solution to either HRC or HDC that is always within a factor of two of the optimum. We can implement this algorithm by updating the Hamming distance of each string outside P^* to the nearest string in P^* after each augmentation of P^*. To update and then compute a string in S furthermost from P^* takes $O(kn)$ time in each iteration. Hence, we obtain the following theorem.

Theorem 9. *An approximative solution to either HRC or HDC that is always within a factor of two of the optimum can be found in $O(pkn)$ time.*

4.2 An Approximation Scheme for HRC

In this subsection we present a $2^{O(p\varrho/\varepsilon)}k^{O(p/\varepsilon)}n^2$-time $(1 + \varepsilon)$-approximation algorithm for HRC. Our scheme is partly based on the idea used in the PTAS for 1-HRC in [13].

Algorithm B

STEP 1: Set $\mathcal{C}$ to an empty subset of $\mathbb{Z}_2^n$. For each subset R of S having exactly r strings, compute the set Q consisting of all positions m, $1 \leq m \leq n$, on which all strings in R contain the same symbol. Set P to $\{1, 2, .., n\} \setminus Q$. For every possible $f : P \to \{0,\ 1\}$, let q_f be the string in $\mathbb{Z}_2^n$ which agrees with the strings in R on the positions in Q and contains $f(j)$ in each position $j \in P$. Augment $\mathcal{C}$ by q_f.

STEP 2: Let $\mathcal{C}^p$ be the family of all subsets of the set $\mathcal{C}$ of size p. Test all sets in $\mathcal{C}^p$ and return the $P^* \in \mathcal{C}^p$ that minimizes $\max\limits_{1 \leq i \leq k} \min\limits_{c \in P^*} d_H(\alpha_i, c)$. □

The next lemma can be proved analogously as Lemma 11 in [13] (the key lemma for the PTAS for the Hamming radius 1-clustering problem) is proved in case of a logarithmic or smaller sized radius.

Lemma 10. *For any subset U of S, there is a c in $\mathcal{C}$ such that*

$$\max_{\alpha \in U} d_H(\alpha, c) \leq (1 + \frac{1}{2r-1}) \min_{\beta \in \mathbb{Z}_2^n} \max_{\alpha \in U} d_H(\alpha, \beta)$$

Theorem 11. *Algorithm B constructs a p-center with the approximation factor $1 + \frac{1}{2r-1}$ in $O(2^{pr\varrho+1} k^{pr+1} n^2)$ time.*

Proof. To prove the correctness and the approximation factor of Algorithm B, consider an optimal p-center for S, say $\{\beta_1, .., \beta_p\}$. Partition S into subsets U_1 through U_p such that for $1 \leq j \leq p$ and $\alpha \in U_j$, β_j has minimum Hamming distance to α among $\beta_1, .., \beta_p$. By Lemma 10, the set $\mathcal{C}^p$ constructed in STEP 2 contains $\{\beta_1^*, .., \beta_p^*\}$ such that for $1 \leq j \leq p$ and any $\alpha \in U_j$, the Hamming distance between α and β_j^* is at most $1 + \frac{1}{2r-1}$ times the radius of U_j. Thus, Algorithm B yields a solution within $1 + \frac{1}{2r-1}$ of the optimum.

To derive the upper bound on the running time of Algorithm B, first observe that each of the sets P has size at most $r\varrho$ and that a string q_f can be constructed in $O(nr)$ time. Hence, the size of the set $\mathcal{C}$ doesn't exceed $2^{r\varrho} k^r$, and $\mathcal{C}$ can be constructed in $O(r 2^{r\varrho} k^r n)$ time. Consequently, $\mathcal{C}^p$ is of size at most $k^{rp} 2^{pr\varrho}$ and its construction from $\mathcal{C}$ takes $O(2^{pr\varrho} k^{pr} n)$ time. All that remains is to note that the test of each p-tuple in $\mathcal{C}^p$ can be performed in $O(kn)$ time. □

Note that the running time of Algorithm B is polynomial in n and k as long as p is a constant and $\varrho = O(\log(k + n))$.

Corollary 12. *Algorithm B yields a polynomial-time approximation scheme for the Hamming radius $O(1)$-clustering problem restricted to instances with the p-radius in $O(\log(k + n))$.*

4.3 A Relaxed Type of Approximation for HRC

In this subsection, we consider twofold approximation for HRC allowing for producing more than p approximative centers and slightly exceeding the p-radius.

For each c in $\mathcal{C}$ (see Algorithm B), let $S(c)$ be the set of all strings in S within distance $(1 + \frac{1}{2r-1})\varrho$ of c. By Lemma 10, there is a set consisting of p such sets, covering all of S. If ϱ is known, we run the classical greedy heuristic for minimum set cover (see [8]) on the instance $(S, \{S(c) \mid c \in \mathcal{C}\})$ to find a set of $O(p \log k)$ sets covering S. Otherwise, we perform a binary search for the smallest possible value of $\varrho \in \{0, 1, .., n\}$ in the definition of the sets $S(c)$ by running the aforementioned heuristic $O(\log n)$ times and each time testing whether or not the resulting cover of S has size $O(p \log k)$. Recall that $|\mathcal{C}| \leq 2^r \varrho k^r$ and that $\mathcal{C}$ can be constructed in $O(r2^r \varrho k^r n)$ time. The instance of set cover corresponding to a given value of ϱ can be constructed in $O(|\mathcal{C}|kn)$ time; the greedy heuristic can be implemented to run in $O(|\mathcal{C}|k^2)$ time. By choosing r so that $\frac{1+\varepsilon}{2\varepsilon} < r < \frac{2}{\varepsilon}$, we obtain the following result.

Theorem 13. *For any constant $0 < \varepsilon < 1$, we can construct a set L of $O(p \log k)$ strings of length n in $O((\frac{n}{\varepsilon} + kn \log n + k^2 \log n)(2\varrho k)^{2/\varepsilon})$ time such that for each of the k strings in S there is at least one string in L within distance $(1 + \varepsilon)$ of the p-radius.*

The time bound in Theorem 13 is polynomial in n and k as long as $\varrho = O(\log(k + n))$.

5 Conclusions

We have shown not only that two is the best approximation factor for HRC and HDC achievable in polynomial time unless P=NP, but also that it is possible to provide exact solutions or much better approximation solutions to HRC or HDC in several special or relaxed cases. It seems that there are plenty of interesting open problems in the latter direction. For example, is it possible to design very close and efficient approximation algorithms for protein data (see Section 1.1) taking into account the specific distribution of the input?

References

1. M. Bellare, O. Goldreich, and M. Sudan. Free Bits, PCPs, and Non-Approximability – Towards Tight Results. *SIAM Journal on Computing* 27(3), 1998, pp. 804–915.
2. M. Chrobak and T.H. Payne. A linear-time algorithm for drawing a planar graph on a grid. *Information Processing Letters* 54, 1995, pp. 241–246.
3. T. Feder and D. Greene. Optimal Algorithms for Approximate Clustering. *Proceedings of the 20th Annual ACM Symposium on Theory of Computing* (STOC'88), 1988, pp. 434–444.
4. M. Frances and A. Litman. On Covering Problems of Codes. *Theory of Computing Systems* 30, 1997, pp. 113–119.

5. L. Gąsieniec, J. Jansson, and A. Lingas. Efficient Approximation Algorithms for the Hamming Center Problem. *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA'99), 1999, pp. S905–S906.
6. T. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science* 38, 1985, pp. 293–306.
7. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
8. D.S. Hochbaum (editor). *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, Boston, 1997.
9. D.S. Hochbaum and D.B. Shmoys. A best possible heuristic for the k-center problem. *Mathematics of Operational Research* 10(2), 1985, pp. 180–184.
10. D.S. Hochbaum and D.B. Shmoys. A Unified Approach to Approximation Algorithms for Bottleneck Problems. *Journal of the Association for Computing Machinery* 33(3), 1986, pp. 533-550.
11. B. Kolman, R. Busby, and S. Ross. *Discrete Mathematical Structures* [3rd ed.]. Prentice Hall, New Jersey, 1996.
12. J.K. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang. Distinguishing String Selection Problems. *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA'99), 1999, pp. 633–642.
13. M. Li, B. Ma, and L. Wang, Finding Similar Regions in Many Strings. *Proceedings of the 31st Annual ACM Symposium on Theory of Computing* (STOC'99), 1999, pp. 473–482.
14. C. Papadimitriou. On the Complexity of Integer Programming. *Journal of the ACM* 28(4), 1981, pp. 765–768.
15. S. Vishwanathan. An $O(\log^* n)$ Approximation Algorithm for the Asymmetric p-Center Problem. *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA'96), 1996, pp. 1–5.

Approximating the Maximum Isomorphic Agreement Subtree Is Hard

Paola Bonizzoni, Gianluca Della Vedova, and Giancarlo Mauri

Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano - Bicocca
via Bicocca degli Arcimboldi 8, 20126 Milano (Italy)
{bonizzoni,dellavedova,mauri}@disco.unimib.it

Abstract. The Maximum Isomorphic Agreement Subtree (MIT) problem is one of the simplest versions of the Maximum Interval Weight Agreement Subtree method (MIWT) which is used to compare phylogenies. More precisely MIT allows to provide a subset of the species such that the exact distances between species in such subset is preserved among all evolutionary trees considered. In this paper, the approximation complexity of the MIT problem is investigated, showing that it cannot be approximated in polynomial time within factor $\log^{\delta} n$ for any $\delta > 0$ unless $\mathbf{NP} \subseteq \mathbf{DTIME}(2^{polylog\, n})$ for instances containing three trees. Moreover, we show that such result can be strengthened whenever instances of the MIT problem can contain an arbitrary number of trees, since MIT shares the same approximation lower bound of MAX CLIQUE.

1 Introduction

Evolutionary trees are unordered trees where each leaf is labeled by a distinct element in a set S of species and where all internal nodes have degree at least three. Evolutionary trees are frequently used by biologists to represent classifications of species, more precisely extant species label the leaves and edges are weighted with the estimated distance (i.e. temporal) between the two species represented by the endpoints of such edge. A number of methods to infer evolutionary trees have been proposed, moreover it is rather common to study different biological sequences or different sites of DNA, consequently various trees for the same set of species can be obtained. This fact motivates the compelling need to compare different trees, in order to extract a common history. The Maximum Agreement Subtree method is a basic approach that allows to reconciliate different evolutionary trees over the same set of species: it computes a subset of the extant species about which all trees are confident or "agree". A general way to define an agreement subtree from a set $T_1, \cdots, T_k$ of S-labeled trees has been formalized in [1]. This method assumes that each edge is labeled by an interval weight (a range of time to measure the duration of the evolution process) and looks for a subset S' of the extant species S such that:

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 119–128, 2000.

- each edge of the subtree induced in each tree of the given trees is labeled by a value belonging to the given interval,
- for each pair of extant species in S', the total distance between them is the same in all trees.

The problem stated above is called Maximum Interval Weight Agreement Subtree (MIWT), and is a very general formulation of the problem of comparing phylogenies. In order to obtain more efficient solutions, some restrictions have been introduced to MIWT. A first natural restriction requires that each interval contains one distinct value; such problem is called Maximum Weight Agreement Subtree (MWT). A different restriction of MIWT is the one where an agreement subtree is homeomorphic to a subtree of each tree in the instance, since it is equivalent to require all intervals to be of the form $[1, n-1]$, where n is the number of extant species considered. This problem is called Maximum Homeomorphic Agreement Subtree (MHW). Note that this problem is sometimes referred to as Maximum Agreement Subtree and is abbreviated by (MAST). A third restriction of MIWT is the one where all intervals are of the form $[1, 1]$, and is called Maximum Isomorphic Agreement Subtree (MIT), as all subtrees induced by a feasible solution must be isomorphic. The MIT problem is also a restricted case of the maximum isomorphic subgraph problem, investigated in [11]. Since MIT and MHT are the two more restricted problems among the ones we have mentioned, most of the efforts of developing efficient algorithms have been concentrated on them.

Efficient algorithms for the MHT problem for instances of two trees have been widely investigated in literature. While some heuristics have been known [7,14], the first polynomial time algorithm has been described only in 1993 by Steel and Warnow [17]. Afterwards successive improvements have appeared in literature [4,12,15]. To our knowledge the most efficient algorithms for the problem are due to Farach and Thorup which developed a $O(n^{3/2} \log n)$ algorithm for rooted trees of bounded degree [5,6], to Cole and Hariharan [3,2] for the case of rooted trees of unbounded degree, which gave a $O(n \log n)$ algorithm, and to Kao, Lam, Przytycka, Sung and Ting [13] which described a technique allowing to match the time complexity of the two previously cited algorithms also in the case of unrooted trees. The problems MHT and MIT over a set of trees, where at least one of the trees has bounded degree, can be solved in polynomial time [1], even though the time complexity is exponential in the bound for the degree. Moreover both problems are NP-hard for instances containing three trees of unbounded degree, hence it is necessary to study the possibility of designing polynomial time approximation algorithms. The approximation complexity of the MHT problem has been deeply investigated in [9], where some strong negative results have been obtained. Since the MIT is a simplified version of the MIWT, it seems natural to investigate if the negative results for MHT hold also for MIT or if such problem is easier to approximate. In our paper we show that the negative results of [9] hold also for the MIT problem, as a consequence of a nontrivial application of the self-improvement technique, consequently the search for polynomial time approximation algorithms achieving a constant error

ratio is NP-hard even for instances consisting of only three trees. Moreover we have strengthened such negative results in the case of instances containing an arbitrary number of trees, as we show that MIT shares the same inapproximability properties of MAX CLIQUE [8], implying that there cannot exist a polynomial time $n^{1-\epsilon}$ ratio approximation algorithm for each $\epsilon > 0$.

2 Preliminaries

Let $S = \{s_1, \ldots, s_n\}$ be a set of labels. An S-labeled tree has n leaves, each one labeled with a distinct element of S, since each label identifies unambiguously a leaf of the tree, in the following of the paper we will write a label x meaning the leaf of the tree with label x. The Maximum Isomorphic Agreement Subtree Problem (shortly MIT) is defined formally as follows:
Instance: a set $\mathcal{T} = \{T_1, \ldots, T_m\}$ of S-labeled trees.
Solution: an S^*-labeled tree T^*, with $S^* \subseteq S$, such that T^* is isomorphic to a subgraph of all trees in $\mathcal{T}$.
Measure: $|S^*|$, to be maximized.

All trees we will deal with in this paper are rooted, that is we distinguish a special vertex of the tree T and we call such vertex *root*, denoted by $r(T)$. All results presented in the paper are referred to rooted tree, but can be generalized to the unrooted case.

Let T be a tree and let a, b be two nodes of T, then we will denote by $d_T(a, b)$ the distance between a and b in T, that is the number of edges in the unique simple path from a to b in T. Let T be a rooted tree, and let t be a node of T, then the depth of t in T is the distance of t from the root of T. The depth of a tree T, denoted by $\text{depth}(T)$, is the maximum among the depths of its nodes. Given two leaves a, b of T we define the *least common ancestor*, of a and b in T, denoted by $\text{lca}_T(a, b)$, as the maximum depth node of T which is ancestor of both a and b.

It is immediate to note that the NP-completeness proof given by Amir and Keselman in [1] is an L-reduction, as pointed out in [9] for the MHT problem. Similarly it is possible to prove that MIT is **MAX SNP**-hard, that is there is no polynomial time approximation scheme for it, unless $\mathbf{P} = \mathbf{NP}$.

Anyway, differently from [9], to prove our inapproximability results such **MAX SNP**-hardness proof is not sufficient, but we have to deal with a restricted version of the problem; more precisely we consider only instances consisting of trees having leaves all at the same depth in every tree. Formally $d_{T_i}(a, r(T_i)) = d_{T_j}(b, r(T_j))$ for all $a, b \in S$ and every pair of trees T_i, T_j in the instance. We will say that trees in such instances are *restricted*. This new problem will be called R-MIT. Clearly all inapproximability results for that problem hold also for MIT.

The following Lemma, proved in [16], characterizes all feasible solutions of each instance of R-MIT.

Lemma 2.1. *Let $\mathcal{T}$ be a set of S-labeled trees, and let $S^* \subseteq S$. Then there exists a S^*-labeled tree T^* that is isomorphic to a subgraph of each tree in $\mathcal{T}$ iff for each pair of labels $a, b \in S^*$, a and b have the same distance in all trees in $\mathcal{T}$.*

As a consequence we can identify a feasible solution of an instance of MIT as a subset of its label set. The following property of trees, whose straightforward proof is omitted, will be used in the following of the paper.

Proposition 2.2. *Let a, b be two leaves of a S-labeled tree with root r. Then* $d_T(a, b) = d_T(a, r) + d_T(b, r) - 2\, d_T(r, \mathrm{lca}_T(a, b))$.

3 R-MIT Is MAX SNP-Hard

In this section we are going to prove that the R-MIT problem is **MAX SNP**-hard. This results is necessary to prove that MIT is hard to approximate even on instances consisting of only three trees.

The problem used in the L-reduction to R-MIT is the Threedimensional Bounded Matching (shortly 3DM-B). An instance of 3DM-B consists of three pairwise disjoint sets $< X_1, X_2, X_3 >$ and a set M of triples where $M \subseteq X_1 \times X_2 \times X_3$ and every element in $X_1 \cup X_2 \cup X_3$ occurs in at least one and at most B triples of M. The goal is to find a maximum cardinality subset M_1 of M such that no two triples in M_1 agree in any coordinate. The general 3DM-B problem is **MAX SNP**-hard [10].

Let $\mathcal{M} =< X_1, X_2, X_3, M >$ be an instance of the 3DM-B problem, with $M \subseteq X_1 \times X_2 \times X_3$, $X_i = \{x_{i,1}, x_{i,2}, \ldots x_{i,|X_i|}\}$. Then we will associate to $\mathcal{M}$ an instance $< T_1, T_2, T_3 >$ of MIT. Each tree T_i consists of the following nodes and edges: a root labeled r_i, a node connected to the root for each element of X_i, and finally for each element $x_{i,j}$ there is a node for each triple in M containing $x_{i,j}$ and such node is adjacent to the node associated to $x_{i,j}$. Then each tree T_i is M-labeled.

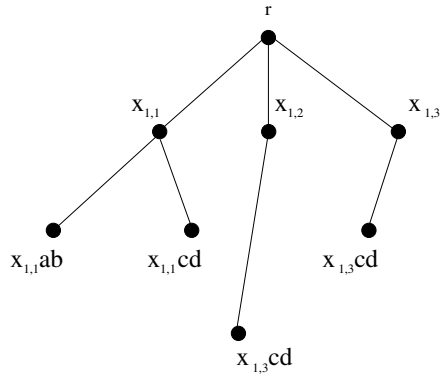


Fig. 1. Example of instance of R-MIT associated to an instance of 3DM-B

Since the distance from each node to the root is 2 in all trees of the instance of MIT associated to an instance of 3DM-B, such set of trees is an instance of R-MIT. The following Lemma is an immediate consequence of such fact.

Lemma 3.1. *Let* $\mathcal{M} =< X_1, X_2, X_3, M >$ *be an instance of* 3DM-B *and let* $< T_1, T_2, T_3 >$ *be the associated instance of* MIT. *Given a tree* T_i *with* $1 \leq i \leq 3$, *and given two distinct leaves* s, t *of* T_i, *then the distance of* s *and* t *in* T_i *is* 2 *or* 4.

Note that the distance of two leaves s and t in a tree T_i is 2 if and only if s and t are labeled by triples of M that share the same element in the set X_i.

Lemma 3.2. *Let* $\mathcal{M} =< X_1, X_2, X_3, M >$ *be an instance of* 3DM-B, *let* $< T_1, T_2, T_3 >$ *be the associated instance of* R-MIT *and let* $S \subseteq M$. *Then* S *is a feasible solution of* $< T_1, T_2, T_3 >$ *iff each pair* s, t *of distinct triples in* S *has distance* 4 *in all trees* T_i.

Proof. By Lemma 2.1 S is a feasible solution iff each distinct pair s, t of triples in S have the same distance in all trees T_i, that, by Lemma 3.1 is either 2 or 4. Assume to the contrary that there exists a pair s, t that has distance 2 in all trees. Then by construction s is equal to t, contradicting the fact the all sets in M are distinct. The other direction follows immediately by Lemma 2.1.

Theorem 3.3. *The reduction from* 3DM-B *to* R-MIT *is an L-reduction.*

Proof. Follows from Lemmas 3.1, 3.2.

4 Product of Trees

The inapproximability result over instances of three trees is obtained by means of the *self-improvement* technique. In [9] such technique has been exploited to prove a similar result for the MHT problem. Such technique requires a careful definition of product between instances of the problem, defined as follows:

Definition 4.1. *Let* T_1 *be a* S_1*-labeled tree, let* T_2 *be a* S_2*-labeled tree and let* s *be a leaf of* T_1, *then the tree* $T_{2,s}$ *is the tree obtained from* T_2 *relabeling each leaf* s_2 *with the sequence* ss_2. *Then the product* $T_1 \cdot T_2$ *is the tree obtained from* T_1 *replacing each leaf* s *with the tree* $T_{2 \cdot s}$.

Let T be a S-labeled tree, then $T^2 = T \cdot T$ and $T^i = T^{i-1} \cdot T$, $i > 2$. Note that the label of a leaf of the tree T^k is a string $s_1 \ldots s_k$ of k symbols over the alphabet S.

An immediate property of the product of trees is stated below:

Proposition 4.1. *Let* T_1, T_2 *be two restricted trees. Then* $T_1 \cdot T_2$ *is also a restricted tree.*

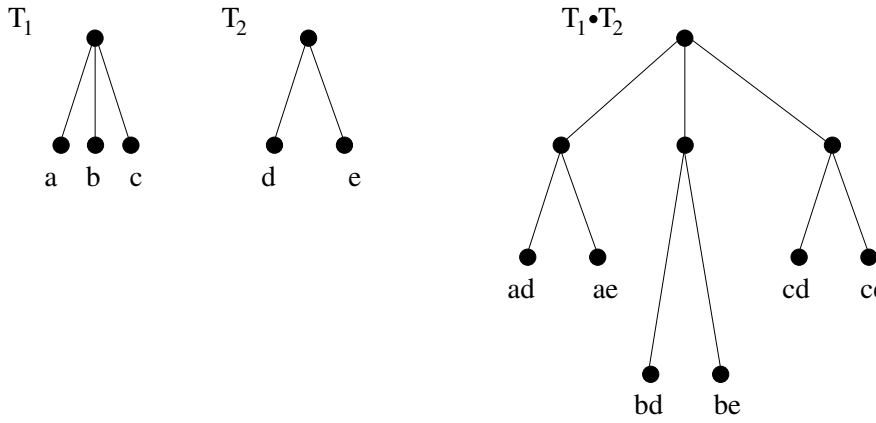


Fig. 2. Product of trees

The following Lemma points out the motivation for our definition of product.

Lemma 4.2. *Let T_1, T_2 be two restricted S-labeled trees, let a, b be two labels in S and let α, β be two strings of $k-1$ symbols of S. Then $d_{T_1^k}(\alpha a, \beta b) = d_{T_2^k}(\alpha a, \beta b)$ iff $d_{T_1}(a,b) = d_{T_2}(a,b)$ and $d_{T_1^{k-1}}(\alpha, \beta) = d_{T_2^{k-1}}(\alpha, \beta)$*

Proof. Please note that Prop. 2.2, together with the fact that T_1 and T_2 are restricted trees (that is in T_1 and T_2 all leaves have the same depth), implies that it is sufficient to prove that $d_{T_1^k}(\mathrm{lca}_{T_1^k}(\alpha a, \beta b), r(T_1^k)) = d_{T_2^k}(\mathrm{lca}_{T_2^k}(\alpha a, \beta b), r(T_2^k))$ iff $d_{T_1}(\mathrm{lca}_{T_1}(a,b), r(T_1)) = d_{T_2}(\mathrm{lca}_{T_2}(a,b), r(T_2))$ and $d_{T_1^{k-1}}(\mathrm{lca}_{T_1^{k-1}}(\alpha, \beta), r(T_1^{k-1})) = d_{T_2^{k-1}}(\mathrm{lca}_{T_2^{k-1}}(\alpha, \beta), r(T_2^{k-1}))$.

Initially let us consider the case $\alpha = \beta$, then, by definition of product, $d_{T_1^k}(\mathrm{lca}_{T_1^k}(\alpha a, \beta b), r(T_1^k)) = d_{T_{1\cdot\alpha}}(\mathrm{lca}_{T_{1\cdot\alpha}}(\alpha a, \alpha b), r(T_{1\cdot\alpha})) + \mathrm{depth}(T_1^{k-1})$, $d_{T_2^k}(\mathrm{lca}_{T_2^k}(\alpha a, \beta b), r(T_2^k)) = d_{T_{2\cdot\alpha}}(\mathrm{lca}_{T_{2\cdot\alpha}}(\alpha a, \alpha b), r(T_{2\cdot\alpha}))+ \mathrm{depth}(T_2^{k-1})$, hence $d_{T_1^k}(\mathrm{lca}_{T_1^k}(\alpha a, \beta b), r(T_1^k)) = d_{T_2^k}(\mathrm{lca}_{T_2^k}(\alpha a, \beta b), r(T_2^k))$ if and only if $d_{T_1}(\mathrm{lca}_{T_1}(a,b), r(T_1)) = d_{T_2}(\mathrm{lca}_{T_2}(a,b), r(T_2))$. Assume now that $\alpha \neq \beta$, then $\mathrm{lca}_{T_1^k}(\alpha a, \beta b) = \mathrm{lca}_{T_1^{k-1}}(\alpha, \beta)$ and $\mathrm{lca}_{T_2^k}(\alpha a, \beta b) = \mathrm{lca}_{T_2^{k-1}}(\alpha, \beta)$. Consequently $d_{T_1^k}(\mathrm{lca}_{T_1^k}(\alpha a, \beta b), r(T_1^k)) = d_{T_2^k}(\mathrm{lca}_{T_2^k}(\alpha a, \beta b), r(T_2^k))$ if and only if $d_{T_1^{k-1}}(\mathrm{lca}_{T_1^{k-1}}(\alpha, \beta), r(T_1^{k-1})) = d_{T_2^{k-1}}(\mathrm{lca}_{T_2^{k-1}}(\alpha, \beta), r(T_2^{k-1}))$. This suffices to prove the Lemma.

The following lemma relates a feasible solution of $< T_1, T_2, T_3 >$ with a feasible solution of $< T_1^k, T_2^k, T_3^k >$.

Lemma 4.3. *Let $< T_1, T_2, T_3 >$ be an instance of* R-MIT*, and let F be a feasible solution of such instance. Then it is possible to compute in polynomial time a solution of $< T_1^k, T_2^k, T_3^k >$ whose cost is $\mathrm{cost}(F)^k$.*

Proof. Let F_k be the set of strings of labels $\{f_1 \cdots f_k : f_i \in F,\ 1 \leq i \leq k\}$. We will prove that for each pair of strings of labels $f_{\alpha_1} \cdots f_{\alpha_k}$, $f_{\beta_1} \cdots f_{\beta_k}$ in F_k, their distance is the same in all trees T_1^k, T_2^k, T_3^k. Let k_1 be the minimum integer such that F_{k_1} is not a solution of $< T_1^{k_1}, T_2^{k_1}, T_3^{k_1} >$, w.l.o.g. we can assume that $f_{\alpha_1} \cdots f_{\alpha_k}$, $f_{\beta_1} \cdots f_{\beta_k}$ do not have the same distance in $T_1^{k_1}$ and in $T_2^{k_1}$. Note that $k_1 > 1$, as $d_{T_1}(f_{\alpha_i}, f_{\beta_i}) = d_{T_2}(f_{\alpha_i}, f_{\beta_i}) = d_{T_3}(f_{\alpha_i}, f_{\beta_i})$ for all i, since all $f_{\alpha_i}, f_{\beta_i}$ are in S. But Lemma 4.2 implies that $d_{T_1}(f_{\alpha_k}, f_{\beta_k}) \neq d_{T_2}(f_{\alpha_k}, f_{\beta_k})$ or $d_{T_1^{k_1-1}}(f_{\alpha_1} \cdots f_{\alpha_{k-1}}, f_{\beta_1} \cdots f_{\beta_{k_1-1}}) \neq d_{T_2^{k_1-1}}(f_{\alpha_1} \cdots f_{\alpha_{k-1}},\ f_{\beta_1} \cdots f_{\beta_{k_1-1}})$, contradicting the minimality of k_1.

Theorem 4.4. *Let $< T_1^k, T_2^k, T_3^k >$ be an instance of* R-MIT *and let S_k be a feasible solution of $< T_1^k, T_2^k, T_3^k >$, then it is possible to compute in polynomial time a feasible solution S_1 of $< T_1, T_2, T_3 >$ such that $cost(S_k) \leq cost(S_1)^k$.*

Proof. Let $S_k = \{f_{\alpha_1}, \ldots, f_{\alpha_k}\}$ be a feasible solution of $< T_1^k, T_2^k, T_3^k >$. By applying Lemma 4.2 iteratively we can obtain k feasible solutions F_i of $< T_1, T_2, T_3 >$, where each solution F_i contains exactly the symbols f_{α_i} of S that are in the i-th position of a string in S_k. Let F^* be the largest of such F_i and let F_k^* be the set of strings $\{f_1 \ldots f_k : f_j \in F^*,\ 1 \leq j \leq k\}$. Just as in the proof of Lemma 4.3 it is possible to prove that F_k^* is a feasible solution of $< T_1^k, T_2^k, T_3^k >$. An immediate counting argument and the fact that F^* is the F_i of maximum cardinality imply that $|S_k| \leq |F_k^*| = |F^*|^k$.

We are now able to state our main results:

Theorem 4.5. *There does not exists a constant-ratio polynomial-time approximation algorithm for* R-MIT *unless* **NP**=**P**.

Proof. Assume to the contrary that there exists an ϵ-approximation polynomial-time algorithm for R-MIT. Then let $\alpha > 0$, and pose $k = \lceil \log_\alpha \epsilon \rceil$, consequently $\alpha^k \geq \epsilon$. Since there exists an ϵ-approximation algorithm for R-MIT let be $\mathrm{Apx}(< T_1, T_2, T_3 >)$ be the solution returned by such algorithm for the instance $< T_1, T_2, T_3 >$, while $\mathrm{Opt}(< T_1, T_2, T_3 >)$ denotes the optimum solution. Then, by Lemmas 4.3, 4.4,

$$\left(\frac{\mathrm{Opt}(< T_1, T_2, T_3 >)}{\mathrm{Apx}(< T_1, T_2, T_3 >)}\right)^k = \frac{\mathrm{Opt}(< T_1^k, T_2^k, T_3^k >)}{\mathrm{Apx}(< T_1^k, T_2^k, T_3^k >)} \leq \epsilon \leq \alpha^k$$

hence $(\frac{\mathrm{Opt}(<T_1,T_2,T_3>)}{\mathrm{Apx}(<T_1,T_2,T_3>)}) =\leq \alpha$. Note that computing $< T_1^k, T_2^k, T_3^k >$ from $< T_1, T_2, T_3 >$ can be done in $O(n^{\lceil \log_\alpha \epsilon \rceil})$ time, hence we have described a PTAS for R-MIT. By Theorem 3.3 **NP**=**P**.

Corollary 4.6. *There exists a constant $\delta > 0$ such that* R-MIT *cannot be approximated within factor $\log^\delta n$ in polynomial time, unless* **NP** $\subseteq$ **DTIME**$[2^{polylog\, n}]$.

Proof. Assume to the contrary that for all $\delta > 0$ there exists a $\log^{\delta} n$-approximation polynomial-time algorithm for R-MIT. Then let $\alpha > 0$, and pose $k = \lceil \log \log^{\delta} n \rceil$, consequently $e^k \geq \log^{\delta} n$. Just as in the proof of Theorem 4.5 we will denote with $\mathrm{Apx}(< T_1, T_2, T_3 >)$ the solution returned by the approximation algorithm for the instance $< T_1, T_2, T_3 >$, while $\mathrm{Opt}(< T_1, T_2, T_3 >)$ denotes the optimum solution. Then, by Lemmas 4.3, 4.4,

$$\left(\frac{\mathrm{Opt}(< T_1, T_2, T_3 >)}{\mathrm{Apx}(< T_1, T_2, T_3 >)}\right)^k = \frac{\mathrm{Opt}(< T_1^k, T_2^k, T_3^k >)}{\mathrm{Apx}(< T_1^k, T_2^k, T_3^k >)} \leq \log^{\delta} n$$

taking the logarithms of both sides

$$k \log \left(\frac{\mathrm{Opt}(< T_1, T_2, T_3 >)}{\mathrm{Apx}(< T_1, T_2, T_3 >)}\right) \leq \log(\log^{\delta} n)$$

Consequently

$$\lceil \log \log^{\delta} n \rceil \log \left(\frac{\mathrm{Opt}(< T_1, T_2, T_3 >)}{\mathrm{Apx}(< T_1, T_2, T_3 >)}\right) \leq \log(\log^{\delta} n)$$

implying that $\log(\frac{\mathrm{Opt}(<T_1,T_2,T_3>)}{\mathrm{Apx}(<T_1,T_2,T_3>)}) \leq 1$. Hence $\frac{\mathrm{Opt}(<T_1,T_2,T_3>)}{\mathrm{Apx}(<T_1,T_2,T_3>)} \leq e$. It is immediate to note that that computing $< T_1^k, T_2^k, T_3^k >$ from $< T_1, T_2, T_3 >$ can be done in $O(n^{\lceil \log \log^{\delta} n \rceil}) = 2^{polylog n}$ time. Thus the claim follows from Thm. 4.5.

5 Inapproximability over Unbounded Number of Trees

The inapproximability result presented in the previous section can be strengthened when instances are not required to contain exactly three trees, but can contain an arbitrary number of trees. This can be proved by a simple L-reduction from MAX CLIQUE. Since such reduction preserves the optimum and the cost of approximate solutions, MIT with unbounded number of tree inherits the same inapproximability results of MAX CLIQUE, that is it cannot be approximated within $n^{1-\epsilon}$ for each $\epsilon > 0$, unless $\mathbf{ZPP} = \mathbf{NP}$ [8]. An instance of MAX CLIQUE is an unoriented graph $G =< V, E >$, and a feasible solution is a *clique* of G, that is is a subset $C \subseteq V$ such that $(c_1, c_2) \in E$ for each pair c_1, c_2 of vertices in C. The goal is to maximize $|C|$.

The reduction is quite simple: let $G = (V, E)$ be a graph with $E \neq \emptyset$. The instance of MIT contains the V-labeled trees in the set $\{T_{edge}\} \cup \{T_{ij} : i, j \in V,\ (i, j) \notin E\}$, where T_{edge} has root r and each leaf v of T_{edge} has p_v as parent and p_v is a child of r. Each tree T_{ij} consists of a root r, a node p_{ij} that is the parent of both leaves v_i, v_j a node p_z for each $z \in V - \{v_i, v_j\}$ and each p_z is the parent of the leaf v_z. Moreover p_{ij} and all p_z with $z \in V - \{i, j\}$ are the children of the root.

The following Lemma points out the structure of all feasible solutions consider in our reduction.

Lemma 5.1. *Let $\mathcal{T}$ be the set of V-labeled trees associated to an instance $G =< V, E >$ of* MAX CLIQUE*, and let T be a feasible solution of* MIT$(\mathcal{T})$*. Let v_1, v_2 be two distinct leaves of T. Then the distance between v_1 and v_2 in T is four.*

Proof. Let v_1, v_2 be two distinct leaves of T. Since v_1 and v_2 are both in a feasible solution T of $\mathcal{T}$, by Lemma 2.1 their distance must be the same in all trees in $\mathcal{T}$. Since $d_{T_{edge}}(v_1, v_2) = 4$ then $d_T(v_1, v_2) = 4$.

We will show how to compute a feasible solution of MIT from a feasible solution of MAX CLIQUE and vice versa, so that the costs of the solution are the same.

Let $\mathcal{T}$ be the instance of MIT, associated to the instance $G =< V, E >$ of MAX CLIQUE, and let $V_1 \subset V$ be a feasible solution of $\mathcal{T}$. Please note that, by Lemmas 2.1 and 5.1 a subset $V_1 \subseteq V$ is a feasible solution of $\mathcal{T}$ iff $d_T(v_1, v_2) = 4$ for each pair of distinct elements $v_i, v_j \in V_1$ and each tree $T \in \mathcal{T}$. We will prove that V_1 is a clique of G. Assume to the contrary that V_1 is not a clique of G, that is there exist two vertices $v_i, v_j \in V_1$ such that $(v_i, v_j) \in E$. By construction in $\mathcal{T}$ there is the tree T_{ij}, and $d_{T_{ij}}(v_i, v_j) = 2$. Consequently by Lemma 5.1 v_i and v_j cannot both be in a feasible solution of $\mathcal{T}$. To compute a feasible solution of $\mathcal{T}$ from a clique of G is trivial, hence the following theorem follows:

Theorem 5.2. MIT *over an unbounded number of tree cannot be approximated within $n^{1-\epsilon}$ for each $\epsilon > 0$, unless* **ZPP** = **NP**.

6 Conclusions

The MIT problem is one of the simplest formulations of evolutionary trees comparison proposed in literature, while the most studied of such formulations is the MHT problem. In our paper we have shown that MIT shares the same inapproximability bounds of MHT whenever the instances are restricted to contain exactly 3 trees, while it inherits the same bounds of MAX CLIQUE when the instances are unrestricted.

Acknowledgments

We gratefully acknowledge the support of MURST grant "Bioinformatica e Ricerca genomica".

References

1. A. Amir and D. Keselman. Maximum agreement subtree in a set of evolutionary trees: Metrics and efficient algorithms. *SIAM Journal on Computing*, 26(6):1656–1669, 1997.
2. R. Cole, M. Farach, R. Hariharan, T. Przytycka, and M. Thorup. An O($n \log n$) algorithm for the maximum agreement subtree problem for binary trees. *SIAM Journal on Computing*, to appear.

3. R. Cole and R. Hariharan. An $O(n \log n)$ algorithm for the maximum agreement subtree problem for binary trees. In *Proc. of the 7thAnnual ACM-SIAM Symposium on Discrete Algorithms (SODA96)*, pages 323–332, 1996.
4. M. Farach, T. M. Przytycka, and M. Thorup. On the agreement of many trees. *Information Processing Letters*, 55(6):297–301, 1995.
5. M. Farach and M. Thorup. Fast comparison of evolutionary trees. *Information and Computation*, 123(1):29–37, 1995.
6. M. Farach and M. Thorup. Sparse dynamic programming for evolutionary-tree comparison. *SIAM Journal on Computing*, 26(1):210–230, 1997.
7. C. Finden and A. Gordon. Obtaining common pruned trees. *Journal of Classification*, 2:255–276, 1985.
8. J. Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, to appear.
9. J. Hein, T. Jiang, L. Wang, and K. Zhang. On the complexity of comparing evolutionary trees. In Z. Galil and E. Ukkonen, editors, *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching (CPM95)*, volume 937 of *LNCS*, pages 177–190. Springer-Verlag, 1995.
10. V. Kann. Maximum bounded 3-dimensional matching is MAX SNP-complete. *Information Processing Letters*, 37(1):27–35, 1991.
11. V. Kann. On the approximability of the maximum common subgraph problem. In *Proc. 9th Ann. Symp. on Theoretical Aspects of Comput. Sci. (STACS92)*, volume 577 of *LNCS*, pages 377–388, 1992.
12. M.-Y. Kao. Tree contractions and evolutionary trees. *SIAM Journal on Computing*, to appear.
13. M.-Y. Kao, T. W. Lam, T. M. Przytycka, W.-K. Sung, and H.-F. Ting. General techniques for comparing unrooted evolutionary trees. In *Proceedings of the 29th Symposium on the Theory of Computing (STOC97)*, pages 54–65, 1997.
14. E. Kubicka, G. Kubicki, and F. McMorris. An algorithm to find agreement subtrees. *Journal of Classification*, 12(1):91–99, 1995.
15. T. Lam, W. Sung, and H. Ting. Computing the unrooted maximum agreement subtree in subquadratic time. In *Proc. of the 5th Scandinavian Workshop on ALgorithms Theory*, LNCS, pages 124–135, 1996.
16. E. Smolenskii. *Jurnal Vicisl. Mat. i Matem. Fiz.*, 2(2):371–372, 1962.
17. M. Steel and T. Warnow. Kaikoura tree theorems: Computing the maximum agreement subtree. *Information Processing Letters*, 48(2):77–82, 1993.

A Faster and Unifying Algorithm for Comparing Trees

Ming-Yang Kao[1⋆], Tak-Wah Lam[2⋆⋆], Wing-Kin Sung[3], and Hing-Fung Ting[2]

[1] Department of Computer Science
Yale University, New Haven, CT 06520, USA
kao-ming-yang@cs.yale.edu
[2] Department of Computer Science and Information Systems
University of Hong Kong, Hong Kong
{twlam,hfting}@csis.hku.hk
[3] E-Business Technology Institute
University of Hong Kong, Hong Kong
wksung@eti.hku.hk

Abstract. A widely-used method for determining the similarity of two labeled trees is to compute a maximum agreement subtree of the two trees. Previous work on this similarity measure only concerns with the comparison of labeled trees of two special kinds, namely, uniformly labeled trees (i.e., trees with all their nodes labeled by the same symbol) and evolutionary trees (i.e., leaf-labeled trees with distinct symbols for distinct leaves). This paper presents an algorithm for comparing trees that are labeled in an arbitrary manner. In addition to the generalization, our algorithm is faster than the previous algorithms in many cases.

1 Introduction

A *labeled* tree is a rooted tree with an arbitrary subset of nodes being labeled with symbols. Labeled trees are used to model the relationship of objects in real-life systems. In recent years, many algorithms for comparing such trees have been developed for diverse application areas including biology [6,17,18,22], chemistry [26], linguistics [7,20], computer vision [16], pattern recognition [23,24], and structured text databases [14,15,19].

A widely-used measure of the similarity of two labeled trees is the notion of a maximum agreement subtree. A labeled tree R is said to be a *label-preserving homeomorphic* subtree of another labeled tree T if there exists a one-to-one mapping f from the nodes of R to the nodes of T such that for any nodes u, v, w of R, (1) u and $f(u)$ have the same label; and (2) w is the least common ancestor of u and v if and only if $f(w)$ is the least common ancestor of $f(u)$ and $f(v)$. Let T_1 and T_2 be two labeled trees. An *agreement* subtree of T_1 and T_2 is a labeled tree which is also a label-preserving homeomorphic subtree of the two trees. A *maximum* agreement subtree is one which maximizes the number of labeled nodes.

⋆ Research supported in part by NSF Grant CCR-9531028
⋆⋆ Research supported in part by Hong Kong RGC Grant HKU-7027/98E

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 129–142, 2000.

In some applications, each symbol z may be associated with a positive integer weight $\mu(z)$ to indicate the relative significance of each symbol. We generalize the definition of a *maximum* agreement subtree of T_1 and T_2 to be one which maximizes the total weight of labeled nodes; let $\text{MAST}(T_1, T_2)$ denote this maximum total weight. Let $n = |T_1| + |T_2|$, i.e., the number of nodes in T_1 and T_2. Let d be the maximum degree of T_1 and T_2. Let N be the maximum $\mu(z)$ over all the symbols z. Note that when $N = 1$, $\mu(z) = 1$ for all symbols z and the new definition of a maximum agreement subtree is the same as the original one.

In the literature, many algorithms for computing $\text{MAST}(T_1, T_2)$ have been developed. These algorithms focus on the special cases where $N = 1$ and T_1 and T_2 are (1) *evolutionary tree* [10], i.e., leaf-labeled trees with distinct symbols for distinct leaves or (2) uniformly labeled trees, i.e., trees with all their nodes unlabeled or labeled with the same symbol.

For evolutionary trees, Steel and Warnow [25] gave the first polynomial-time algorithm, which runs in $O(n^{4.5} \log n)$ time. Farach and Thorup [5] reduced the time complexity to $O(n^{1.5} \log n)$. Recently, Kao, et al. [13] further improved the time complexity to $O(n^{1.5})$ with a breakthrough for a long-standing open problem on maximum bipartite matchings. Faster algorithms for the case $d = O(1)$ have also been discovered recently. The algorithm of Farach, Przytycka and Thorup [4] runs in $O(\sqrt{d}n \log^3 n)$ time, and that of Kao [11] takes $O(nd^2 \log^2 n \log d)$ time. Cole and Hariharan [2] gave an $O(n \log n)$-time algorithm for the case where T_1 and T_2 are binary trees. Przytycka [21] removed the degree-2 restriction with an $O(\sqrt{d}n \log n)$-time algorithm.

For uniformly labeled trees T_1 and T_2, $\text{MAST}(T_1, T_2)$ requires longer time to compute. Chung [1] gave an algorithm to determine whether T_1 is a label-preserving homeomorphic subtree of T_2 using $O(n^{2.5})$ time. Their algorithm provides a tool for verifying whether an uniformly labeled tree T_3 is an agreement subtree of T_1 and T_2 in $O(m^{2.5})$ time, where $m = |T_1| + |T_2| + |T_3|$. Gupta and Nishimura [9] gave an algorithm which actually computes a maximum agreement subtree of T_1 and T_2 in $O(n^{2.5} \log n)$ time.

Instead of solving special cases, this paper gives an algorithm to compute $\text{MAST}(T_1, T_2)$ where T_1 and T_2 are without restrictions (i.e., labels are not restricted to leaves and may not be distinct). The generality of our algorithm does not mean a sacrifice on speed. Let $W_{T_1,T_2} = \sum_{u \in T_1} \sum_{v \in T_2} \delta(u, v)$ where $\delta(u, v) = 1$ if nodes u and v are labeled with the same symbol, and 0 otherwise. We will omit the subscripts of W_{T_1,T_2} when the context is clear. When $N \neq 1$, we show that $\text{MAST}(T_1, T_2)$ takes time $O(\sqrt{d}W \log n \log(nN))$. When $N = 1$, we reduce the running time to $O(\sqrt{d}W \log^2 \frac{n}{d})$. Thus, if T_1 and T_2 are uniformly labeled trees, then $W \leq n^2$ and the time complexity of our algorithm is $O(\sqrt{d}n^2 \log^2 \frac{n}{d})$, which is faster than the algorithm in [9] for any d. If T_1 and T_2 are evolutionary trees, then $W \leq n$ and the time complexity of our algorithm is $O(\sqrt{d}n \log^2 \frac{n}{d})$. This time complexity is better than the past results for $d \geq n/2^{O(\sqrt{\log n})}$. In particular, $\sqrt{d}n \log^2 \frac{n}{d} = O(n^{1.5})$ for any degree d.

Section 2 discusses basics. Section 3 details our algorithm which computes $\text{MAST}(T_1, T_2)$ in $O(\sqrt{d}W \log n \log(nN))$ time for $N \neq 1$ and in $O(\sqrt{d}W \log^2 \frac{n}{d})$ time for $N = 1$. Section 4 concludes the paper with open problems.

2 Preliminaries

2.1 Basic Concepts

Restricted Subtrees: For a rooted tree T and any node u of T, let T^u denote the subtree of T that is rooted at u. For any set L of symbols, the *restricted subtree* of T with respect to L, denoted by $T\|L$, is the subtree of T whose nodes are the nodes labeled with L and the least common ancestors of any two nodes labeled with L, and whose edges preserve the ancestor-descendant relationship of T. Note that $T\|L$ may contain nodes with labels outside L; see Figure 1 for an example. For any labeled tree Υ, let $T\|\Upsilon$ denote the restricted subtree of T with respect to the set of symbols used in Υ.

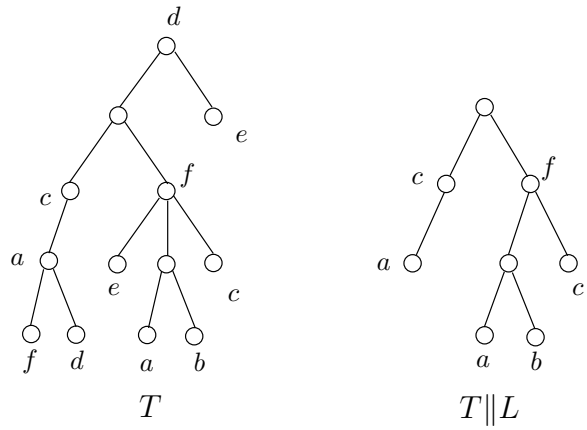


Fig. 1. The restricted subtree $T\|L$ with $L = \{a, b, c\}$. Note that $T\|L$ contains a label not in L.

Centroid Paths: A *centroid path decomposition* [2] of a rooted tree T is a partition of its nodes into disjoint paths as follows. For each internal node u in T, let $C(u)$ denote the set of children of u. Among the children of u, one of them is chosen to be the *heavy* child, denoted by $\text{hvy}(u)$, if the subtree of T rooted at $\text{hvy}(u)$ contains the largest number of nodes; the other children of u are the *side* children. The edge from u to its heavy child is a *heavy* edge. A *centroid path* is a maximal path formed by heavy edges; the *root* centroid path is the centroid path which contains the root of T. Let $\mathcal{D}(T)$ denote the set of the centroid paths of T. $\mathcal{D}(T)$ forms the desired partition and can be constructed in $O(|T|)$ time.

For each path P in $\mathcal{D}(T)$, let $r(P)$ denote the node in P which is the closest to the root of T. Let $\prec$ denote the ordering on $\mathcal{D}(T)$ where $P_1 \prec P_2$ if and only if $r(P_1)$ is a descendant of $r(P_2)$.

For each $P \in \mathcal{D}(T)$ and each node u in P, the subtree of T rooted at a side child of u is a *sidetree* of u as well as of P. Let $\text{SIDE}(P)$ denote the set of all the sidetrees of P. Note that the size of a sidetree of P is at most $|T^{r(P)}|/2$.

Fact 1. *Let P be the root centroid path of T_1.*

1. $W_{T_1,T_2} = W_{T_1,T_2\|T_1}$; *thus,* $W_{\Upsilon,T_2\|\Upsilon} = W_{\Upsilon,T_2}$ *for any sidetree* Υ *of* P.
2. $W_{T_1,T_2} = W_{P,T_2} + \sum_{\Upsilon\in\text{SIDE}(P)} W_{\Upsilon,T_2}$.

Intersecting Pairs: Consider $P \in \mathcal{D}(T_1)$ and $Q \in \mathcal{D}(T_2)$. For any nodes $u \in P$ and $v \in Q$ and any sidetrees Υ of u and Γ of v, the sidetree pair (Υ, Γ) of (P, Q) is *intersecting* if Υ and Γ have some common symbol. The sidetree-node pair (Υ, v) (and (u, Γ), respectively) of (P, Q) is *intersecting* if there exists a node in Υ (and Γ, respectively) which has the same label as v (and u, respectively). The *node* pair (u, v) of (P, Q) is *intersecting* if (1) u and v have the same label; or (2) there exist sidetrees Υ of u and Γ of v such that (Υ, Γ), (Υ, v), or (u, Γ) is intersecting.

Let ℓ_{PQ} be the total number of node pairs (u, v) of (P, Q) such that u, v have the same symbol plus the total number of intersecting sidetree pairs and sidetree-node pairs of (P, Q). Let $\mathcal{B}(u, v)$ be the union of

- $\{(\Upsilon, \Gamma), (\Upsilon, T_2^v), (T_1^u, \Gamma) \mid (\Upsilon, \Gamma)$ is intersecting and Υ and Γ are sidetrees of u and v, respectively $\}$:
- $\{(\Upsilon, T_2^v) \mid (\Upsilon, v)$ is intersecting and Υ is a sidetree of $u\}$;
- $\{(T_1^u, \Gamma) \mid (u, \Gamma)$ is intersecting and Γ is a sidetree of $v\}$.

Let $\mathcal{B}(P, Q) = \bigcup_{u\in P, v\in Q} \mathcal{B}(u, v)$. Note that $|\mathcal{B}(P, Q)| \le 3\ell_{PQ}$.

Maximum Weighted Matchings: Let $\text{MWM}(G)$ denote the maximum possible weight of any matching of a weighted bipartite graph $G = (X, Y, E)$.

Fact 2. (see [13]) *Let $s = \max\{|X|, |Y|\}$. Let w be the total weight of the edges in G. $\text{MWM}(G)$ can be computed in $O(\sqrt{s}w)$ time.*

Our subtree algorithm needs to find maximum weight matchings for a number of bipartite graphs. For many of these graphs, the size of the two vertex sets may differ a lot. We improve the above result to take advantage of such difference.

Lemma 1. *Let $t = \min\{|X|, |Y|\}$. Let w be the total weight of the edges in G. $\text{MWM}(G)$ can be computed in $O(\sqrt{t}w)$ time.*

Proof. To be given in the full paper.

This paper uses maximum weight matchings of the bipartite graphs G_{uv} and H_{uv} for any nodes $u \in T_1$ and $v \in T_2$ defined as follows:

- G_{uv} is the bipartite graph between $C(u)$ and $C(v)$ where each edge (x, y) has weight $\text{MAST}(T_1^x, T_2^y)$.

– H_{uv} is the bipartite graph between $C(u) - \{\text{hvy}(u)\}$ and $C(v) - \{\text{hvy}(v)\}$ where (x, y) is an edge of H_{uv} if and only if $\text{MAST}(T_1^x, T_2^y) > 0$ (i.e., (T_1^x, T_2^y) is an intersecting sidetree pair with T_1^x and T_2^y are sidetrees of u and v, respectively), and where the weight of each edge (x, y) is $\text{MAST}(T_1^x, T_2^y)$.

Lemma 2. *Consider any centroid paths P of T_1 and Q of T_2. Let $n_1 = |T_1^{r(P)}|$ and $n_2 = |T_2^{r(Q)}|$. Let $\mathcal{M}_{PQ}$ be the time required to compute $\text{MWM}(H_{uv})$ for all intersecting node pairs (u, v) of (P, Q). If $N \neq 1$, then*

$$\mathcal{M}_{PQ} = O\left(\min\left\{\sqrt{d}, \sqrt{n_1}, \sqrt{n_2}\right\} \ell_{PQ} \log(nN)\right);$$

if $N = 1$, then $\mathcal{M}_{PQ} = O\left(\min\left\{\sqrt{d}, \sqrt{n_1}, \sqrt{n_2}\right\} W_{T_1^{r(P)}, T_2^{r(Q)}}\right)$.

Proof. For $N \neq 1$, by Gabow-Tarjan matching algorithm [8], $\text{MWM}(H_{uv})$ can be computed in $O\left(\min\left\{\sqrt{d}, \sqrt{n_1}, \sqrt{n_2}\right\} \ell_{uv} \log(nN)\right)$ time where ℓ_{uv} is the number of intersecting sidetree pairs (Υ, Γ) with Υ and Γ being sidetrees of u and v, respectively. Thus, $\mathcal{M}_{PQ} =$

$$\begin{aligned} &O\left(\min\left\{\sqrt{d}, \sqrt{n_1}, \sqrt{n_2}\right\} \sum_{u \in P, v \in Q} \ell_{uv} \log(nN)\right) \\ &= O\left(\min\left\{\sqrt{d}, \sqrt{n_1}, \sqrt{n_2}\right\} \ell_{PQ} \log(nN)\right). \end{aligned}$$

For $N = 1$, $\text{MWM}(H_{uv})$ can be computed in $O\left(\min\left\{\sqrt{d}, \sqrt{n_1}, \sqrt{n_2}\right\} w_{uv}\right)$ time where $w_{uv} = \sum\{W_{\Upsilon,\Gamma} \mid \Upsilon \text{ is a sidetree of } u \text{ and } \Gamma \text{ is a sidetree of } v\}$ (by Lemma 1). Thus, $\mathcal{M}_{PQ} =$

$$O\left(\min\left\{\sqrt{d}, \sqrt{n_1}, \sqrt{n_2}\right\} \sum_{u \in P, v \in Q} w_{uv}\right) = O\left(\min\left\{\sqrt{d}, \sqrt{n_1}, \sqrt{n_2}\right\} W_{T_1^{r(P)}, T_2^{r(Q)}}\right).$$

2.2 Technical Lemmas

This section states two technical results, which are crucial to our algorithm for computing $\text{MAST}(T_1, T_2)$. It is based on the following formula, which is a straightforward generalization of the formula used in [5] for finding maximum agreement subtree of two evolutionary trees.

$$\text{MAST}(T_1^u, T_2^v) = \max \begin{cases} \max\{\text{MAST}(T_1^u, T_2^{c_2}) \mid c_2 \in C(v)\}; \\ \max\{\text{MAST}(T_1^{c_1}, T_2^v) \mid c_1 \in C(u)\}; \\ \text{MWM}(G_{uv}) \qquad \text{if } u \text{ and } v \text{ are unlabeled;} \\ \text{MWM}(G_{uv}) + \mu(z) \text{ if both } u \text{ and } v \text{ are labeled } z. \end{cases} \tag{1}$$

Based on Equation (1), $\text{MAST}(T_1, T_2)$ can be computed by bottom-up dynamic programming. However, to find $\text{MAST}(T_1^u, T_2^v)$ for all nodes u of T_1 and v of T_2,

the dynamic program needs to compute $\Theta(n^2)$ maximum weight matchings. To reduce the time on compute the maximum weight matchings, we make use of the following lemma, which implies that it suffices to focus on intersecting node pairs of centroid paths.

Lemma 3. *Let P and Q be the root centroid paths of T_1 and T_2, respectively. Suppose that we are given the values $\text{MAST}(\Upsilon, \Gamma)$ for all the pairs $(\Upsilon, \Gamma) \in \mathcal{B}(P, Q)$ and the values $\text{MWM}(H_{uv})$ for all intersecting node pairs (u, v) of (P, Q). Then, for any subset S of node pairs of (P, Q), we can compute $\text{MAST}(T_1^x, T_2^y)$ for all $(x, y) \in S$ in $O((\ell_{PQ} + |S|) \log n)$ time.*

Proof. See Appendix A.

Remark. Cole and Hariharan [2] and Przytycka [21] obtained this lemma for the case where T_1 and T_2 are evolutionary trees.

The next theorem combines Lemmas 2 and 3 and is important to our algorithm for computing $\text{MAST}(T_1, T_2)$.

Theorem 1. *Consider the root centroid path P of T_1 and any centroid path Q of T_2. Suppose that the values $\text{MAST}(\Upsilon, \Gamma)$ for all the pairs $(\Upsilon, \Gamma) \in \mathcal{B}(P, Q)$ are given. The following values can be computed in $O(\ell_{PQ} \log n + \mathcal{M}_{PQ})$ time where $\mathcal{M}_{PQ}$ is defined in Lemma 2.*

1. $\text{MAST}(T_1, T_2)$.
2. $\text{MAST}(T_1, T_2^v)$ *and* $\text{MAST}(T_1^u, T_2)$ *for all intersecting node pairs (u, v) of (P, Q).*

Proof. Let $S = \{(r(P), r(Q))\} \cup \{(r(P), v), (u, r(Q)) \mid (u, v)$ is an intersecting node pair$\}$. Note that $|S| = O(\ell_{PQ})$ and the set of values required by this lemma is $\{\text{MAST}(\Upsilon, \Gamma) \mid (\Upsilon, \Gamma) \in S\}$. Based on Lemmas 2 and 3, these values can be computed using $O(\ell_{PQ} \log n + \mathcal{M}_{PQ})$ time, as stated.

3 An Algorithm for Computing $\text{MAST}(T_1, T_2)$

In this section, we present a recursive algorithm for computing $\text{MAST}(T_1, T_2)$. This algorithm recursively computes $\text{MAST}(\Upsilon, T_2)$ for every sidetree Υ attached to the root centroid path of T_1. The information gathered is then used to compute $\text{MAST}(T_1, T_2)$ using a dynamic programming approach based on Theorem 1. Figure 2 presents our algorithm. For the sake of recursion, the algorithm computes values other than $\text{MAST}(T_1, T_2)$, namely, $\text{MAST}(T_1, T_2^v)$ for all nodes v of T_2. Steps 3 and 4 of the algorithm are further explained below.

Step 3 is based on the following observation of Farach and Thorup [5]. Let P be the root centroid path of T_1. Let Υ be a sidetree of P. Let U be a set of nodes in $T_2 \| \Upsilon$. Then for all $v \in T_2$, $\text{MAST}(\Upsilon, T_2^v) = 0$ if v has no descendant in U; otherwise, $\text{MAST}(\Upsilon, T_2^v) = \text{MAST}(\Upsilon, (T_2 \| \Upsilon)^u)$ where u is the highest descendant of v in U. Thus, for every $\Upsilon \in \text{SIDE}(P)$ and every node v of T_2, $\text{MAST}(\Upsilon, T_2^v)$ can be retrieved by finding the highest descendant of v in U. Using Euler tours [3], Step 3 can preprocess in linear time the $O(n)$ values obtained in Step 2 so that $\text{MAST}(\Upsilon, T_2^v)$ can be retrieved in $O(\log n)$ time.

Input: T_1, T_2;
Output: $\text{MAST}(T_1, T_2^v)$ for all nodes v of T_2;

1. Let P be the root centroid path of T_1;
2. For each sidetree Υ of P, call recursively $\text{Agree}(\Upsilon, T_2 \| \Upsilon)$;
3. Using the values found in Step 2, do an $O(n)$ time preprocessing so that for every $\Upsilon \in \text{SIDE}(P)$ and every node v of T_2, $\text{MAST}(\Upsilon, T_2^v)$ can be retrieved in $O(\log n)$ time;
4. For each centroid path $Q \in \mathcal{D}(T_2)$ in increasing order according to $\prec$:
   (a) Extract the values of $\text{MAST}(\Upsilon, \Gamma)$ for all $(\Upsilon, \Gamma) \in \mathcal{B}(P, Q)$;
   (b) Find $\text{MAST}(T_1, T_2^v)$ and $\text{MAST}(T_1^u, T_2^{r(Q)})$ for all intersecting node pairs (u, v) of (P, Q) based on Theorem 1;
   (c) Find $\text{MAST}(T_1, T_2^v)$ for all nodes v of Q;

Fig. 2. Algorithm $\text{Agree}(T_1, T_2)$

Let us consider Step 4. We handle $Q \in \mathcal{D}(T_2)$ in increasing order according to $\prec$. When the algorithm handles Q, it first retrieves $\text{MAST}(\Upsilon, \Gamma)$ for all $(\Upsilon, \Gamma) \in \mathcal{B}(P, Q)$ based on Step 4a. Recall that for every pair (Υ, Γ) of $\mathcal{B}(P, Q)$, either Υ is a sidetree of P or Γ is a sidetree of Q. For all $(\Upsilon, \Gamma) \in \mathcal{B}(P, Q)$ such that Υ is a sidetree of P, the values $\text{MAST}(\Upsilon, \Gamma)$ can be extracted in $O(\ell_{PQ} \log n)$ time after the preprocessing in Step 2. The rest of the pairs in $\mathcal{B}(P, Q)$ must be of the form (T_1^u, Γ) where u is a node of P and Γ is a sidetree of Q. For each such (T_1^u, Γ), it can be verified that $\text{MAST}(T_1^u, \Gamma)$ is already available when the path pair (P, Q') is handled where Q' is the root centroid path of Γ. After the retrieval of $\text{MAST}(\Upsilon, \Gamma)$ for all $(\Upsilon, \Gamma) \in \mathcal{B}(P, Q)$, the values wanted in Step 4b can be found based on Theorem 1.

To explain Step 4c, let $v_1, v_2, \ldots, v_k$ be a subsequence of nodes in Q such that $v_1 = r(Q)$ and for each v_i with $i > 1$, there exists some u in P such that (u, v_i) is intersecting. The values of $\text{MAST}(T_1, T_2^{v_i})$ for all v_i are available from Step 4b. The value of $\text{MAST}(T_1, T_2^v)$ for v located in a subpath between v_i and v_{i+1} is $\text{MAST}(T_1, T_2^{v_{i+1}})$. As a result, Step 4 computes the values of $\text{MAST}(T_1, T_2^v)$ for all nodes v of T_2.

Section 3.1 shows that $\text{Agree}(T_1, T_2)$ takes $O(\sqrt{d} W_{T_1,T_2} \log n \log(nN))$ time for $N \neq 1$. Section 3.2 considers the case where $N = 1$, for which the time complexity is only $O(\sqrt{d} W_{T_1,T_2} \log^2 \frac{n}{d})$.

3.1 Time Complexity of $\text{Agree}(T_1, T_2)$ for $N \neq 1$

Lemma 4. *Let P be the root centroid path of T_1. Then,*

$$\sum_{Q \in \mathcal{D}(T_2)} \ell_{PQ} = O\left(W_{P,T_2} \log |T_2| + \sum \left\{ W_{\Upsilon,T_2} \log \frac{|T_2|}{|T_2 \| \Upsilon|} \mid \Upsilon \in \text{SIDE}(P) \right\} \right).$$

Proof. Note that $\sum\{\ell_{PQ} \mid Q \in \mathcal{D}(T_2)\} = A_1 + A_2 + A_3 + A_4$ where $A_1 =$ the total number of node pairs of (P, Q) which are labeled with the same symbol over

all $Q \in \mathcal{D}(T_2)$; A_2 = the total number of intersecting sidetree-node pairs (u, Γ) of (P, Q) over all $Q \in \mathcal{D}(T_2)$; A_3 = the total number of intersecting sidetree-node pairs (Υ, v) of (P, Q) over all $Q \in \mathcal{D}(T_2)$; and A_4 = the total number of intersecting sidetree pairs (Υ, Γ) of (P, Q) over all $Q \in \mathcal{D}(T_2)$.

For a fixed Q, the number of node pairs of (P, Q) which are labeled with the same symbol is at most $\sum_{u \in P, v \in Q} \delta(u, v) = W_{P,Q}$. Therefore, $A_1 = \sum\{W_{PQ} \mid Q \in \mathcal{D}(T_2)\} = W_{P,T_2}$.

For a fixed sidetree Υ of P and a fixed Q, the number of intersecting sidetree-node pairs (Υ, v) of (P, Q) is at most $W_{\Upsilon,Q}$. Therefore, $A_2 = \sum\{W_{\Upsilon,Q} \mid \Upsilon \in \text{SIDE}(P)$ and $Q \in \mathcal{D}(T_2)\} = \sum\{W_{\Upsilon,T_2} \mid \Upsilon \in \text{SIDE}(P)\}$.

For a fixed sidetree Γ of a particular Q, the number of intersecting sidetree-node pairs (u, Γ) of (P, Q) is at most $W_{P,\Gamma}$. Therefore, $A_3 = \sum\{W_{P,\Gamma} \mid \Gamma \in \text{SIDE}(Q)$ and $Q \in \mathcal{D}(T_2)\} = W_{P,T_2} \log |T_2|$.

For a fixed sidetree Υ of P, let R be the tree formed from $T_2 \| \Upsilon$ by adding an edge between the root of T_2 and that of $T_2 \| \Upsilon$. Let $\mathcal{S} = \cup\{\text{SIDE}(Q) \mid Q \in \mathcal{D}(T_2)\}$. For each edge (x, y) in R with x being the parent of y, (x, y) corresponds to a simple path Q_{xy} in T_2 from x to y. By the definition of intersecting, for all sidetrees $\Psi \in \mathcal{S}$, (Υ, Ψ) is an intersecting sidetree pair if and only if the root of Ψ is on some path Q_{xy} where (x, y) is an edge in R.

For each edge (x, y) in R, the number of sidetrees whose roots are on Q_{xy} is less than $\log \frac{|T_2^x|}{|T_2^y|}$. Thus, the number of sidetrees in $\mathcal{S}$ which is intersecting with Υ is less than $\text{SUM}(R) = \sum_{(x,y) \in R} \log \frac{|T_2^x|}{|T_2^y|}$. We claim that $\text{SUM}(R) = O\left(|T_2 \| \Upsilon| \log \frac{|T_2|}{|T_2 \| \Upsilon|}\right)$. Then,

$$\begin{aligned} A_4 &= O\left(\sum \left\{ |T_2 \| \Upsilon| \log \frac{|T_2|}{|T_2 \| \Upsilon|} \mid \Upsilon \in \text{SIDE}(P) \right\}\right) \\ &= O\left(\sum \left\{ W_{\Upsilon,T_2} \log \frac{|T_2|}{|T_2 \| \Upsilon|} \mid \Upsilon \in \text{SIDE}(P) \right\}\right). \end{aligned}$$

The rest of this proof shows that $\text{SUM}(R) = O\left(|T_2 \| \Upsilon| \log \frac{|T_2|}{|T_2 \| \Upsilon|}\right)$. Let Z be the set of all the edges $(x, y) \in R$ where y is a leaf. Since $|Z| \leq |R|$ and $\{T_2^x \mid (x, y) \in Z\}$ is a set of disjoint subtrees of T_2, $\sum_{(x,y) \in Z} \log \frac{|T_2^x|}{|T_2^y|} \leq \sum_{(x,y) \in Z} \log |T_2^x| \leq |R| \log \frac{|T_2|}{|R|}$. Let R' be the tree obtained by first removing all the edges in Z and then replacing every maximal internally branchless unlabeled path with an edge between its two endpoints. (An internally branchless unlabeled path is a path from a node to a proper descendant such that all its nodes, except possibly the endpoints, are unlabeled and have at most one child each.) Note that $|R'| \leq |R|/2$ and $\text{SUM}(R) = |R| \log \frac{|T_2|}{|R|} + \text{SUM}(R')$. Since $|R| = |T_2 \| \Upsilon|$, $\text{SUM}(R) = O\left(|R| \log \frac{|T_2|}{|R|}\right) = O\left(|T_2 \| \Upsilon| \log \frac{|T_2|}{|T_2 \| \Upsilon|}\right)$.

Theorem 2. *Agree*(T_1, T_2) *takes* $O(\sqrt{d} W_{T_1,T_2} \log n \log(nN))$ *time.*

Proof. Let $\Phi(T_1, T_2)$ be the time complexity of Agree(T_1, T_2). Steps 1 and 3 of Agree(T_1, T_2) takes $O(n)$ time. Step 2 takes time $\sum\{\Phi(\Upsilon, T_2 \| \Upsilon) \mid \Upsilon \in \text{SIDE}(P)\}$.

Step 4a takes time $O(\sum\{\ell_{PQ}\log n \mid Q \in \mathcal{D}(T_2)\})$. By Theorem 1, Step 4b takes time $O(\sum\{\sqrt{d}\ell_{PQ}\log(nN) \mid Q \in \mathcal{D}(T_2)\})$. Step 4c requires $O(\sum\{|Q| \mid Q \in \mathcal{D}(T_2)\}) = O(n)$ time. In summary,

$$\Phi(T_1,T_2) = O\left(\sum_{Q\in\mathcal{D}(T_2)} \sqrt{d}\ell_{PQ}\log(nN)\right) + \sum_{\Upsilon\in\text{SIDE}(P)} \Phi(\Upsilon, T_2\|\Upsilon)$$

By Lemma 4 and Fact 1,

$$\Phi(T_1,T_2) = O(\sqrt{d}W_{T_1,T_2}\log|T_2|\log(nN)) = O(\sqrt{d}W_{T_1,T_2}\log n\log(nN)).$$

3.2 Time Complexity of Agree(T_1, T_2) for $N = 1$

This section assumes $N = 1$. Let $\mathcal{T}'(T_1,T_2)$ be the time required for Steps 1, 3, 4a, 4c, and the non-matching computation of Step 4b for all recursion levels of Agree (T_1,T_2). Let $\mathcal{T}_1''(T_1,T_2)$ and $\mathcal{T}''(T_1,T_2)$ be the times required for the matching computation of Step 4b for the first recursion level and for all recursion levels of Agree(T_1,T_2), respectively.

Lemma 5. $\mathcal{T}'(T_1,T_2) = O(W_{T_1,T_2}\log^2 n)$.

Proof. In the first recursion level, Steps 1, 3, 4a, and 4c take time $O(\sum\{\ell_{PQ}\log n \mid Q \in \mathcal{D}(T_2)\})$ by an argument similar to the proof of Theorem 2. The non-matching computation of Step 4b requires $O(\sum\{\ell_{PQ}\log n \mid Q \in \mathcal{D}(T_2)\})$ time. Steps 1, 3, 4a, 4c, and the non-matching computation part of Step 4b in all the remaining recursion levels take time $\sum\{\mathcal{T}'(\Upsilon,T_2\|\Upsilon) \mid \Upsilon \in \text{SIDE}(P)\}$. Therefore, $\mathcal{T}'(T_1,T_2) = \sum_{Q\in\mathcal{D}(T_2)}\ell_{PQ}\log n + \sum_{\Upsilon\in\text{SIDE}(P)}\mathcal{T}'(\Upsilon,T_2\|\Upsilon)$. By Lemma 4 and Fact 1, $\mathcal{T}'(T_1,T_2) = O(W_{T_1,T_2}\log|T_2|\log n) = O(W_{T_1,T_2}\log^2 n)$.

Lemma 6. *If $N = 1$, then*

$$\mathcal{T}_1''(T_1,T_2) = O\left(\min\left\{\sqrt{d}W_{T_1,T_2}\log\frac{|T_2|}{d}, \sqrt{|T_1|}W_{T_1,T_2}\log\frac{|T_2|}{|T_1|}\right\}\right).$$

Proof. Let the root centroid path in T_2 be the *level*-0 centroid path. For every centroid path X, if the parent of $r(X)$ is a node in a level-i centroid path, let X be a *level*-$(i+1)$ centroid path. This classification divides $\mathcal{D}(T_2)$ into level-i centroid paths for $i = 1,\ldots\lceil\log|T_2|\rceil$. Let $\Delta_i = \sum\left\{\min\left[\sqrt{d}, \sqrt{|T_1|}, \sqrt{|T_2|^{r(Q)}}\right] W_{T_1,T_2^{r(Q)}} \mid Q \text{ is a level-}i \text{ centroid path of } T_2\right\}$. Note that for each level-i centroid path Q, $|T_2^{r(Q)}| \le \frac{|T_2|}{2^i}$. Also, if Q_1 and Q_2 are two different level-i centroid paths, then $T_2^{r(Q_1)}$ and $T_2^{r(Q_2)}$ are disjoint. Therefore, Δ_i is not greater than

$$\min\left\{\sqrt{d}, \sqrt{|T_1|}, \sqrt{\frac{|T_2|}{2^i}}\right\}\sum\{W_{T_1,T_2^{r(Q)}} \mid Q \text{ is a level-}i \text{ centroid path of } T_2\}$$
$$\le \min\left\{\sqrt{d}, \sqrt{|T_1|}, \sqrt{\frac{|T_2|}{2^i}}\right\} W_{T_1,T_2}.$$

By Theorem 1, the matching computation of Step 4b takes time $O\left(\sum\left\{\min\left[\sqrt{d}, \sqrt{|T_1|}, \sqrt{|T_2^{r(Q)}|}\right] W_{T_1,T_2^{r(Q)}} \mid Q \in \mathcal{D}(T_2)\right\}\right)$. Therefore, Step 4b takes time

$$
\begin{aligned}
& O\left(\sum_{i=1}^{\lceil \log |T_2| \rceil} \Delta_i\right) \\
&= O\left(\sum_{i=1}^{\left\lceil \log \frac{|T_2|}{\min\{d,|T_1|\}} \right\rceil} \min\left\{\sqrt{d}, \sqrt{|T_1|}\right\} W_{T_1,T_2} + \sum_{i=\left\lceil \log \frac{|T_2|}{\min\{d,|T_1|\}} \right\rceil}^{\lceil \log |T_2| \rceil} W_{T_1,T_2} \sqrt{\frac{|T_2|}{2^i}}\right) \\
&= O\left(\min\left\{\sqrt{d}, \sqrt{|T_1|}\right\} \log \frac{|T_2|}{\min\{d,|T_1|\}} W_{T_1,T_2}\right) \\
&= O\left(\min\left\{\sqrt{d} W_{T_1,T_2} \log \frac{|T_2|}{d}, \sqrt{|T_1|} W_{T_1,T_2} \log \frac{|T_2|}{|T_1|}\right\}\right), \text{ as stated.}
\end{aligned}
$$

Lemma 7. *If $N = 1$, then $\mathcal{T}''(T_1, T_2) = O\left(\sqrt{d} W_{T_1,T_2} \log^2 \frac{n}{d}\right)$.*

Proof. By Lemma 6, the matching computation of Step 4b in the first recursion level takes time $\mathcal{T}_1''(T_1, T_2) = O\left(\min\left\{\sqrt{d} W_{T_1,T_2} \log \frac{|T_2|}{d}, \sqrt{|T_1|} W_{T_1,T_2} \log \frac{|T_2|}{|T_1|}\right\}\right)$. For the remaining recursion levels, the matching computation of Step 4b takes time $\sum\{\mathcal{T}''(\Upsilon, T_2 \| \Upsilon) \mid \Upsilon \in \text{SIDE}(P)\}$. Therefore, $\mathcal{T}''(T_1, T_2) =$

$$
O\left(\min\left\{\sqrt{d} W_{T_1,T_2} \log \frac{|T_2|}{d}, \sqrt{|T_1|} W_{T_1,T_2} \log \frac{|T_2|}{|T_1|}\right\}\right) + \sum_{\Upsilon \in \text{SIDE}(P)} \mathcal{T}''(\Upsilon, T_2 \| \Upsilon) \tag{2}
$$

As in Lemma 6, the centroid paths of T_1 can be classified into level-i centroid paths for $i = 1, \ldots, \lceil \log |T_1| \rceil$. Define Δ_i to be

$$
\sum\left\{\min\left[\sqrt{d} W_{\Upsilon,T_2} \log \frac{|T_2|}{d}, \sqrt{|\Upsilon|} W_{\Upsilon,T_2} \log \frac{|T_2|}{|\Upsilon|}\right] \mid \Upsilon \text{ is a sidetree of some level-}i \text{ centroid path of } T_1\right\}.
$$

Note that $|\Upsilon| \le \frac{|T_1|}{2^i}$ for every sidetree Υ of some level-i centroid path of T_1. Also, for two different sidetrees Υ_1 and Υ_2 of some level-i centroid paths of T_1, Υ_1 and Υ_2 are disjoint. Hence,

$$
\Delta_i \le \min\left\{\sqrt{d} W_{T_1,T_2} \log \frac{|T_2|}{d}, \sqrt{\frac{|T_1|}{2^i}} W_{T_1,T_2} \log \frac{2^i |T_2|}{|T_1|}\right\}.
$$

Since Equation (2) is a tail recursion, $\mathcal{T}''(T_1, T_2) =$

$$
\begin{aligned}
& O\left(\sum_{i=1}^{\lceil \log |T_1| \rceil} \Delta_i\right) \\
&= O\left(\sum_{i=1}^{\lceil \log \frac{|T_1|}{d} \rceil} \sqrt{d} W_{T_1,T_2} \log \frac{|T_2|}{d} + \sum_{i=\lceil \log \frac{|T_1|}{d} \rceil}^{\lceil \log |T_1| \rceil} \sqrt{\frac{|T_1|}{2^i}} W_{T_1,T_2} \log \frac{2^i |T_2|}{|T_1|}\right) \\
&= O\left(\sqrt{d} W_{T_1,T_2} \log \frac{|T_2|}{d} \log \frac{|T_1|}{d}\right) \\
&= O\left(\sqrt{d} W_{T_1,T_2} \log^2 \frac{n}{d}\right).
\end{aligned}
$$

Theorem 3. *If $N = 1$, then Agree(T_1, T_2) takes $O(\sqrt{d} W_{T_1,T_2} \log^2 \frac{n}{d})$ time.*

Proof. By Lemmas 5 and 7, Agree(T_1, T_2) takes time $\mathcal{T}'(T_1, T_2) + \mathcal{T}''(T_1, T_2)$ $= O(\sqrt{d} W_{T_1,T_2} \log^2 \frac{n}{d})$.

4 Open Problems

This paper shows that MAST(T_1, T_2) can be computed in $O(\sqrt{d} W_{T_1,T_2} \log n \log(nN))$ time for $N \neq 1$ and in $O(\sqrt{d} W_{T_1,T_2} \log^2 \frac{n}{d})$ time for $N = 1$. Note that if T_1 and T_2 are evolutionary trees, our results do not match the result in [21] for small d. Thus, we conjecture that MAST(T_1, T_2) can be computed in $O(\sqrt{d} W_{T_1,T_2} \log(nN))$ time for $N \neq 1$ and in $O(\sqrt{d} W_{T_1,T_2} \log \frac{n}{d})$ time for $N = 1$.

References

1. M. J. Chung. $O(n^{2.5})$ time algorithms for the subgraph homeomorphism problem on trees. *Journal of Algorithms*, 8:106–112, 1987.
2. R. Cole and R. Hariharan. An $O(n \log n)$ algorithm for the maximum agreement subtree problem for binary trees. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 323–332, 1996.
3. T. H. Cormen, C. L. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1991.
4. M. Farach, T. M. Przytycka, and M. Thorup. Computing the agreement of trees with bounded degrees. In P. Spirakis, editor, *Lecture Notes in Computer Science 979: Proceedings of the 3rd Annual European Symposium on Algorithms*, pages 381–393. Springer-Verlag, New York, NY, 1995.
5. M. Farach and M. Thorup. Sparse dynamic programming for evolutionary-tree comparison. *SIAM Journal on Computing*, 26:210–230, 1997.
6. C. R. Finden and A. D. Gordon. Obtaining common pruned trees. *Journal of Classification*, 2:255–276, 1985.
7. J. Friedman. Expressing logical formulas in natural languages. In J. Groenendijk, T. Janssen, and M. Stokhof, editors, *Formal methods in the study of language*, pages 113–130. Mathmatical Centre, Amsterdam, 1981.

8. H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18:1013–1036, 1989.
9. A. Gupta and N. Nishimura. Finding largest subtrees and smallest supertrees. *Algorithmica*, 21(2):183–210, 1998.
10. D. M. Hillis, C. Moritz, and B. K. Mable, editors. *Molecular Systematics*. Sinauer Associates, Sunderland, Ma, 2nd edition, 1996.
11. M. Y. Kao. Tree contractions and evolutionary trees. *SIAM Journal on Computing*, 27:1592–1616, 1998.
12. M. Y. Kao, T. W. Lam, W. K. Sung, and H. F. Ting. All-cavity maximum matchings. In *Lecture Notes in Computer Science 1350: Proceedings of the 8th Annual International Symposium on Algorithms and Computation*, pages 364–373, 1997.
13. M. Y. Kao, T. W. Lam, W. K. Sung, and H. F. Ting. A decomposition theorem for maximum weight bipartite matchings with applications to evolutionary trees. In *Lecture Notes in Computer Science: Proceedings of the 8th Annual European Symposium on Algorithms*, pages 438–449. Springer-Verlag, New York, NY, 1999.
14. P. Kilpeläinen and H. Mannila. Retrieval from hierarchical texts by partial patterns. In *Proceedings of the 16th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 214–222, 1991.
15. P. Kilpeläinen and H. Mannila. Grammatical tree matching. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Lecture Notes in Computer Science 644: Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching*, pages 162–174. Springer-Verlag, New York, NY, 1992.
16. B. Kimia, A. Tannenbaum, and S. W. Zucker. Shapes, shocks, and deformations, I. *International Journal of Computer Vision*, pages 189–224, 1995.
17. E. Kubicka, G. Kubicki, and F. McMorris. An algorithm to find agreement subtrees. *Journal of Classification*, 12:91–99, 1995.
18. S. Y. Le, J. Owens, R. Nussinov, J. H. Chen, B. Shapiro, and J. V. Maizel. RNA secondary structures: comparison and determination of frequently recurring substructures by consensus. *Computer Application in Bioscience*, 5:205–210, 1989.
19. H. Mannila and K. J. Räihä. On query languages for the p-string data model. In H. Kangassalo, S. Ohsuga, and H. Jaakkola, editors, *Information Modelling and Knowledge Bases*, pages 469–482. IOS Press, Amsterdam, 1990.
20. P. Materna, P. Sgall, and Z. Hajicova. Linguistic constructions in transparent intensional logic. *Prague Bulletin on Mathematical Linguistics*, pages 27–32, 1985.
21. T. Przytycka. Sparse dynamic programming for maximum agreement subtree problem. In B. Mirkin, F. R. McMorris, F. S. Roberts, and A. Rzhetsky, editors, *Mathematical Hierarchies and Biology*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 249–264, Providence, RI, 1997. American Mathematical Society.
22. B. Shapiro and K. Zhang. Comparing multiple RNA secondary structures using tree comparisons. *Compter Applications in Bioscience*, pages 309–318, 1990.
23. F. Y. Shih. Object representation and recognition using mathematical morphology model. *Journal of Systems Integration*, pages 235–256, 1991.
24. F. Y. Shih and O. R. Mitchell. Threshold decomposition of grayscale morphology into binary morphology. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11:31–42, 1989.
25. M. Steel and T. Warnow. Kaikoura tree theorems: Computing the maximum agreement subtree. *Information Processing Letters*, 48:77–82, 1993.
26. Y. Takahashi, Y. Satoh, H. Suzuki, and S. Sasaki. Recognition of largest common structural fragment among a variety of chemical structures. *Analytical Science*, pages 23–28, 1987.

A Lemma 3

Consider $P \in \mathcal{D}(T_1)$ and $Q \in \mathcal{D}(T_2)$. Let $u_1, u_2, \ldots, u_p$ ($v_1, v_2, \ldots, v_q$, respectively) be the nodes in P (Q, respectively) in order from root to leaf. Let $G(P,Q)$ be the bipartite multisubgraph between $u_1, \ldots, u_p$ and $v_1, \ldots, v_q$ where (u_i, v_j) is an edge if and only if (u_i, v_j) is an intersecting node pair of (P,Q). The multiplicity and weights of (u_i, u_j) are specified as follows.

If u_i and v_j have the same label or are both unlabeled, (u_i, v_j) has two weighted copies:

- **white edge:** The weight is $\text{MWM}(H_{u_i v_j})$ if u and v are both unlabeled; the weight is $\text{MWM}(H_{u_i v_j}) + \mu(z)$ if u and v have the same label z.
- **gray edge:** Let $H'_{u_i v_j}$ be the bipartite subgraph of $(C(u_i), C(v_j), C(u_i) \times C(v_j) - \{(\text{hvy}(u_i), \text{hvy}(v_j))\})$ where (x,y) is an edge of $H'_{u_i v_j}$ if and only if $\text{MAST}(T_1^x, T_2^y) > 0$ and where the weight of (x,y) is $\text{MAST}(T_1^x, T_2^y)$. If u and v are both unlabeled, the weight of the gray edge is $\text{MWM}(H'_{u_i v_j})$; if u and v have the same label z, the weight is the maximum of $\text{MWM}(H'_{u_i v_j}) + \mu(z)$, $\max\{\text{MAST}(T_1^u, \Gamma) \mid \Gamma \text{ is a sidetree of } v\}$, and $\max\{\text{MAST}(\Upsilon, T_2^v) \mid \Upsilon \text{ is a sidetree of } u\}$.

If u_i and v_j are unlabeled, then (u_i, v_j) has two additional weighted copies:

- **green edge:** The weight is $\max\{\text{MAST}(T_1^{u_i}, \Gamma) \mid \Gamma \text{ is a sidetree of } v_j\}$.
- **red edge:** The weight is $\max\{\text{MAST}(\Upsilon, T_2^{v_j}) \mid \Upsilon \text{ is a sidetree of } u_i\}$.

If u_i and v_j have different symbols, then (u_i, v_j) has only one weighted copy:

- **gray edge:** The weight is the greater of $\max\{\text{MAST}(T_1^u, \Gamma) \mid \Gamma \text{ is a sidetree of } v\}$ and $\max\{\text{MAST}(\Upsilon, T_2^v) \mid \Upsilon \text{ is a sidetree of } u\}$.

Lemma 8 shows that $G(P,Q)$ can be constructed in $O(\ell_{PQ} \log d)$ time based on the following two facts.

Fact 3 (See [12]). *Consider a bipartite graph G and two nodes x and y of G. Let M be a maximum weight matching of $G - \{x, y\}$. Then, $\text{MWM}(G)$ can be computed by augmenting M using at most two augmenting paths.*

Fact 4 (See [21]). *For all intersecting node pairs (u,v) of (P,Q), we can construct graphs H''_{uv} from H'_{uv} in $O(\ell_{PQ} \log d)$ time such that $\text{MWM}(H''_{uv}) = \text{MWM}(H'_{uv})$, $|H''_{uv}| = |H_{uv}|$, and H_{uv} is a subgraph of H''_{uv}.*

Lemma 8. *Given $\text{MAST}(\Upsilon, \Gamma)$ for all $(\Upsilon, \Gamma) \in \mathcal{B}(P,Q)$ and a maximum weight matchings of H_{uv} for every intersecting node pair (u,v) of (P,Q), $G(P,Q)$ can be constructed in $O(\ell_{PQ} \log d)$ time.*

Proof. This proof shows that for all intersecting node pairs (u,v) of (P,Q), the values $\text{MWM}(H_{uv})$, $\text{MWM}(H'_{uv})$, $\max\{\text{MAST}(T_1^u, \Gamma) \mid \Gamma \text{ is a sidetree of } v\}$, and $\max\{\text{MAST}(\Upsilon, T_1^v) \mid \Upsilon \text{ is a sidetree of } u\}$ can be computed in $O(\ell_{PQ} \log d)$ time. Then, $G(P,Q)$ can be constructed using $O(\ell_{PQ})$ additional time.

For a fixed v of Q, let $u_{i_1}, \ldots, u_{i_z}$ be the nodes of P with $i_1 > i_2 > \ldots > i_z$ such that (u_{i_k}, v) is intersecting. We can compute $f(k) = \max\{\textsc{mast}(T_1^{u_{i_k}}, \Gamma) \mid \Gamma \text{ is a sidetree of } v\}$ for all k based on the recurrence $f(k) = \max\{f(k-1), \max\{\textsc{mast}(T_1^{u_{i_k}}, \Gamma) \mid \Gamma \text{ is a sidetree of } v \text{ and } (T_1^{u_{i_k}}, \Gamma) \in \mathcal{B}(u_{i_k}, v)\}\}$. This takes $O(\sum_k |\mathcal{B}(u_{i_k}, v)|)$ time. Therefore, we can compute $\max\{\textsc{mast}(T_1^u, \Gamma) \mid \Gamma$ is a sidetree of $v\}$ for all intersecting node pairs (u, v) in time

$$O(\sum_{v \in Q} \sum_{u \in P} |\mathcal{B}(u, v)|) = O(\ell_{PQ}).$$

Similarly, we can compute $\max\{\textsc{mast}(T_1^u, \Gamma) \mid \Gamma \text{ is a sidetree of } v\}$ for all intersecting node pairs (u, v) with the same time complexity.

The values $\textsc{mwm}(H_{uv})$ are already available. The rest of this proof considers the computation of $\textsc{mwm}(H'_{uv})$. By Fact 4, we can construct H''_{uv} from H'_{uv} for all intersecting node pairs (u, v) using $O(\ell_{PQ} \log d)$ time such that $\textsc{mwm}(H''_{uv}) = \textsc{mwm}(H'_{uv})$, $|H''_{uv}| = O(|H'_{uv}|)$ and H''_{uv} is a subgraph of H'_{uv}. Then, as $H''_{uv} - \{\text{hvy}(u), \text{hvy}(v)\} = H_{uv}$, by Fact 3, each $\textsc{mwm}(H''_{uv})$ can be computed by augmenting a maximum weight matching of H_{uv} using at most two augmenting paths. Therefore, the values of all $\textsc{mwm}(H''_{uv})$ and hence $\textsc{mwm}(H'_{uv})$ can be computed using time $\sum_{u \in P, v \in Q} 2|H''_{uv}| = O(\sum_{u \in P, v \in Q} |H_{uv}|) = O(\ell_{PQ})$. In summary, the values of $\textsc{mwm}(H'_{uv})$ for all intersecting node pairs (u, v) can be computed in $O(\ell_{PQ} \log d)$ time.

For edges (u_i, v_j) and $(u_{i'}, v_{j'})$ of $G(P, Q)$, (u_i, v_j) is *below* $(u_{i'}, v_{j'})$ if $i > i'$ and $j > j'$. They form a *crossing* if ($i < i'$ and $j > j'$) or ($i > i'$ and $j < j'$). A matching in $G(P, Q)$ is an *agreement* matching [21] if the following statements hold:

1. Its white edges, if any, do not form any crossing.
2. In addition to those non-crossing white edges, it may contain (1) at most one red-green crossing (i.e., the crossing between a red edge (u_i, v_j) and a green edge $(u_{i'}, v_{j'})$ with $i < i'$) or (2) at most one gray edge which is below all the white edges.

The *weight* of an agreement matching is the total weights of its edges. A *maximum* agreement matching is an agreement matching in $G(P, Q)$ which maximizes the weight; this maximum weight is denoted as $\textsc{mam}(G(P, Q))$. The following lemma shows the relationship between maximum agreement matchings and maximum agreement subtrees.

Lemma 9. *For any $u \in P$ and $v \in Q$, $\textsc{mast}(T_1^u, T_2^v) = \textsc{mam}(G(P^u, Q^v))$.*

Proof. The proof follows from the definitions of maximum agreement matchings and maximum agreement subtrees.

Fact 5 (See [2,21]). *Given $G(P, Q)$ and a set $S \subseteq P \times Q$, $\textsc{mam}(G(P^u, Q^v))$ for all $(u, v) \in S$ can be computed using $O((\ell_{PQ} + |S|) \log n)$ time.*

Lemma 3 follows from Lemmas 8, 9, and Fact 5.

Incomplete Directed Perfect Phylogeny

Itsik Pe'er, Ron Shamir, and Roded Sharan

Department of Computer Science
Tel-Aviv University, Tel-Aviv, Israel
{izik,shamir,roded}@math.tau.ac.il

Abstract. Perfect phylogeny is one of the fundamental models for studying evolution. We investigate the following generalization of the problem: The input is a species-characters matrix. The characters are binary and directed, i.e., a species can only gain characters. The difference from standard perfect phylogeny is that for some species the state of some characters is unknown. The question is whether one can complete the missing states in a way admitting a perfect phylogeny. The problem arises in classical phylogenetic studies, when some states are missing or undetermined. Quite recently, studies that infer phylogenies using inserted repeat elements in DNA gave rise to the same problem. The best known algorithm for the problem requires $O(n^2m)$ time for m characters and n species. We provide a near optimal $\tilde{O}(nm)$-time algorithm for the problem.

1 Introduction

When studying evolution, the divergence patterns leading from a single ancestor species to its contemporary descendants are usually modeled by a tree structure. Extant species correspond to the tree leaves, while their common progenitor corresponds to the root of this *phylogenetic tree*. Internal nodes correspond to hypothetical ancient species, which putatively split up and evolved into distinct species. Tree branches model changes through time of the hypothetical ancestor species. The common case is that one has information regarding the leaves, from which the phylogenetic tree is to be inferred. This task, called *phylogenetic reconstruction* (cf. [7]), was one of the first algorithmic challenges posed by biology, and the computational community has been dealing with problems of this flavor for over three decades (see, e.g., [12]).

In the character-based approach to tree reconstruction, contemporary species are described by their attributes or *characters*. Each character takes on one of several possible *states*. The input is represented by a matrix $\mathcal{A}$ where a_{ij} is the state of character j in species i, and the i-th row is the *character vector* of species i. The output sought is a hypothesis regarding evolution, i.e., a phylogenetic tree along with the suggested character-vectors of the internal nodes. This output must satisfy properties specified by the problem variant.

One important variant of the phylogenetic reconstruction problem is finding a *perfect phylogeny*. In this variant, the phylogenetic tree is required to have the property that for each state of a character, the set of all nodes that have that

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 143–153, 2002.

state induces a connected subtree. The general perfect phylogeny problem is NP-hard [4,20]. When considering the number of possible states per character as a parameter, the problem is fixed parameter tractable [1,15]. For *binary characters*, having only two states, perfect phylogeny is linear-time solvable [11].

Another common variant of phylogenetic reconstruction is the *parsimony* problem, which calls for a solution with fewest state changes altogether. A change is counted whenever the state of a character changes between a species and an offspring species. This problem is known to be NP-hard [8]. A special case introduced by Camin and Sokal [5] assumes that characters are binary and *directed*, namely, only $0 \rightarrow 1$ changes may occur. Noting by 1 and 0 the presence and absence, respectively, of the character, this means that characters can only be gained during evolution. Another related binary variant is Dollo parsimony [6,19], which assumes that the change $0 \rightarrow 1$ may happen only once, i.e., a character can be gained once, but it can be lost several times. Both of these variants are polynomially solvable (cf. [7]). When no perfect phylogeny is possible, one can seek a largest subset of the characters which admits a perfect phylogeny. Characters in such a subset are said to be *compatible*. Compatibility problems have been studied extensively (see, e.g., [17]).

In this paper, we discuss a generalization of binary perfect phylogeny which combines the assumptions of both Camin-Sokal parsimony and Dollo parsimony. The setup is as follows: The characters are binary and directed. As in perfect phylogeny, the input is a matrix of character vectors, with the difference that some character states are missing. The question is whether one can complete the missing states in a way admitting a perfect phylogeny. We call this problem *Incomplete Directed Perfect phylogeny (IDP)*.

The problem of handling incomplete phylogenetic data arises whenever some of the data is missing. It is also encountered in the context of morphological characters, where for some species it may be impossible to reliably assign a state to a character. The popular PAUP software package [21] provides an exponential solution to the problem by exhaustively searching the space of missing states. Indeed, the problem of determining whether a set of incomplete *undirected* characters is compatible was shown to be NP-complete, even in the case of binary characters [20].

Quite recently, a novel kind of genomic data has given rise to the same problem: Nikaido et al. [18] use inserted repetitive genomic elements, particularly SINEs, as a source of evolutionary information. SINEs are short DNA sequences that were copied and randomly reinserted into various genomic loci during evolution. The specific insertion events are identifiable by the flanking sequences on both sides of the insertion site. These insertions are assumed to be unique events in evolution, because the odds of having separate insertion events at the very same locus are negligible. Furthermore, a SINE insertion is assumed to be irreversible, i.e., once a SINE sequence has been inserted somewhere along the genome, it is practically impossible for the exact, complete SINE to leave that specific locus. However, the site and its flanking sequences may be lost when a large genomic region, which includes them, is deleted. In that case we do not

know whether an insertion had occurred in the missing site. One can model such data by assigning to each locus a character, whose state is '1' if the SINE occurred in that locus, '0' if the locus is present but does not contain the SINE, and '?' if the locus is missing. The resulting reconstruction problem is precisely Incomplete Directed Perfect phylogeny.

The incomplete perfect phylogeny problem becomes polynomial when the characters are directed: Benham et al. [3] studied the compatibility problem on generalized characters. Their work implies an $O(n^2m)$-time algorithm for IDP, where n and m denote the number of species and characters, respectively. A problem related to IDP is the consensus tree problem [2,13]. This problem calls for constructing a consensus tree from homeomorphic binary subtrees, and is solvable in polynomial time. One can reduce IDP to the latter problem, but the reduction may take $\Omega(n^2m)$ time.

Our approach to the IDP problem is graph theoretic. We first provide several graph and matrix characterizations for solvable instances of binary directed perfect phylogeny. We then reformulate IDP as a *graph sandwich* problem: The input data is recast into two graphs, and solving IDP is shown to be equivalent to finding a graph of a particular type "sandwiched" between them. This formulation allows us to devise a polynomial algorithm for IDP. The deterministic complexity of the algorithm is shown to be $O(nm + k\log^2(n+m))$, for an instance with k 1-entries in the matrix. Alternatively, we give a randomized version of the algorithm which takes $O(nm+k\log(l^2/k)+l(\log l)^3\log\log l)$ expected time, where $l = n + m$. Since an $\Omega(nm)$ lower bound was shown by Gusfield for directed binary perfect phylogeny [11], our algorithm has near optimal time complexity.

The paper is organized as follows: In Section 2 we provide some preliminaries. Section 3 gives the characterizations and the graph sandwich formulation. In Section 4 we present the polynomial algorithm. For lack of space, some proofs are shortened or omitted.

2 Problem Formulation

We first specify some terminology and notation. We reserve the terms *nodes* and *branches* for trees, and will use the terms *vertices* and *edges* for other graphs. Matrices are denoted by an upper-case letter, while their elements are denoted by the corresponding lower-case letter.

Let $G = (V, E)$ be a graph. We denote its set of vertices also by $V(G)$, and its set of edges also by $E(G)$. For a vertex $v \in V$ we define its *degree to a subset* $R \subseteq V$ to be the number of edges connecting v to vertices in R. The *length* of a path in G is the number of edges along it.

Let $\mathcal{T}$ be a rooted tree over a leaf set S with branches directed from the root towards the leaves. For a node x in $\mathcal{T}$ we denote the leaf set of the subtree rooted at x by $L(x)$. $L(x)$ is called a *clade* of $\mathcal{T}$. For consistency, we consider $\emptyset$ to be a clade of $\mathcal{T}$ as well, and call it the *empty clade*. $S, \emptyset$ and all singletons are called *trivial clades*. We denote by $triv(S)$ the collection of all trivial clades. Two sets

are said to be *compatible* if they are either disjoint, or one of them contains the other.

Observation 1. *A collection $\mathcal{S}$ of subsets of a set S is the set of clades of some tree over S iff $\mathcal{S}$ contains $triv(S)$ and its subsets are pairwise compatible.*

A tree is uniquely characterized by the set of its clades. The transformation between a branch-node representation of a tree and a list of its clades is straightforward. Hereafter, we identify a tree $\mathcal{T}$ with the set $\{S' : S' \text{ is a clade of } \mathcal{T}\}$. Let $\hat{S}$ be a subset of the leaves of $\mathcal{T}$. Then the subtree of $\mathcal{T}$ *induced* on $\hat{S}$ is the collection $\{\hat{S} \cap S' : S' \in \mathcal{T}\}$ (which defines a tree).

Throughout the paper we denote by $S = \{s_1, \ldots, s_n\}$ the set of all species and by $C = \{c_1, \ldots, c_m\}$ the set of all (binary) characters. For a graph K, we define $C(K) \equiv C \cap V(K)$ and $S(K) \equiv S \cap V(K)$. Let $\mathcal{B}_{n \times m}$ be a binary matrix whose rows correspond to species, each row being the character-vector of the corresponding species. That is, $b_{ij} = 1$ iff the species s_i has the character c_j. A *phylogenetic tree for* $\mathcal{B}$ is a rooted tree $\mathcal{T}$ with n leaves corresponding to the n species of S, such that each character c_j is associated with a clade S' of $\mathcal{T}$, satisfying:
(1) $s_i \in S'$ iff $b_{ij} = 1$.
(2) Every non-trivial clade of $\mathcal{T}$ is associated with at least one character.
For a character c, the node x of $\mathcal{T}$, whose clade $L(x)$ is associated with c, is called the *origin* of c w.r.t. $\mathcal{T}$.

Let $\mathcal{A}_{n \times m}$ be a $\{0, 1, ?\}$ matrix in which $a_{ij} = 1$ if s_i has c_j, $a_{ij} = 0$ if s_i lacks c_j, and $a_{ij} =?$ if it is not known whether s_i has c_j. $\mathcal{A}$ is called *incomplete* if it contains at least one '?'. For a character c_j and a value $x \in \{0, ?, 1\}$, the *x-set* of c_j in $\mathcal{A}$ is the set of species $c_j(\mathcal{A}, x) \equiv \{s_i : a_{ij} = x\}$. c_j is called a *null character* if its 1-set is empty.

A binary matrix $\mathcal{B}$ is called a *completion* of $\mathcal{A}$ if $a_{ij} \in \{b_{ij}, ?\}$ for all i, j. Thus, a completion replaces all the ?-s in $\mathcal{A}$ by zeroes and ones. If $\mathcal{B}$ has a phylogenetic tree $\mathcal{T}$, we say that $\mathcal{T}$ is a *phylogenetic tree for* $\mathcal{A}$ as well. We also say that $\mathcal{T}$ *explains* $\mathcal{A}$ *via* $\mathcal{B}$, and that $\mathcal{A}$ is *explainable*. An example of an incomplete matrix $\mathcal{A}$, a completion of $\mathcal{A}$, and a phylogenetic tree which explains $\mathcal{A}$, is given in Figure 1.

The following lemma, closely related to Observation 1, has been proven independently by several authors:

Lemma 1. *A binary matrix $\mathcal{B}$ has a phylogenetic tree iff the 1-sets of every two characters are compatible.*

An analogous lemma holds for undirected characters (cf. [11]). In contrast, for incomplete matrices, even if every pair of columns has a phylogenetic tree, the full matrix might not have one. Such an example was provided by [7] for undirected characters. We provide a simpler example for directed characters in Figure 2.

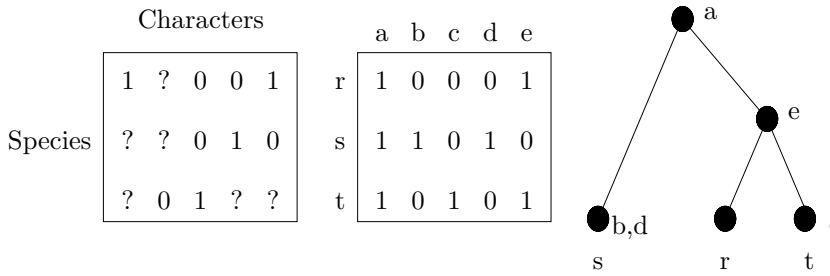


Fig. 1. Left to right: An incomplete matrix $\mathcal{A}$, a completion $\mathcal{B}$ of $\mathcal{A}$, and a phylogenetic tree that explains $\mathcal{A}$ via $\mathcal{B}$. A character x to the right of a vertex v means that v is the origin of x.

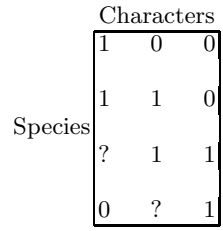


Fig. 2. An incomplete matrix which has no phylogenetic tree although every pair of its columns has one.

We are now ready to state the IDP problem:
Incomplete Directed Perfect Phylogeny (IDP):
Instance: An incomplete matrix $\mathcal{A}$.
Goal: Find a tree which explains $\mathcal{A}$, or determine that no such tree exists.

3 Characterizations of Explainable Binary Matrices

3.1 Forbidden Subgraph Characterization

Let $\mathcal{B}$ be a species-characters binary matrix of order $n \times m$. Construct the bipartite graph $G(\mathcal{B}) = (S, C, E)$ with $E = \{(s_i, c_j) : b_{ij} = 1\}$. For a subset $S' \subseteq S$ of species, we say that a character c is S'-*universal* in $\mathcal{B}$, if $S' \subseteq c(\mathcal{B}, 1)$.

An induced path of length four in $G(\mathcal{B})$ is called a Σ *subgraph* if it starts (and therefore ends) at a vertex corresponding to a species (see Figure 3). A graph with no induced Σ subgraph is said to be Σ-*free*.

Proposition 1. *If $G(\mathcal{B})$ is connected and Σ-free, then there exists a character which is S-universal in $\mathcal{B}$.*

Proof. Suppose to the contrary that $G(\mathcal{B})$ has no S-universal character. Consider the collection of all 1-sets of characters in $\mathcal{B}$. Let c be a character whose 1-set is maximal w.r.t. inclusion in this collection. Let s'' be a species which lacks c.

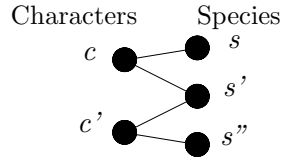


Fig. 3. The Σ subgraph.

Since c has a non-empty 1-set and $G(\mathcal{B})$ is connected, there exists a path from s'' to c in $G(\mathcal{B})$. Consider the shortest such path P. Since $G(\mathcal{B})$ is bipartite, the length of P is odd. P cannot be of length 1, by the choice of s''. Furthermore, if P is of length greater than 3, then its first 5 vertices induce a Σ subgraph, a contradiction. Thus $P = (s'', c', s', c)$ must be of length 3. By maximality of the 1-set of c, it is not contained in the 1-set of c'. Hence, there exists a species s which has the character c but lacks c'. Together with the vertices of P, s induces a Σ subgraph, as depicted in Figure 3, a contradiction.

The following theorem restates Lemma 1 in terms of graph theory.

Theorem 1. *$\mathcal{B}$ has a phylogenetic tree iff $G(\mathcal{B})$ is Σ-free.*

Corollary 1. *Let $\hat{\mathcal{A}}$ be a submatrix of $\mathcal{A}$. If $\mathcal{A}$ is explainable, then so is $\hat{\mathcal{A}}$. Furthermore, if $\mathcal{T}$ explains $\mathcal{A}$, then $\hat{\mathcal{A}}$ is explained by the subtree of $\mathcal{T}$ induced on its rows.*

Let Ψ be a graph property. In the Ψ *sandwich problem* one is given a vertex set V and a partition of $(V \times V) \setminus \{(v, v) : v \in V\}$ into three subsets: E_0 - the forbidden edges, E_1 - the mandatory edges, and $E_?$ - the optional edges. The objective is to find a supergraph of (V, E_1) which satisfies Ψ and contains no forbidden edges. For the property of having no induced Σ subgraphs, the problem is formally defined as follows:
Σ-Free-Sandwich:
Instance: A vertex set V, and two disjoint edge sets E_0, E_1 over V.
Question: Is there a set F of edges such that $F \supseteq E_1$, $F \cap E_0 = \emptyset$, and the graph (V, F) satisfies Ψ?

Proposition 2. *Σ-free-sandwich is equivalent to IDP.*

Hence, the required graph (V, F) must be "sandwiched" between (V, E_1) and $(V, E_1 \cup E_?)$. The reader is referred to [10,9] for a discussion of various sandwich problems.

Theorem 1 motivates looking at the IDP problem with input $\mathcal{A}$ as an instance $((S, C), E_0^{\mathcal{A}}, E_?^{\mathcal{A}}, E_1^{\mathcal{A}})$ of the *Σ-free sandwich problem*. Here, $E_x^{\mathcal{A}} = \{(s_i, c_j) : a_{ij} = x\}$, for $x = 0, ?, 1$. In the sequel, we omit the superscript $\mathcal{A}$ when it is clear from the context.

Note, that there is an obvious 1-1 correspondence between completions of $\mathcal{A}$ and possible solutions of $((S, C), E_0, E_?, E_1)$. Hence, in the sequel we refer to matrices and their corresponding sandwich instances interchangeably.

3.2 Forbidden Submatrix Characterizations

A binary matrix $\mathcal{B}$ is called *canonical* if it can be decomposed as follows:
(1) Its $k_0 \geq 0$ left columns are all zero.
(2) Its next $k_1 \geq 0$ columns are all one.
(3) There exist canonical matrices $\mathcal{B}_1, \ldots, \mathcal{B}_l$, such that the rest (0 or more) of the columns of $\mathcal{B}$ form the block-structure illustrated in Figure 4.

We say that a matrix $\mathcal{B}$ *avoids* a matrix $\mathcal{X}$, if no submatrix of $\mathcal{B}$ is identical to $\mathcal{X}$.

Theorem 2. *Let $\mathcal{B}$ be a binary matrix. The following are equivalent:*

1. *$\mathcal{B}$ has a phylogenetic tree.*
2. *$G(\mathcal{B})$ is Σ-free.*
3. *Every matrix obtained by permuting the rows and columns of $\mathcal{B}$ avoids the following matrix:*
$$\mathcal{Z} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$
4. *There exists an ordering of the rows and columns of $\mathcal{B}$ which yields a canonical matrix.*
5. *There exists an ordering of the rows and columns of $\mathcal{B}$ so that the resulting matrix avoids the following matrices:*

$$\mathcal{X}_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \mathcal{X}_2 = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}, \mathcal{X}_3 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \mathcal{X}_4 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

Proof.

1⇔2 Theorem 1.
2⇔3 Trivial.
1⇒4 Suppose $\mathcal{T}$ is a tree that explains $\mathcal{B}$. Assign to each node of $\mathcal{T}$ an index which equals its position in a preorder visit of $\mathcal{T}$. Sort all characters according to the preorder indices of their origin nodes (letting null characters come first). Sort all species according to the preorder indices of their corresponding leaves in $\mathcal{T}$. The result is a canonical matrix.
4⇒5 Easily verifiable from the definition of canonical matrices.
5⇒3 Suppose to the contrary that $\mathcal{B}$ has an ordering of its rows and columns, so that rows i_1, i_2, i_3 and columns j_1, j_2 of the resulting matrix compose the submatrix $\mathcal{Z}$. Consider the permutations $\theta_{row}, \theta_{col}$ of the rows and columns of $\mathcal{B}$, respectively, which yield a matrix avoiding $\mathcal{X}_1, \ldots, \mathcal{X}_4$. In this ordering, row $\theta_{row}(i_1)$ necessarily lies between rows $\theta_{row}(i_2)$ and $\theta_{row}(i_3)$, or else, the submatrix $\mathcal{X}_4$ occurs. Suppose that $\theta_{row}(i_2) < \theta_{row}(i_3)$ and $\theta_{col}(j_1) < \theta_{col}(j_2)$, then $\mathcal{X}_3$ occurs, a contradiction. The remaining cases are similar.

Note, that a matrix which avoids $\mathcal{X}_4$ has the consecutive ones property in columns. Gusfield [11, Theorem 3] has proven that a matrix which has a phylogenetic tree can be reordered so as to satisfy that property. In fact, the reordering used by Gusfield's proof generates an essentially canonical matrix.

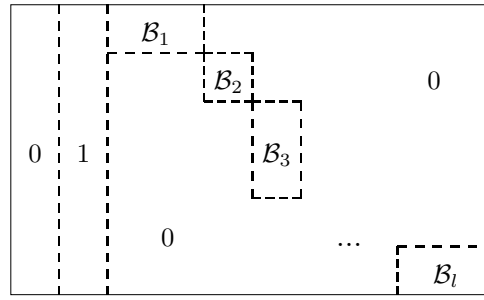


Fig. 4. Recursive definition of canonical matrices. Each $\mathcal{B}_i$ is constructed in the same manner.

Klinz et al. [16] study and review numerous problems of permuting matrices to avoid forbidden submatrices.

4 The Algorithm

Let $\mathcal{A}$ be the input matrix. Define $G^x(\mathcal{A}) = (S, C, E_x^{\mathcal{A}})$ for $x =?, 1$. For a subset $\emptyset \neq S' \subseteq S$, we say that a character is *S'-semi-universal* in $\mathcal{A}$ if its 0-set does not intersect S'.

We now present an algorithm for solving IDP. The algorithm outputs a tree $\mathcal{T}$ which explains $\mathcal{A}$, or outputs *False* if no such tree exists.

1. $G \leftarrow G^1(\mathcal{A})$,$\mathcal{T} \leftarrow triv(S)$.
2. Remove all S-semi-universal and all null characters from G.
3. **While** $E(G) \neq \emptyset$ **do**:
   (a) **For** each connected component K of G such that $|E(K)| \geq 1$ **do**:
      i. Let $\hat{S} \leftarrow S(K)$.
      ii. Compute the set U of all characters in K which are $\hat{S}$-semi-universal in $\mathcal{A}$.
      iii. **If** $U = \emptyset$ **then** output *False* and **halt**.
      iv. **Otherwise**, remove U from G and update $\mathcal{T} \leftarrow \mathcal{T} \cup \{\hat{S}\}$.
4. Output $\mathcal{T}$.

Theorem 3. *The above algorithm correctly solves IDP.*

Proof. Suppose that the algorithm returns *False*. Then there exists an iteration of the 'while' loop at which some connected component K contained no $S(K)$-semi-universal character. Suppose to the contrary that some F^* solves $\mathcal{A}$, and let

$G^* = (S, C, F^*)$. Let H^* be the subgraph of G^* induced by $V(K)$. By definition, H^* is connected and by Theorem 1, it is also Σ-free. Therefore, by Proposition 1 it has an $S(K)$-universal character. That character must be $S(K)$-semi-universal in $\mathcal{A}$, a contradiction.

On the other hand, suppose that the algorithm returns a collection $\mathcal{T}$ of sets. We shall prove that $\mathcal{T}$ is a tree which explains $\mathcal{A}$. We first prove that the collection $\mathcal{T}$ of sets is pairwise compatible, implying, by Observation 1, that $\mathcal{T}$ is a tree. Let S_1, S_2 be two subsets in $\mathcal{T}$. Let t_i denote the iteration of the 'while' loop at which S_i was added to $\mathcal{T}$, for $i = 1, 2$. If $t_1 = t_2$ then S_1 and S_2 are clearly disjoint. Otherwise, suppose w.l.o.g., that $t_1 < t_2$ and $S_1 \cap S_2 \neq \emptyset$. Let K_1 denote the connected component containing S_1 at iteration t_1 of the algorithm. The edge set of G at iteration t_1 contains the one at iteration t_2. Therefore, K_1 contains the vertices in S_2. It follows that $S_1 \supseteq S_2$.

It remains to show that the resulting tree is a phylogenetic tree for $\mathcal{A}$. Suppose that a character c was removed from G as a part of some set U in Step 3(a)iv. Associate with c the clade $\hat{S}$, which was added to $\mathcal{T}$ at that same step. Observe, that each non-trivial clade $\hat{S} \in \mathcal{T}$ is associated with at least one character. Associate with each S-semi-universal character the clade S. Associate with each null character the empty clade. Finally, define a binary matrix $\mathcal{B}_{n \times m}$ with $b_{sc} = 1$ iff s belongs to the clade S_c associated with c. Since $a_{sc} \neq 1$ for all $s \notin S_c$ and $a_{sc} \neq 0$ for all $s \in S_c$, $\mathcal{B}$ is a completion of $\mathcal{A}$. The claim follows.

The algorithm can be naively implemented in $O(hnm)$ time, where $h \leq \min\{m, n\}$ denotes the height of the reconstructed tree. This can be seen by noting that each iteration of the 'while' loop increases the height of the output tree by one. We give a better bound in the following theorem.

Theorem 4. *The complexity of the algorithm is $O(nm+|E_1|\log^2(n+m))$ deterministic time. Alternatively, there exists a randomized algorithm that solves IDP in $O(nm + |E_1|\log(l^2/|E_1|) + l(\log l)^3 \log\log l)$ expected time, where $l = n + m$.*

Proof. Each iteration of the 'while' loop splits the (potential) clades added in the previous one. Thus, the algorithm performs an iteration per level of the tree returned, and at most h iterations. The connected components of G can be initialized in $O(|E_1| + n + m)$ time, and maintained using a dynamic data structure for graph connectivity. Using the dynamic algorithm of [14] the connected components of G can be maintained during $|E_1|$ edge deletions at a cost of $O(|E_1|\log^2(n+m))$ time spent in Step 3(a)iv. Alternatively, using the Las-Vegas type randomized algorithm for decremental dynamic connectivity [22], the edge deletions can be supported in $O(|E_1|\log(l^2/|E_1|) + l(\log l)^3 \log\log l)$ expected time.

The connected components of G must be explicitly recomputed from the dynamic data structure for each iteration. This takes $O(h(m + n)) = O(nm)$ time in total. Since each set added to $\mathcal{T}$ in Step 3(a)iv corresponds to at least one character, and each character is associated with exactly one set, updating $\mathcal{T}$ requires in total $O(nm)$ time.

It remains to show how semi-universal characters can be efficiently found in Step 3(a)ii. Let $G(t)$ denote the graph G at iteration t of the 'while' loop. For

a character c, denote by $d_c^1(t)$ its degree in $G(t)$, and by $d_c^?(t)$ the degree of c in $G^?(\mathcal{A})$ to its connected component K_c in $G(t)$. Given $d_c^1(t)$,$d_c^?(t)$, one can check in $O(1)$ time whether c is $S(K_c)$-semi-universal. $d_c^1(t)$ remains unchanged throughout, and equals $d_c^1(1)$. $d_c^?(t)$ can be maintained as follows. $d_c^?(1)$ is initialized in $O(|E_?|)$ time (given the connected components of $G(1)$). At the beginning of iteration t, $d_c^?(t+1)$ is initialized to $d_c^?(t)$. Each time a connected component K_c of $G(t)$ is split into sub-components $K_1, \ldots, K_l$ due to the removal of characters in Step 3(a)iv, we update $d_c^?(t+1)$ as follows: For $c \in C(K_j)$, we decrease $d_c^?(t+1)$ by $|\{(s,c) \in E_? : s \in S(K_p), p \neq j\}|$. This takes $O(|E_?|)$ time for all c, t. Finally, finding the semi-universal characters over all iterations costs $O(hm)$ time. The complexity follows.

We remark, that an $\Omega(mn)$ time lower bound for (undirected) binary perfect phylogeny was proven by Gusfield [11]. A closer look at Gusfield's proof reveals that it applies, as is, also to the directed case. As IDP generalizes directed binary perfect phylogeny, any algorithm for this problem would require $\Omega(mn)$ time.

5 Concluding Remarks

We have given a polynomial algorithm for reconstructing a phylogeny from incomplete binary directed data, using a graph theoretic reformulation of the problem. The algorithm is near optimal and takes $\tilde{O}(nm)$ time.

An interesting question regarding IDP is whether one can identify if there exists a "most general" solution, so that all others are obtained from it by refinements (additional clades). We have proven that in case a "most general" solution exists, the algorithm described here provides that solution. The full version of this manuscript will include this proof, along with a more general polynomial time algorithm, that also determines if such a solution exists.

Acknowledgments

We thank Dan Graur and Tal Pupko for drawing our attention to this phylogenetic problem, and for helpful discussions. We thank Joe Felsenstein, Dan Gusfield, Haim Kaplan, Mike Steel, and an anonymous CPM '00 referee for referring us to helpful literature.

The first author was supported by the Clore foundation scholarship. The second author was supported in part by the Israel Science Foundation formed by the Israel Academy of Sciences and Humanities. The third author was supported by an Eshkol fellowship from the Ministry of Science, Israel.

References

1. R. Agarwala and D. Fernández-Baca. A polynomial-time algorithm for the perfect phylogeny problem when the number of character states is fixed. *SIAM Journal on Computing*, 23(6):1216–1224, 1994.

2. A. V. Aho, Y. Sagiv, T. G. Szymanski, and J. D. Ullman. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM Journal on Computing*, 10(3):405–421, 1981.
3. C. Benham, S. Kannan, M. Paterson, and T.J. Warnow. Hen's teeth and whale's feet: generalized characters and their compatibility. *Journal of Computational Biology*, 2(4):515–525, 1995.
4. H. L. Bodlaender, M. R. Fellows, and T. J. Warnow. Two strikes against perfect phylogeny. In W. Kuich, editor, *Proc. 19th ICALP*, pages 273–283, Berlin, 1992. Springer. Lecture Notes in Computer Science, Vol. 623.
5. J. H. Camin and R. R. Sokal. A method for deducing branching sequences in phylogeny. *Evolution*, 19:409–414, 1965.
6. L. Dollo. Le lois de l'évolution. *Bulletin de la Societé Belge de Géologie de Paléontologie et d'Hydrologie*, 7:164–167, 1893.
7. J. Felsenstein. *Inferring Phylogenies*. Sinaur Associates, Sunderland, Massachusetts, 2000. In press.
8. L. R. Foulds and R. L. Graham. The Steiner problem in phylogeny is NP-complete. *Advances in Applied Mathematics*, 3:43–49, 1982.
9. M. C. Golumbic. Matrix sandwich problems. *Linear algebra and its applications*, 277:239–251, 1998.
10. M. C. Golumbic, H. Kaplan, and R. Shamir. Graph sandwich problems. *Journal of Algorithms*, 19:449–473, 1995.
11. D. Gusfield. Efficient algorithms for inferring evolutionary trees. *Networks*, 21:19–28, 1991.
12. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
13. M. Henzinger, V. King, and T.J. Warnow. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica*, 24:1–13, 1999.
14. J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge and biconnectivity. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC-98)*, pages 79–89, New York, May 23–26 1998. ACM Press.
15. S. Kannan and T. Warnow. A fast algorithm for the computation and enumeration of perfect phylogenies. *SIAM Journal on Computing*, 26(6):1749–1763, 1997.
16. B. Klinz, R. Rudolf, and G. J. Woeginger. Permuting matrices to avoid forbidden submatrices. *Discrete applied mathematics*, 60:223–248, 1995.
17. C. A. Meecham and G. F. Estabrook. Compatibility methods in systematics. *Annual Review of Ecology and Systematics*, 16:431–446, 1985.
18. M. Nikaido, A. P. Rooney, and N. Okada. Phylogenetic relationships among cetartiodactyls based on insertions of short and long interspersed elements: Hippopotamuses are the closest extant relatives of whales. *Proceedings of the National Academy of Science USA*, 96:10261–10266, 1999.
19. W. J. Le Quesne. The uniquely evolved character concept and its cladistic application. *Systematic Zoology*, 23:513–517, 1974.
20. M. A. Steel. The complexity of reconstructing trees from qualitative characters and subtrees. *Journal of Classification*, 9:91–116, 1992.
21. D. L. Swofford. *PAUP, Phylogenetic Analysis Using Parsimony (and Other Methods)*. Sinaur Associates, Sunderland, Massachusetts, 1998. Version 4.
22. M. Thorup. Decremental dynamic connectivity. *Journal of Algorithms*, 33:229–243, 1999.

The Longest Common Subsequence Problem for Arc-Annotated Sequences

Tao Jiang[1,4⋆], Guo-Hui Lin[2,4⋆⋆], Bin Ma[2⋆⋆⋆], and Kaizhong Zhang[3†]

[1] Department of Computer Science, University of California, Riverside, CA 92521
jiang@cs.ucr.edu
[2] Department of Computer Science, University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
ghlin@math.uwaterloo.ca
[3] Department of Computer Science, University of Western Ontario
London, Ontario N6A 5B7, Canada
b3ma@math.uwaterloo.ca
[4] Department of Computing and Software, McMaster University
Hamilton, Ontario L8S 4L7, Canada
kzhang@csd.uwo.ca

Abstract. Arc-annotated sequences are useful in representing the structural information of RNA and protein sequences. Recently, the longest arc-preserving common subsequence problem has been introduced in [6,7] as a framework for studying the similarity of arc-annotated sequences. In this paper, we consider arc-annotated sequences with various arc structures and present some new algorithmic and complexity results on the longest arc-preserving common subsequence problem. Some of our results answer an open question in [6,7] and some others improve the hardness results in [6,7].
Keywords: Sequence annotation, longest common subsequence, approximation algorithm, maximum independent set, MAX SNP-hard, dynamic programming.

1 Introduction

Given two sequences S and T over some fixed *alphabet* Σ, the sequence T is a *subsequence* of S if T can be obtained from S by deleting some letters from S. Notice that the order of the remaining letters of S must be preserved. The *length* of a sequence S is the number of letters in it and is denoted as $|S|$. Given two sequences S_1 and S_2 (over some fixed *alphabet* Σ), the classical *longest common subsequence (LCS)* problem asks for a longest sequence T that is a subsequence of both S_1 and S_2. Suppose $|S_1| = n$ and $|S_2| = m$, the longest common subsequence of S_1 and S_2 can be found by dynamic programming in time $O(nm)$ [9,13]. For simplicity, we use $S[i]$ to denote the ith letter in sequence

⋆ Supported in part by NSERC Research Grant OGP0046613, a CITO grant, and a UCR startup grant.
⋆⋆ Supported in part by NSERC Research Grant OGP0046613 and a CITO grant.
⋆⋆⋆ Supported in part by NSERC Research Grant OGP0046506 and a CGAT grant.
† Supported in part by NSERC Research Grant OGP0046373.

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 154–165, 2000.

S, and $S[i, j]$ to denote the substring of S consisting of the ith letter through the jth letter.

For any sequence S, an *arc annotation* (or *arc set*) P of S is a set of *unordered* pairs of positions in S. Each pair $(i, j) \in P$, where $1 \leq i < j \leq |S|$, is said to *connect* the two letters at positions i and j and is called an *arc* between the two letters. Such a pair (S, P) of sequence and arc annotation will be referred to as an *arc-annotated* sequence [6]. Observe that a (plain) sequence without any arc annotation can be viewed as an arc-annotated sequence with an empty arc set.

Arc-annotated sequences are useful in describing the secondary and tertiary structures of RNA and protein sequences [1,4,6,8,10,12,14]. For example, one may use arcs to represent bonds between nucleotides in an RNA sequence or contact forces between amino acids in a protein sequence. Therefore, the problem of comparing arc-annotated sequences has applications in the structural comparison of RNA and protein sequences and has received much recent attention in the literature [1,6,8,10,14]. In this paper, we follow the LCS approach proposed in [6] and study the LCS problem for arc annotated sequences.

1.1 The Arc-Annotated LCS Problem and Its Restricted Versions

Given two arc annotated sequences S_1 and S_2 with arc sets P_1 and P_2 respectively, a common subsequence T of S_1 and S_2 induces a bijective mapping from a subset of $\{1, \ldots, |S_1|\}$ to a subset of $\{1, \ldots, |S_2|\}$ (for the detail description, see [6,7]). The common subsequence T is *arc-preserving* if the arcs induced by the mapping are preserved, *i.e.*, for any (i_1, j_1) and (i_2, j_2) in the mapping,

$$(i_1, i_2) \in P_1 \Longleftrightarrow (j_1, j_2) \in P_2.$$

The *longest arc-preserving common subsequence* problem is to find a longest common subsequence of S_1 and S_2 that is arc-preserving (with respect to the given arc sets P_1 and P_2) [6].

It is shown in [6,7] that the (general) longest arc-preserving common subsequence problem is NP-hard, if the arc annotations are unrestricted. Since in the practice of RNA and protein sequence comparison arc sets are likely to satisfy some constraints (*e.g.*, bond arcs do not cross in the case of RNA sequences), it is of interest to consider various restrictions on arc structures. We consider the following four natural restrictions on an arc set P which are first discussed in [6]:

1. no sharing of endpoints:
   $\forall (i_1, i_2), (i_3, i_4) \in P$, $i_1 \neq i_4$, $i_2 \neq i_3$, and $i_1 = i_3 \Longleftrightarrow i_2 = i_4$.
2. no crossing:
   $\forall (i_1, i_2), (i_3, i_4) \in P$, $i_1 \in [i_3, i_4] \Longleftrightarrow i_2 \in [i_3, i_4]$.
3. no nesting:
   $\forall (i_1, i_2), (i_3, i_4) \in P$, $i_1 \leq i_3 \Longleftrightarrow i_2 \leq i_3$.
4. no arcs:
   $P = \emptyset$.

These restrictions are used progressively and inclusively to produce five distinct *levels* of permitted arc structures for the longest arc-preserving common subsequence problem:

- *unlimited* — no restrictions;
- *crossing* — restriction 1;
- *nested* — restrictions 1 and 2;
- *chain* — restrictions 1, 2 and 3;
- *plain* — restriction 4.

1.2 Summary of Results

In the following, we will use the notation LCS(level-1, level-2) to represent the longest arc-preserving common subsequence problem where the arc structure of sequence S_1 is of level level-1 and the arc structure of sequence S_2 is of level level-2. Without loss of generality, we always assume that level-1 is at the same level or higher than level-2. In other words, we assume that the arc structure of sequence S_1 is at least as complex as that of sequence S_2.

For each position of a sequence, the number of arcs crossing it (including the arcs connecting to this position) is the *arc-cutwidth* of the sequence at this position [6]. The *cutwidth* of the sequence is defined to be the maximum arc-cutwidth over all positions. Given two arc-annotated sequences (S_1, P_1) and (S_2, P_2), the maximum of their cutwidths is defined as the cutwidth of the longest arc-preserving common subsequence problem concerning (S_1, P_1) and (S_2, P_2). Some of our results are concerned with sequences with bounded cutwidth.

The following table summarizes the algorithmic and complexity results on the longest arc-preserving common subsequence problem obtained in [6,7] and this paper. Denote $|S_1| = n$ and $|S_2| = m$.

<table>
<tr><th></th><th>plain</th><th>chain</th><th>nested</th><th>crossing</th><th>unlimited</th></tr>
<tr><td>unlimited</td><td colspan="5">NP-hard [6,7]
not approximable within ratio n^{ϵ}, $\epsilon \in (0, \frac{1}{4})$</td></tr>
<tr><td>crossing</td><td colspan="4">NP-hard [6,7]
MAX SNP-hard
admits a 2-approximation</td><td></td></tr>
<tr><td>nested</td><td colspan="2">$O(nm^3)$</td><td colspan="2">complexity open
admits a 2-approximation</td><td></td></tr>
<tr><td>chain</td><td colspan="2">$O(nm)$ [6,7]</td><td colspan="3"></td></tr>
<tr><td>plain</td><td colspan="2">$O(nm)$ [9,13]</td><td colspan="3"></td></tr>
</table>

Fig. 1. Results on LCS(level-1, level-2).

Our results on problems LCS(nested, chain) and LCS(nested, plain) in fact answer an open question in [6,7].

The rest of the paper is organized as follows. In Section 2, we design the 2-approximation algorithm for problem LCS(crossing, crossing), which implies that problems LCS(crossing, nested), LCS(crossing, chain), LCS(crossing, plain), and LCS(nested, nested) all admit 2-approximation algorithms. In Section 3, we show that problem LCS(crossing, plain) is MAX SNP-hard. This implies that problems LCS(crossing, chain), LCS(crossing, nested) and LCS(crossing, crossing) are also MAX SNP-hard. This excludes the possibility for any of these problems to have a polynomial time approximation scheme (PTAS). In Section 4, we first design a dynamic programming algorithm to solve problem LCS(nested, plain) and then extend it to problem LCS(nested, chain) and to problem LCS(crossing, nested) where the first sequence has a bounded cutwidth. Some concluding remarks are given in Section 5.

2 Approximation Algorithms

The NP-hardness of problem LCS(crossing, plain) [6,7] implies that problem LCS(level-1, level-2) is NP-hard whenever an input arc-annotated sequence has an unlimited or crossing arc structure. In the following, we give a 2-approximation algorithm for problem LCS(crossing, crossing), which also implies 2-approximation algorithms for problem LCS(crossing, nested), problem LCS(crossing, chain), and problem LCS(crossing, plain).

Let (S_1, P_1) and (S_2, P_2) be a pair of arc-annotated sequences. The algorithm begins with the classical dynamic programming for the classical LCS problem for the two plain sequences S_1 and S_2. Let Y denote the subsequence achieved and M denote the mapping between positions in S_1 and S_2 induced by Y. Assume that $|Y| = L$ and $M = \{(i_1, j_1), \ldots, (i_L, j_L)\}$. We note that M is not necessarily arc-preserving so far.

We now construct a graph G_M, where each match (i_k, j_k) represents a vertex and two vertices (i_k, j_k) and (i_l, j_l) are connected by an edge if and only if either $(i_k, i_l) \in P_1$ or $(j_k, j_l) \in P_2$, but not both. Notice that the maximum degree of any vertex in G_M is at most 2. Consider a connected component of G_M. If it contains more than one vertices, then it must be either a simple path or a simple cycle. It follows that we may delete at most half of the vertices in the component to make each of the remaining vertices isolated. Repeating this deletion procedure for all connected components containing more than one vertices until we get a graph G'_M, in which every component is a singleton. Let M' denote the subset of matches corresponding to those remaining vertices in G'_M. It follows that $|M'| \geq |M|/2$. Clearly, the subsequence inducing M' (which is obtained by deleting those letters corresponding to the deleted matches), denoted by Y', is an arc-preserving common subsequence. The high level description of the algorithm, called Algorithm $\mathcal{MLCS}$, is depicted in Figure 2.

Since $|Y'| \geq |Y|/2$ and $|Y|$ is an upper bound for the length of a longest arc-preserving common subsequence for (S_1, P_1) and (S_2, P_2), the above algorithm is a 2-approximation for LCS(crossing, crossing). Hence, we have the following theorem.

Algorithm $\mathcal{MLCS}$

Input: An instance of problem LCS(crossing, crossing): (S_1, P_1) and (S_2, P_2).
Output: an arc-preserving common subsequence.

1. Compute an LCS Y for (S_1, S_2) using dynamic programming.
   Let M denote the mapping induced by Y.
2. Construct the graph G_M for M.
3. **For** each connected component C containing more than one vertices
3.1 Delete (at most) half of the vertices therein to isolate the remaining vertices.
4. Let G'_M be the final graph and M' the set of matches corresponding to the vertices in G'_M.
5. Construct Y', the subsequence corresponding to mapping M'.
6. Output Y' as the solution subsequence.

Fig. 2. The algorithm $\mathcal{MLCS}$.

Theorem 1. *Problem* LCS(crossing, crossing) *has a* 2*-approximation algorithm with time complexity* $O(nm)$.

The following corollary is straightforward.

Corollary 1. *Problem* LCS(crossing, nested)*, problem* LCS(crossing, chain)*, and problem* LCS(crossing, plain) *have* 2*-approximation algorithms running in* $O(nm)$ *time.*

3 Inapproximability Results

In this section, we show that (1) problem LCS(unlimited, plain) cannot be approximated within ratio n^ϵ for any $\epsilon \in (0, \frac{1}{4})$, where n denotes the length of the longer input sequence and (2) problem LCS(crossing, plain) is MAX SNP-hard. To do that, we need the following well known problems.

Maximum Independent Set-B (MaxIS-B): Given a graph G in which every vertex has a degree at most B, find the largest independent set of G, *i.e.*, a subset of vertices in which no two vertices are connected in graph G.

Maximum Independent Set-Cubic (MaxIS-Cubic): Given a cubic graph G (*i.e.*, every vertex has a degree 3), find a largest independent set of G.

Lemma 1. [11,3] MaxIS-B *is MAX SNP-complete when* $B \geq 3$.

The following lemma is necessary.

Lemma 2. MaxIS-Cubic *is MAX SNP-complete.*

Proof. The proof will be done by constructing an L-reduction [11] from MaxIS-3 to MaxIS-Cubic, using a technique from [5]. Given an instance of MaxIS-3, which is a graph $G(V, E)$ with $|V| = n$, we may assume that G is connected. Suppose that there are i vertices of degree 1, j vertices of degree 2 and $n - i - j$ vertices

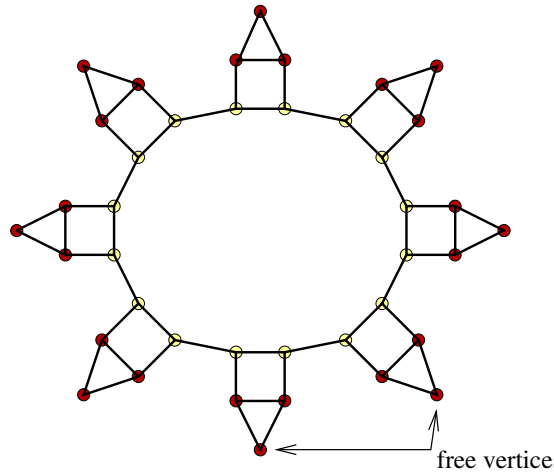


Fig. 3. The graph H.

of degree 3. Let $opt(G)$ denote the size of a maximum independent set of G, then it follows trivially that $opt(G) \geq n/4$. Therefore, we have

$$i + j \leq 4 \cdot opt(G).$$

We construct an instance of MaxIS-Cubic, a graph denoted by G', as follows. Construct $2i + j$ triangles of which each has two vertices connecting to two adjacent vertices in a cycle of size $2(2i + j)$ as shown in Figure 3. Denote this graph as H. Call the vertex of each triangle that is not adjacent to the cycle a *free* vertex in H. Clearly, the cycle itself has a maximum independent set of size $2i + j$. Since for each triangle the maximum independent set is a singleton, graph H has a maximum independent set of size exactly $2(2i + j)$. Moreover, we see that there is a maximum independent set of size $2(2i + j)$ for graph H consisting of no free vertices. The graph G' is then constructed from graphs G and H by connecting each degree 1 vertex in G to two unique free vertices in H and connecting each degree 2 vertex in G to one unique free vertex in H. Notice that resulting graph G' is indeed a cubic graph.

Suppose that V' is a maximum independent set for G (of size $opt(G)$), we may add a maximum independent set for H consisting of no free vertices into it to form an independent set for graph G'. This independent set would have a size of $k' = opt(G) + 2(2i + j)$. It follows that

$$opt(G') \geq opt(G) + 2(2i + j). \tag{1}$$

On the other hand, let V'' be an independent set for G' with $|V''| = k'$. Then deleting from V'' the vertices which are in graph H (the number of such vertices is at most $2(2i + j)$) will form an independent set for G of size k satisfying

$$k \geq k' - 2(2i + j). \tag{2}$$

Therefore, (1) becomes

$$opt(G') = opt(G) + 2(2i + j) \leq opt(G) + 4(i + j) \leq 17 \cdot opt(G), \tag{3}$$

and (2) becomes $k - opt(G) \geq k' - opt(G')$, or equivalently,

$$|k - opt(G)| \leq |k' - opt(G')|. \tag{4}$$

This completes the L-reduction.

Since problem MaxIS-Cubic is a special case of MaxIS-3, it is in the class MAX SNP. Hence, it is MAX SNP-complete. □

Theorem 2. *Problem* LCS(unlimited, plain) *cannot be approximated within ratio* n^ϵ *for any* $\epsilon \in (0, \frac{1}{4})$*, where* n *is the length of the longer input sequence.*

Proof. We will show that MaxIS, the unrestricted *Maximum Independent Set* problem, can be L-reduced to problem LCS(unlimited, plain). Since MaxIS cannot be approximated within ratio n^ϵ for any $\epsilon \in (0, \frac{1}{4})$ [2], where n is the number of vertices in the graph, the theorem follows.

Let graph $G(V, E)$, where $V = \{1, \ldots, n\}$, be an instance of MaxIS. Without loss of generality, we assume that the graph G is connected. The instance I of LCS(unlimited, plain) consists of two sequences defined as follows: $S_1 = \mathsf{a}^n$ with $P_1 = E$ and $S_2 = \mathsf{a}^n$ with $P_2 = \emptyset$. It is clear then that an independent set $V' = \{v_{i_1}, \ldots, v_{i_k}\}$ one-to-one corresponds to a arc-preserving common subsequence consisting of the i_1th, ..., i_kth a's from sequence S_1. That is, LCS(unlimited, plain) in fact includes MaxIS as a subproblem. □

As a corollary, we have

Corollary 2. *Problem* LCS(unlimited, chain)*, problem* LCS(unlimited, nested)*, and problem* LCS(unlimited, unlimited) *cannot be approximated within ratio* n^ϵ *for any* $\epsilon \in (0, \frac{1}{4})$*, where* n *is the length of the longer input sequence.*

Theorem 3. *Problem* LCS(crossing, plain) *is MAX SNP-hard.*

Proof. To show its MAX SNP-hardness, we will L-reduce MaxIS-Cubic to problem LCS(crossing, plain). Given a cubic graph $G(V, E)$, let $n = |V|$. For each vertex $u \in V$, construct a segment T_u of letters aaaabbccc. Sequence S_1 is obtained by concatenating the n segments $T_u, u \in V$. The arc set P_1 on sequence S_1 is constructed as follows: Whenever there is an edge $(u, v) \in E$, introduce an arc between a letter c from T_u to a letter c from T_v, ensuring that each letter c is used only once. Sequence S_2 is obtained by concatenating n identical segments of aaaacccbb and its arc set $P_2 = \emptyset$. This constitutes an instance I of problem LCS(crossing, plain). See Figure 4 for an illustration.

Suppose that graph G has a maximum independent set V' with cardinality k. We put four letters a and three letters c from each $T_u, u \in V'$ into sequence Y and put four letters a and two letters b from each $T_u, u \notin V'$ into sequence Y. Clearly, sequence Y is a common subsequence of both S_1 and S_2. From the independency of V', we know that sequence Y inherits no arcs from P_1, and thus it is an arc-preserving common subsequence. Let $opt(I)$ denote the length of an

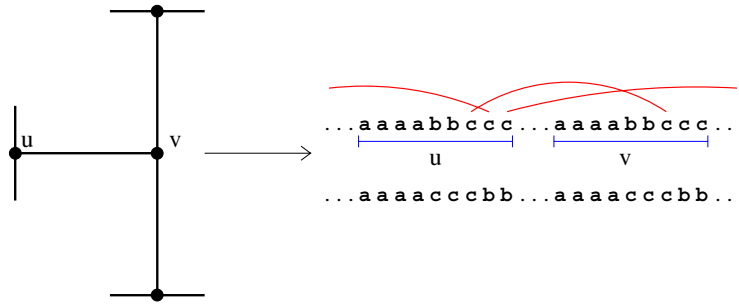


Fig. 4. Instance I constructed from cubic graph G.

longest arc-preserving common subsequence of S_1 and S_2, and $opt(G)$ denote the cardinality of a maximum independent set of graph G. It follows from the above that

$$opt(I) \geq opt(G) + 6n. \tag{5}$$

On the other hand, suppose that we have an arc-preserving common subsequence Y of length k' for (S_1, P_1) and (S_2, P_2). We observe first that Y inherits no arcs from P_1 since $P_2 = \emptyset$. Define a match between letters from the two segments at the same position in S_1 and S_2, *i.e.*, $S_1[9l+1, 9l+9]$ and $S_2[9l+1, 9l+9]$ for some $1 \leq l \leq n$, to be *canonical*. It is easy to see that the abundance of letters a in both S_1 and S_2 allow us to consider only common subsequences Y that induce mappings M consisting of only canonical matches. *I.e.*, we assume that for each $(i, j) \in M$, $i, j \in [9l + 1, 9l + 9]$ for some l.

Consider the canonical matches defined by the mapping M (induced by Y) between segments $S_1[9l+1, 9l+9]$ and $S_2[9l+1, 9l+9]$ for every l. Clearly, (i) the four a's should be matched and (ii) if a b is matched then no c's can be matched and vice versa. If only one or two c's are matched, we modify M (and thus Y) as follows: unmatch the c's and match the two b's. Clearly, this modification doesn't make the solution worse. We repeat this modification until the canonical matches between each segment pair contains either all three c-matches or two b-matches. Denote the final mapping as M'.

We now define a subset V' of vertices of G as follows: for every segment T_u in sequence S_1, if all its three c's are matched M', we put u in V'. By the construction of arc set P_1, no pair of vertices in V' are connected in graph G and hence V' is an independent set for G. Let $k = |V'|$, then we have $k \geq k' - 6n$. Since G is a cubic graph, $n/4 \leq opt(G) \leq n/2$ and inequality (5) becomes

$$opt(I) = opt(G) + 6n \leq 25 \cdot opt(G). \tag{6}$$

Hence $k \geq k' - 6n = k' - (opt(I) - opt(G))$, which is equivalent to

$$|k - opt(G)| \leq |k' - opt(I)|. \tag{7}$$

Inequalities (6) and (7) show that our reduction is an L-reduction, and thus problem LCS(crossing, plain) is MAX SNP-hard. □

As a corollary, we have

Corollary 3. *Problem* LCS(crossing, chain)*, problem* LCS(crossing, nested)*, and problem* LCS(crossing, crossing) *are all MAX SNP-hard.*

We note in passing that there is a similar, but more restrictive, LCS definition [14], where in addition to our condition that, for any (i_1, j_1) and (i_2, j_2) in the mapping, $(i_1, i_2) \in P_1$ if and only if $(j_1, j_2) \in P_2$, it also requires that for any (i_1, j_1) in the mapping, if $(i_1, i_2) \in P_1$ then, for some j_2, (i_2, j_2) is in the mapping, and if $(j_1, j_2) \in P_2$ then, for some i_2, (i_2, j_2) is in the mapping. For that definition LCS(crossing, crossing) is NP-hard and LCS(crossing, nested) is solvable in polynomial time [14].

4 Dynamic Programming Algorithm for LCS(nested, plain)

Given a pair (S_1, P_1) and $(S_2, \emptyset)$ of arc-annotated sequences with P_1 being nested, denote $n = |S_1|$ and $m = |S_2|$. For any arc $u \in P_1$, let u_l and u_r denote its *left* and *right* endpoints, respectively. We also use $u(i)$ to denote the arc in P_1 incident on position i of sequence S_1. If $u(i)$ does not exist, then we call position i *free*. We formulate a recurrence relation for computing the length of a longest arc-preserving common subsequence for the pair (S_1, P_1) and $(S_2, \emptyset)$.

Let $DP(i, i'; j, j')$, where $1 \le i \le i' \le n$ and $1 \le j \le j' \le m$, denote the length of a longest arc-preserving common subsequence for the pair $(S_1[i, i'], P_1[i, i'])$ and $(S_2[j, j'], \emptyset)$. Define $\chi(S_1[i], S_2[j]) = 1$ if $S_1[i] = S_2[j]$, or 0 otherwise. Our algorithm is a two-step dynamic programming.

The first step of our algorithm is as follows:

For each arc $(i_1, j_1) \in P_1$, we perform the following computation.

Phase 1: If $j_1 - i_1 > 1$, then let $i = i_1 + 1$ and let i' be a position in $[i, j_1 - 1]$, such that for any arc $(k, k') \in P_1$, either $[i, i'] \subseteq [k, k']$, or $[k, k'] \subseteq [i, i']$, or $[i, i'] \cap [k, k'] = \emptyset$.

If i' is free, then

$$DP(i, i'; j, j') = \max \begin{cases} DP(i, i' - 1; j, j' - 1) + \chi(S_1[i'], S_2[j']), \\ DP(i, i' - 1; j, j'), \\ DP(i, i'; j, j' - 1). \end{cases}$$

If i' is not free (i.e., $i' = u(i')_r$) and $i \neq u(i')_l$, then

$$DP(i, i'; j, j') = \max_{j \le j'' \le j'} \Big\{ DP(i, u(i')_l - 1; j, j'' - 1) + DP(u(i')_l, i'; j'', j') \Big\}.$$

Phase 2:

$$DP(i_1, j_1; j, j') = \max \begin{cases} DP(i_1 + 1, j_1 - 1; j + 1, j') + \chi(S_1[i_1], S_2[j]), \\ DP(i_1 + 1, j_1 - 1; j, j' - 1) + \chi(S_1[j_1], S_2[j']), \\ DP(i_1 + 1, j_1 - 1; j, j'), \\ DP(i_1, j_1; j, j' - 1), \\ DP(i_1, j_1; j + 1, j'). \end{cases}$$

Note that for any two arcs (i_1, j_1) and (i_2, j_2) in P_1, let s_1 and s_2 be the sets of i' satisfying conditions of *Phase 1*, then s_1 and s_2 are disjoint. Therefore the total number of entries for $DP(i, i'; j, j')$ is $O(nm^2)$. This means that the first step can be done in (nm^3).

The second step of the algorithm is very similar to *Phase 1* of the first step.

Let i' be any position in $[1, n]$, such that for any arc $(k, k') \in P_1$, either $[1, i'] \subseteq [k, k']$, or $[k, k'] \subseteq [i, i']$, or $[1, i'] \cap [k, k'] = \emptyset$.

If i' is free, then

$$DP(1, i'; j, j') = \max \begin{cases} DP(1, i' - 1; j, j' - 1) + \chi(S_1[i'], S_2[j']), \\ DP(1, i' - 1; j, j'), \\ DP(1, i'; j, j' - 1). \end{cases}$$

If i' is not free (i.e., $i' = u(i')_r$) and $u(i')_l \neq 1$, then

$$DP(1, i'; j, j') = \max_{j \leq j'' \leq j'} \Big\{ DP(1, u(i')_l - 1; j, j'' - 1) + DP(u(i')_l, i'; j'', j') \Big\}.$$

It is clear that this step can be done in $O(nm^3)$ too. Therefore the time complexity of the algorithm is $O(nm^3)$. The value $DP(1, n; 1, m)$, which is computed either in *Phase 2* of the first step, or in the second step, is the length of the longest arc-preserving common subsequence. By using a standard back-tracing technique, we can find a longest arc-preserving common subsequence for the pair (S_1, P_1) and $(S_2, \emptyset)$ from the table entries. Therefore, we have the following theorem, which answers an open question in [6,7].

Theorem 4. *Problem* LCS(nested, plain) *can be solved in time* $O(nm^3)$.

We may extend the above construction to solve problem LCS(nested, chain) as well as to problem LCS(crossing, nested) in which the cutwidth of the first sequence is upper bounded by a constant k. For example, for problem LCS(nested, chain), we expand the entry $DP(i, i'; j, j')$ to entry $DP(i, i'; j, j'; \alpha, \beta)$, where α and β denote the positions between the jth and j'th on the second sequence $S_2[j, j']$ that the letters therein cannot be matched to any letter in the first sequence $S_1[i, i']$. More specifically, for a given pair of j and j', there might be an arc in P_2 crossing (not connecting to) position j (j', respectively) and its right (left, respectively) endpoint lying inside $[j + 1, j' - 1]$. Since we want to compute the arc-preserving common subsequence of $S_1[i, i']$ and $S_2[j, j']$ independently, we have to register if this endpoint can be included into the common subsequence. If it cannot, we will then let α (β, respectively) denote this right (left, respectively) endpoint of the arc crossing position j (j', respectively). In the other cases, α (β, respectively) denotes nothing (represented by $-$). The recurrence relation needs modifications to fit the computation. For example, in *Phase 1* and i' is free, let ρ denote the rightmost position in $S_2[j, j' - 1]$ that is not α, neither β. If $j' = u(j')_l$ then we need to compute

$$DP(i, i'; j, j'; \alpha, -) = \max \begin{cases} DP(i, i' - 1; j, \rho; \alpha', -) + \chi(S_1[i'], S_2[j']), \\ DP(i, i' - 1; j, j'; \alpha, -), \\ DP(i, i'; j, \rho; \alpha', -), \end{cases}$$

and

$$DP(i, i'; j, j'; \alpha, j') = DP(i, i'; j, \rho; \alpha', -),$$

where $\alpha' = \alpha$ if $\alpha < \rho$, $-$ otherwise;

If $j' = u(j')_r (\neq \alpha)$ then we need to compute

$$DP(i, i'; j, j'; \alpha, -) = \max \begin{cases} DP(i, i'-1; j, \rho; \alpha, \beta') + 1, \text{ if } S_1[i'] = S_2[j'], \\ DP(i, i'-1; j, j'; \alpha, -), \\ DP(i, i'; j, j'-1; \alpha, -), \end{cases}$$

where $\beta' = u(j')_l$ if $j \leq u(j')_l < \rho$, $-$ otherwise; and

$$DP(i, i'; j, j'; \alpha, j') = DP(i, i'; j, j'-1; \alpha, -);$$

In the other cases, we need to compute

$$DP(i, i'; j, j'; \alpha, \beta) = \max \begin{cases} DP(i, i'-1; j, \rho; \alpha', \beta') + \chi(S_1[i'], S_2[j']), \\ DP(i, i'-1; j, j'; \alpha, \beta), \\ DP(i, i'; j, \rho; \alpha', \beta'), \end{cases}$$

where $\alpha' = \alpha$ if $\alpha < \rho$, $-$ otherwise; $\beta' = \beta$ if $\beta < \rho$, $-$ otherwise.

The case in which i' is not free, the *Phase 2* and the second step of the algorithm can be similarly modified. Notice that each entry $DP(i, i'; j, j')$ is expanded to at most four entries. Again, each entry can be computed in $O(m)$ time from its "preceding" entries. It follows that problem LCS(nested, chain) can be solved in time $O(nm^3)$. For problem LCS(crossing, nested), each entry $DP(i, i'; j, j')$ is expanded by introducing k pairs of parameters, $\alpha_1, \beta_1, \alpha_2, \beta_2, \cdots, \alpha_k, \beta_k$, to denote the $2k$ possible positions in sequence $S_1[i, i']$ that the letters therein cannot be matched to any letter in sequence $S_2[j, j']$. These expanded entries can be used to solve problem LCS(crossing, nested) where the cutwidth of the first input sequence is upper bounded by a constant k.

Theorem 5. *Problem* LCS(nested, chain) *can be solved in time* $O(nm^3)$. *The restricted version of problem* LCS(crossing, nested) *where the cutwidth of the first input sequence is upper bounded by a constant* k *can be solved in time* $O(4^k nm^3)$.

We note in passing that the restricted version of problem LCS(crossing, crossing) where both input sequences have cutwidth upper bounded by k is solvable in time $O(9^k nm)$ [6].

5 Concluding Remarks

In this paper, we have considered the longest common subsequence problem for arc-annotated sequences with arc structures of various types, and presented some new algorithmic and complexity results. We leave as an open problem to determine the computational complexity of problem LCS(nested, nested).

Acknowledgment. We thank Professor Paul Kearney for many helpful discussions.

References

1. V. Bafna, S. Muthukrishnan and R. Ravi, Computing similarity between RNA strings, *DIMACS Technical Report* 96-30, 1996.
2. M. Bellare, O. Goldreich and M. Sudan, Free bits, PCPs and non-approximability - towards tight results, *SIAM Journal on Computing*, 27(1998), 804-915.
3. P. Berman and T. Fujito, On approximation properties of the independent set problem for degree 3 graphs, in *Proceedings of 4th International Workshop on Algorithms and Data Structures (WADS'95)*, LNCS 955, pp. 449–460.
4. F. Corpet and B. Minchor, RNAling program: alignment of RNA sequences using both primary and secondary structures, *Computer Applications in the Bio-sciences*, 10(1994), 389–399.
5. D.Z. Du, B. Gao and W. Wu, A special case for subset interconnection designs, *Discrete Applied Mathematics*, 78(1997), 51–60.
6. P.A. Evans, *Algorithms and Complexity for Annotated Sequence Analysis*, Ph. D Thesis, University of Victoria, 1999.
7. P.A. Evans, Finding common subsequences with arcs and pseudoknots, in *Proceedings of 10th Annual Symposium on Combinatorial Pattern Matching (CPM'99)*, LNCS 1645, pp. 270–280.
8. D. Goldman, S. Istrail and C.H. Papadimitriou, Algorithmic aspects of protein structure similarity, *Proc. IEEE 40th Annual Conference of Foundations of Computer Science (FOCS'99)*, 1999.
9. D.S. Hirschberg, The longest common subsequence problem. *Ph.D. Thesis*, Princeton University, 1975.
10. H. Lenhof, K. Reinert and M. Vingron, A polyhedral approach to RNA sequence structure alignment, in *Proceedings of the Second Annual International Conference on Computational Molecular Biology (RECOMB'98)*, 153–159.
11. C.H. Papadimitriou and M. Yannakakis, Optimization, approximation, and complexity classes, *Journal of Computer and System Science*, 43(1991), 425–440.
12. D. Sankoff, Simultaneous solution of the RNA folding, alignment, and protosequence problems, *SIAM Journal on Applied Mathematics*, 45(1985), 810–825.
13. R.A. Wagner and M.J. Fischer, The string-to-string correction problem. Journal of the ACM, 21(1)(1974), 168-173.
14. K. Zhang, L. Wang and B. Ma, Computing similarity between RNA structures, in *Proceedings of 10th Annual Symposium on Combinatorial Pattern Matching (CPM'99)*, LNCS 1645, pp. 281–293.

Boyer–Moore String Matching over Ziv-Lempel Compressed Text

Gonzalo Navarro[1⋆] and Jorma Tarhio[2⋆⋆]

[1] Dept. of Computer Science, University of Chile
gnavarro@dcc.uchile.cl
[2] Dept. of Computer Science, University of Joensuu, Finland
tarhio@cs.joensuu.fi

Abstract. We present a Boyer–Moore approach to string matching over LZ78 and LZW compressed text. The key idea is that, despite that we cannot exactly choose which text characters to inspect, we can still use the characters explicitly represented in those formats to shift the pattern in the text. We present a basic approach and more advanced ones. Despite that the theoretical average complexity does not improve because still all the symbols in the compressed text have to be scanned, we show experimentally that speedups of up to 30% over the fastest previous approaches are obtained. Moreover, we show that using an encoding method that sacrifices some compression ratio our method is twice as fast as decompressing plus searching using the best available algorithms.

1 Introduction

The *string matching problem* is defined as follows: given a pattern $P = p_1 \dots p_m$ and a text $T = t_1 \dots t_u$, find all the occurrences of P in T, i.e. return the set $\{|x|,\ T = xPy\}$. The complexity of this problem is $O(u)$ in the worst case and $O(u \log_\sigma(m)/m)$ on average (where σ is the size of the alphabet Σ), and there exist algorithms achieving both time complexities using $O(m)$ extra space [3,6].

A particularly interesting case of string matching is related to text compression. Text compression [4] tries to exploit the redundancies of the text to represent it using less space. There are many different compression schemes, among which the Ziv-Lempel family [23,24] is one of the best in practice because of their good compression ratios combined with efficient compression and decompression time.

The *compressed matching problem* was first defined in the work of Amir and Benson [1] as the task of performing string matching in a compressed text without decompressing it. Given a text T, a corresponding compressed string $Z = z_1 \dots z_n$, and a pattern P, the compressed matching problem consists in

⋆ Work developed during postdoctoral stay at the University of Helsinki, partially supported by the Academy of Finland and Fundación Andes. Also supported by Fondecyt grant 1-990627.
⋆⋆ Supported in part by the Academy of Finland.

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 166–180, 2000.

finding all occurrences of P in T, using only P and Z. A naive algorithm, which first decompresses the string Z and then performs standard string matching, takes time $O(m+u)$. An optimal algorithm takes worst-case time $O(m+n+R)$, where R is the number of matches (note that it could be that $R = u > n$).

The compressed matching problem is important in practice. Today's textual databases are an excellent example of applications where both aspects of the problem are crucial: the texts should be kept compressed to save space and I/O time, and they should be efficiently searched. These two combined requirements are not easy to achieve together, as the only solution before the 90's was to process queries by uncompressing the texts and then searching into them.

There exist a few works about searching on compressed text, which we cover in the next section. The most promising ones run over the LZ78/LZW variants of the LZ family. They have achieved a good $O(m^2 + n + R)$ worst case search time, and there exist practical implementations able to search in less time than that needed for decompression plus searching. All those works have concentrated in the worst case.

However, Boyer–Moore type techniques, which are able to skip some characters in the text, have never been explored for searching compressed text. Our work points in this direction. We present an application of Boyer–Moore techniques for string matching over LZ78/LZW compressed texts. The resulting algorithms are $\Omega(n)$ time on average, $O(mu)$ time the worst case, and $O(n)$ extra space. This does not improve the existing complexities, but they are faster in practice than all previous work for $m \geq 15$, taking up to 30% less time than the fastest existing implementation. We also present experiments showing that, using an LZ78 encoder that sacrifices some compression ratio for decompression speed, our algorithms are twice as fast as a decompression followed by a search using the best algorithms for both tasks.

2 Related Work

Two different approaches exist to search compressed text. The first one is rather practical. Efficient solutions based on Huffman coding [10] on words have been presented in [16], but they need that the text contains natural language and is large (say, 10 Mb or more). Moreover, they allow only searching for whole words and phrases. There are also other practical ad-hoc methods [15], but the compression they obtain is poor. Moreover, in these compression formats $n = \Theta(u)$, so the speedups can only be measured in practical terms.

The second line of research considers Ziv-Lempel compression, which is based on finding repetitions in the text and replacing them with references to similar strings previously appeared. LZ77 [23] is able to reference any substring of the text already processed, while LZ78 [24] and LZW [20] reference only a single previous reference plus a new letter that is added. String matching in Ziv-Lempel compressed texts is much more complex, since the pattern can appear in different forms across the compressed text. The first algorithm for exact searching is from

1994 [2], which searches in LZ78 needing time and space $O(m^2 + n)$ for the existence problem.

The only search technique for LZ77 [7] is a randomized algorithm to determine in time $O(m + n \log^2(u/n))$ whether a pattern is present or not in the text. It seems that with $O(R)$ extra time both [2] and [7] could find all the R occurrences of the pattern.

An extension of [2] to multipattern searching was presented in [13], together with the first experimental results in this area. They achieve $O(m^2 + n)$ time and space, although this time m is the total length of all the patterns. Other algorithms for different specific search problems have been presented in [8,11].

New practical results appeared in [17], who presented a general scheme to search on Ziv-Lempel compressed texts (simple and extended patterns) and specialized it for the particular cases of LZ77, LZ78 and a new variant proposed which was competitive and convenient for search purposes. A similar result, restricted to the LZW format, was independently found and presented in [14]. Finally, [12] generalized the existing algorithms and nicely unified the concepts in a general framework.

3 Basic Concepts

3.1 The Ziv-Lempel Compression Formats LZ78 and LZW

The general idea of Ziv-Lempel compression is to replace substrings in the text by a pointer to a previous occurrence of them. If the pointer takes less space than the string it is replacing, compression is obtained. Different variants over this type of compression exist, see for example [4]. We are particularly interested in the LZ78/LZW format, which we describe in depth (this is taken from [17]).

The Ziv-Lempel compression algorithm of 1978 (usually named LZ78 [24]) is based on a dictionary of blocks, in which we add every new block computed. At the beginning of the compression, the dictionary contains a single block b_0 of length 0. The current step of the compression is as follows: if we assume that a prefix $T_{1...j}$ of T has been already compressed in a sequence of blocks $Z = b_1 \ldots b_r$, all them in the dictionary, then we look for the longest prefix of the rest of the text $T_{j+1...u}$ which is a block of the dictionary. Once we found this block, say b_s of length ℓ_s, we construct a new block $b_{r+1} = (s, T_{j+\ell_s+1})$, we write the pair at the end of the compressed file Z, i.e $Z = b_1 \ldots b_r b_{r+1}$, and we add the block to the dictionary. It is easy to see that this dictionary is prefix-closed (i.e. any prefix of an element is also an element of the dictionary) and a natural way to represent it is a trie.

We give as an example the compression of the word *ananas* in Figure 1. The first block is $(0, a)$, and next $(0, n)$. When we read the next a, a is already the block 1 in the dictionary, but an is not in the dictionary. So we create a third block $(1, n)$. We then read the next a, a is already the block 1 in the dictionary, but as do not appear. So we create a new block $(1, s)$.

The compression algorithm is $O(u)$ in the worst case and efficient in practice if the dictionary is stored as a trie, which allows rapid searching of the new text

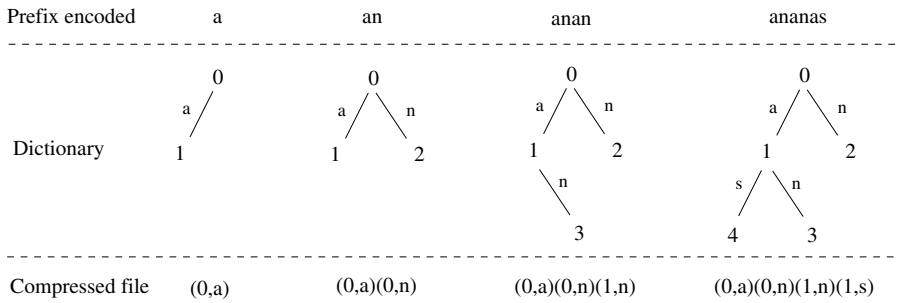


Fig. 1. Compression of the word *ananas* with the algorithm LZ78.

prefix (for each character of T we move once in the trie). The decompression needs to build the same dictionary (the pair that defines the block r is read at the r-th step of the algorithm), although this time it is not convenient to have a trie, and an array implementation is preferable. Compared to LZ77, the compression is rather fast but decompression is slow.

Many variations on LZ78 exist, which deal basically with the best way to code the pairs in the compressed file, or with the best way to cope with limited memory for compression. A particularly interesting variant is from Welch, called LZW [20]. In this case, the extra letter (second element of the pair) is not coded, but it is taken as the first letter of the next block (the dictionary is started with one block per letter). LZW is used by Unix's *Compress* program.

In this paper we do not consider LZW separately but just as a coding variant of LZ78. This is because the final letter of LZ78 can be readily obtained by keeping count of the first letter of each block (this is copied directly from the referenced block) and then looking at the first letter of the next block.

3.2 Boyer–Moore String Matching

The Boyer–Moore (BM) family of text searching algorithms proceed by sliding a *window* of length m over the text. The window is a potential occurrence of the pattern in the text. The text inside the window is checked against the pattern normally from right to left (although not always). If the whole window matches then an occurrence is reported. To shift the window, a number of criteria are used, which try to balance between the cost to compute the shift and the amount of shifting obtained. Two main techniques are used:

Occurrence Heuristic: Pick a character in the window and shift the window forward the minimum necessary to align the selected text character with the same character in the pattern. Horspool [9] uses the m-th window character and Sunday [19] the $(m+1)$-th (actually outside the window). These methods need a table d that for each character gives its last occurrence in the pattern (the details depend on the versions). The Simplified BM (SBM) method [5] uses the character at the position that failed while checking the window, which needs a larger table indexed by window position and character.

Match Heuristic: If the pattern was compared from right to left, some part of it has matched the text in the window, so we precompute the minimum shift necessary to align the part that matched again with the pattern. This requires a table of size m that for each pattern position gives that last occurrence of $P_{i...m}$ in $P_{1...m-1}$. This is used in the original Boyer and Moore method [5].

4 A Simple Boyer–Moore Technique

Consider Figure 2, where we have plotted a hypothetical window approach to a text compressed using LZ78. Each LZ78 block is formed by a line and a final box. The box represents the final *explicit* character c of the block $b = (s, c)$, while the line represents the *implicit* characters, i.e. a text that has to be obtained by resorting to previous referenced blocks (s, then the block referenced by s, and so on).

Fig. 2. A window approach over LZ78 compressed text. Black boxes are the explicit characters at the end of each block, while the lines are the implicit text that is represented by a reference.

Trying to apply a pure BM in this case may be costly, because we need to access the characters "inside" the blocks (the implicit ones). A character at distance i to the last character of a block needs going i blocks backward in the referencing chain, as each new LZ78 block consists of a previous one concatenated with a new letter.

Therefore we prefer to start by considering the explicit characters in the window. To maximize the shifts, we go from the rightmost to the leftmost. We precompute a table

$$B(i, c) = \min(\{i\} \cup \{i - j,\ 1 \leq j \leq i\ \wedge\ P_j = c\})$$

which gives the maximum safe shift given that at window position i the text character is c (this is similar to the SBM table, and can be easily computed in $O(m^2 + m\sigma)$ time). Note that the shift is zero if the pattern matches that window position.

As soon as one of the explicit characters permits a non-zero shift, we shift the window. Otherwise, we have to consider the implicit characters. Figure 3 shows the order in which we consider them. The last block is left for the end, since despite it can give good shifts, it is costly to reach the relevant characters (the block can be unfolded only from right to left). The other blocks are unfolded

in right to left order, block by block. When unfolding a block, we obtain a new text character (right to left) for each step backward in the referencing chain. For each such character, if we obtain a non-zero shift we immediately advance the window and restart the whole process with a new window.

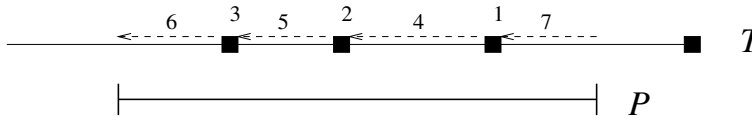


Fig. 3. Evaluation order for the simple algorithm. First the explicit characters right to left, then the implicit ones right to left (save the last one), and finally the last block.

If, after having considered all the characters we have not obtained a non-zero shift, we can report an occurrence of the pattern at the current window position. The window can then be advanced by one.

The algorithm can be applied on-line, that is, reading the compressed file block by block from disk. We read zero or more blocks until the last block read finishes ahead the window, then apply the previous procedure until we can shift the window, and start again. For each block read we store its last character, the block it references, its position in the uncompressed text and its length (these last two are not stored in the compressed file but computed on the fly).

Note that it is possible that the pattern is totally contained in a block, in which case the above algorithm will unfold the block to compare its internal characters against the pattern. It is clear that the method is efficient only if the pattern is not too short compared to the block length.

A slight improvement over this scheme is to add a kind of "skip-loop": instead of delaying the shifting until we read enough blocks, try to shift with the explicit character of each new block read. This is in practice like considering the explicit characters in left to right order. It needs more and shorter shifts but resorts less to previously stored characters. We call "BM-simple" our original version and "BM-simple-opt" this improvement.

Note that even in the best case we have a complexity of $\Omega(n)$ because all the text blocks have to be scanned. However, the method is faster in practice than previous algorithms, as shown later. Appendix A depicts the complete algorithm.

5 Multicharacter Boyer–Moore

Although the simple method is fast enough for reasonably large alphabets, it fails to produce good shifts when the alphabet is small (e.g. DNA). Multicharacter techniques, consisting in shifting by q-tuples of characters instead of one character, have been successfully applied to search uncompressed DNA [18]. Those

techniques effectively increase the alphabet size and produce longer shifts in exchange for slightly more costly comparisons.

We have attempted such an approach for our problem. We select a number q and build the shift tables considering q-grams. For instance, for the pattern `"abcdefg"`, the 3-gram `"cde"` considered at the last position yields a shift of 2, while `"xxx"` yields a shift of 5. Once the pattern is preprocessed we can shift using text q-grams instead of text characters. That is, if the text window is $x_1 x_2 \ldots x_m$ we try to shift using the q-grams $x_{m-q+1} \ldots x_m$, then $x_{m-q} \ldots x_{m-1}$, etc. until $x_1 \ldots x_q$. If none of these q-grams produces as positive shift, then the pattern matches the window. The preprocessing takes $O(m^2 + m\sigma^q)$ time.

The method is applied to the same LZ78 encoding as follows. At search time, we do not store anymore the last character of each block but its last q-gram. This last q-gram is computed on the fly, the format of the compressed file is the same as before. To compute it, we take the referenced block, strip the first character of its final q-gram and append the extra character of the new block. Then, the basic method is used except because we shift using the whole q-grams.

One complication appears when the block is shorter than q. In this case the best choice is to pad its q-gram with the last characters of the block that appears before it (if this is done all the time then the previous block does have a complete q-gram, except for the first blocks of the text). However, we must be careful when this short block is referenced, since only the characters that really belong to it must be taken from its last q-gram.

Finally, if q is not very small, the shift tables can be very large ($O(\sigma^q)$ size). We have used hashing from the q-grams to an integer range $0 \ldots N-1$ to reduce the size of the tables and to lower the preprocessing time to $O(m^2 + mN)$. This makes it necessary an explicit character-wise checking of possible matches, which is anyway required because we cannot efficiently check the first $q-1$ characters of the pattern.

We have implemented this technique using $q = 4$ (which is appropriate to store the q-gram in a word of our machine), and it is called "BM-multichar" in the experiments, where we show that it improves over BM-simple on DNA text.

6 Shifting by Complete Blocks

Despite that we have obtained good results with BM-multichar, we present now a more sophisticated technique that gave better results.

The idea is that, upon reading a new block, we could shift using the whole block. However, we cannot have an $B(i, b)$ table with one entry for each possible block b. Instead, we precompute

$$J(i,\ell) = \max(\ \{j,\ \ell \le j < i\ \wedge\ P_{j-\ell+1\ldots j} = P_{i-\ell+1\ldots i}\}$$
$$\cup\quad \{j,\ 0 \le j < \ell\ \wedge\ P_{1\ldots j} = P_{i-j+1\ldots i}\})$$

that tells, for a given pattern substring of length ℓ ending at i, the ending point of its closest previous occurrence in P (a partial occurrence trimmed at the

beginning of the pattern is also valid). The J table can be computed in $O(m^2)$ time by the simple trick of going from $\ell = 0$ to $\ell = m$ and using $J(*, \ell - 1)$ to compute $J(*, \ell)$, so that for all the cells of the form $J(i, *)$ there is only one backward traversal over the pattern.

Now, for each new block read $b_r = (s, c)$, we compute its last occurrence in P, $last(r)$. This is acomplished as follows. We start considering $last(s)$, i.e. the last position where the referenced block appears in P. We check if $P_{last(s)+1} = c$, in which case $last(r) = last(s) + 1$. If this is not the case, we need to obtain the previous occurrence of b_s in P, but this is also the previous occurrence of a pattern substring ending at $last(s)$. So we can use the J table to obtain all the following occurrences of b_s inside P, until we find one that is followed by c (and then this is the last occurrence of $b_r = b_s c$ in P) or we conclude that $last(r) = 0$. This process takes $O(\min(mn, \sigma^m))$ across all the search and is cheap in practice.

Once we have computed the last occurrence of each block inside P, we can use the information to shift the window. However, it is possible that the last occurrence of a block b_r inside P is indeed after the current position of b_r inside the window. In the simple approach (Section 4) this is solved by computing $B(i, c)$, i.e. the last occurrence of c inside P *before position* i. This would require too much effort in our case. We prefer to use J again in order to find previous occurrences of b_r inside P until we find one that is at the same position of b_r in the window or before. If it is at the same position we cannot shift, otherwise we displace the window. Figure 4 illustrates.

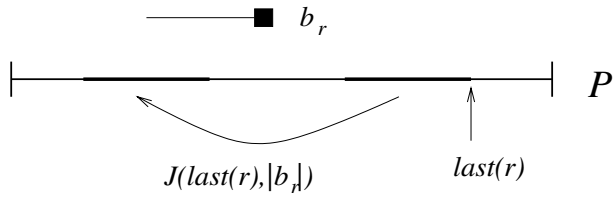


Fig. 4. Using the whole LZ78 block to shift the window. If its last occurrence in P is ahead, we use J until finding the adequate occurrence.

The blocks covered by the window are checked one by one, from right to left (excluding the last one whose endpoint is not inside the window). As soon as one allows a shift the window is advanced and the process restarted. If no shift is possible, the last block is unfolded until we obtain the contained block that corresponds exactly to the end of the window and make a last attempt with it. If all the shifting attempts fail, the window position is reported as a match and shifted in one.

As before, we read blocks until surpass the window and then try to shift. This method is called "BM-blocks" in this paper. The alternative method of trying to shift with each new block read is called "BM-blocks-opt". The algorithm is depicted in Appendix B.

In the same spirit of shifting using a variable number of characters, we have also adapted the match heuristic of BM. In this case, however, we cannot guarantee that the pattern will be matched right-to-left as on uncompressed text. Therefore, we compute a table $C(i, j)$ that gives the maximum shift if we have matched $P_{i...j}$ and P_{i-1} has mismatched. The definition of C is very similar to that of J. The C table is used when we compare the internal characters, since in that case a contiguous portion of P has been compared (the situation when comparing the explicit characters is much more complex since an arbitrary subset of the pattern positions have been compared). This method, called "BM-complete" in the experiments, did not yield good results.

7 Experimental Results

We tested our algorithms against the fastest existing implementation of previous work [17][1], using the same LZ78 compression format. The format uses a version that loses some compression in exchange for better decompression/search time. It stores the pair (s, c) as follows: s is stored as a sequence of bytes where the last bit is used to signal the end of the code; and c is coded as a whole byte.

The experiments were run on an Intel Pentium III machine running Linux. We have averaged user times over two different files of 10 Mb each. Patterns of lengths 10 to 100 were randomly selected from the texts (1,000 patterns of each length) and the same patterns were used for all the algorithms. The first text, WSJ, is a set of articles from The Wall Street Journal 1987 (natural language), while the second one is DNA with lines cut every 60 characters. We show user times in all the experiments.

Table 1 shows results related to compression efficiency for our compression format and two widely used compressors. As can be seen, our compression ratios are worse than those of classical compressors. On the other hand, decompression time is faster for our format, which improves search time.

Method	Compression ratio	Compression time	Decompression time
Ours	WSJ: 45.02%	WSJ: 5.09 sec	WSJ: 0.79 sec
(LZ78)	DNA: 39.69%	DNA: 4.31 sec	DNA: 0.72 sec
Unix *Compress*	WSJ: 38.75%	WSJ: 2.52 sec	WSJ: 0.92 sec
(LZW)	DNA: 27.91%	DNA: 2.43 sec	DNA: 0.75 sec
Gnu *gzip*	WSJ: 33.57%	WSJ: 10.63 sec	WSJ: 0.81 sec
(LZ77)	DNA: 30.43%	DNA: 25.10 sec	DNA: 0.78 sec

Table 1. Results on compression and decompression using different formats.

Figure 5 shows a comparison of the diverse search methods we have proposed along the paper. As can be seen, BM-simple-opt is the best choice for natural

[1] The bit-parallel algorithm of [14] should be similar, but it is implemented over Unix *Compress* and it is slower.

language, while BM-blocks (without the "optimization") is the best on DNA. BM-multichar works better than BM-simple on DNA, but BM-blocks is superior.

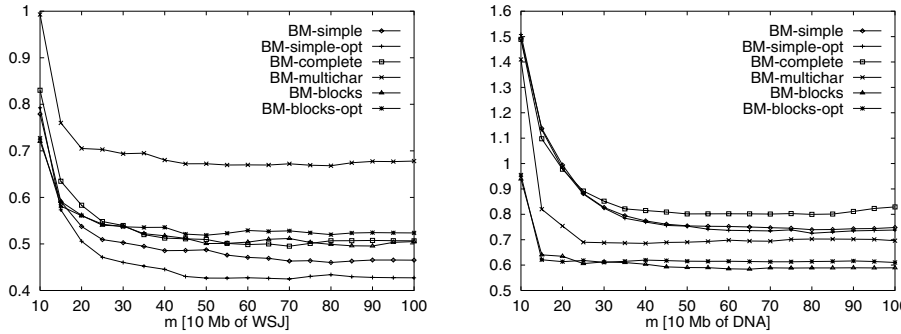


Fig. 5. Search time (in seconds of user time) for our different algorithms.

Figure 6 compares our best algorithms against previous work. The previous algorithm [17] is called "Bit-parallel" and our implementation of it works only until $m = 32$ (it would be slower, not faster, for longer patterns). We have also considered the "naive" approach of decompressing-then-searching. Two choices are shown: DS uses our LZ78 format and decompresses the file in memory while applying a Sunday [19] search algorithm over it; "D+Agrep" first decompresses the text and then then runs *agrep* over it. *Agrep* [21,22] is considered the fastest text searching tool, and we recall that the decompression time of our format is the fastest.

As can be seen, our algorithms are significantly faster than Bit-parallel (up to 30%) and than both decompress-then-search approaches (up to 50%), even for short patterns ($m \geq 15$).

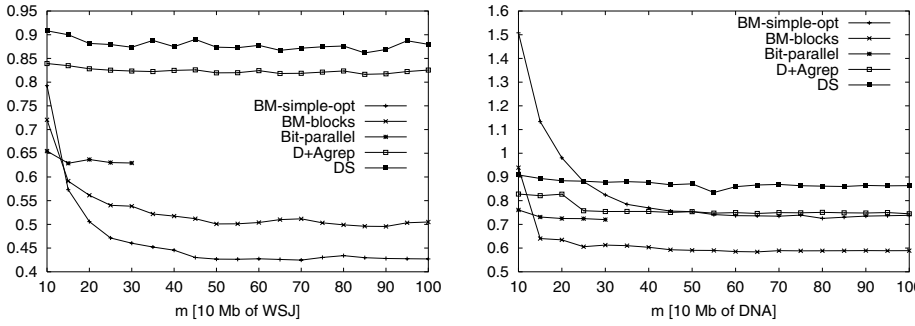


Fig. 6. Search time (in seconds of user time) of the best previous algorithms and our new ones.

It is also interesting that the methods reach soon a limit on m from where they do not improve anymore. This is due to the need for reading all the text blocks, regardless of how long is the pattern. This is unavoidable in principle to know the text position we are on.

If we compute the scanning efficiency of our best algorithms, we have that they are able to search at up to 16.6 Mb/sec on DNA and 22.2 Mb/sec on natural language text (computed on the uncompressed text). Bit-parallel obtains 14 Mb/sec on DNA and 16 Mb/sec on WSJ, while decompress-then-search achieves 13 Mb/sec on DNA and 12 Mb/sec on WSJ. If we have the text already decompressed, then *agrep* alone scans the text at about 300 Mb/sec times faster. This shows that, despite that we offer an interesting alternative to decompressing and searching of texts that have to be compressed by some other reason, we are far from giving an extra reason to compress the text (i.e. achieving less time in searching the compressed text than for searching the uncompressed text).

On the other hand, we would like to point out that these results have a strong dependence with respect to the type of machine used. The same experiments run on a Sun UltraSparc-1 of 167 MHz gave, for $m = 30$ on WSJ, 1.4 seconds for BM-simple-opt as well as for the bit-parallel algorithm of [17], while *agrep* took about 0.45 seconds. We developed another version of BM-simple-opt based on a different coding that, in exchange for 2% to 4% extra space, permits to know the length of the new block without accessing the referenced one. This new algorithm takes about 0.85 seconds under the above conditions, which is 60% of the time of the bit-parallel algorithm and about twice the time of pure *agrep*. This version, however, is slower than the original one on the Intel machine. This shows that locality of reference is much more important on the Sun machine.

8 Conclusions

We have presented the first Boyer–Moore approaches to string matching over Ziv-Lempel compressed text (specifically, the LZ78 format). We first presented a simple approach close to the Simplified-Boyer–Moore algorithm that is the best for all but very small alphabets (e.g. it is suitable for natural language). Then we presented stronger approaches, the most successful one based on shifting on complete LZ78 blocks. This one is the fastest for small alphabets (e.g. DNA text). Our experimental results show that the new algorithms are faster than the best previous approaches even from patterns of length 15, achieving up to 30% reductions in the search time. The new algorithms hold the characteristic feature of all the search algorithms of Boyer–Moore type: the algorithms run faster when the pattern gets longer (up to a certain limit).

We could theoretically strenghten the algorithms in the following way: if the window contains a border between blocks then we apply our shifting machinery, but when it is inside a block we simply copy the matches already found in the text area that the containing block references, and shift the window to the end of the block. However, in practice the blocks are not so long for this to make a real difference.

We are currently working in improving the current techniques and exploring new ones, as there are many open options. A very interesting question is whether it is possible to avoid reading all the text blocks, as this is the major bottleneck for further improvement.

References

1. A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proc. DCC'92*, pages 279–288, 1992.
2. A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *J. of Comp. and Sys. Sciences*, 52(2):299–307, 1996.
3. A. Apostolico and Z. Galil. *Pattern Matching Algorithms*. Oxford University Press, Oxford, UK, 1997.
4. T. Bell, J. Cleary, and I. Witten. *Text Compression*. Prentice Hall, 1990.
5. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *CACM*, 20(10):762–772, 1977.
6. M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.
7. M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20:388–404, 1998.
8. L. Gasieniec, M. Karpinksi, W. Plandowski, and W. Rytter. Efficient algorithms for Lempel-Ziv encodings. In *Proc. SWAT'96*, 1996.
9. R. N. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10:501–506, 1980.
10. D. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the I.R.E.*, 40(9):1090–1101, 1952.
11. J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching over ziv-lempel compressed text. In *Proc. CPM'2000*, LNCS1848, 2000, pp. 195–209.
12. T. Kida, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. A unifying framework for compressed pattern matching. In *Proc. 6th Intl. Symp. on String Processing and Information Retrieval (SPIRE'99)*, pages 89–96. IEEE CS Press, 1999.
13. T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In *Proc. DCC'98*, 1998.
14. T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Shift-And approach to pattern matching in LZW compressed text. In *Proc. CPM'99*, LNCS 1645, pages 1–13, 1999.
15. U. Manber. A text compression scheme that allows fast searching directly in the compressed file. *ACM Trans. on Information Systems*, 15(2):124–136, 1997.
16. E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Trans. on Information Systems*, 2000. To appear. Previous versions in *SIGIR'98* and *SPIRE'98*.
17. G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In *Proc. CPM'99*, LNCS 1645, pages 14–36, 1999.
18. H. Peltola and J. Tarhio. String matching in the DNA alphabet. *Software Practice and Experience*, 27(7):851–861, 1997.
19. D. Sunday. A very fast substring search algorithm. *CACM*, 33(8):132–142, 1990.
20. T. A. Welch. A technique for high performance data compression. *IEEE Computer Magazine*, 17(6):8–19, June 1984.

21. S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, October 1992.
22. S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. USENIX Technical Conference*, pages 153–162, Berkeley, CA, USA, Winter 1992.
23. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23:337–343, 1977.
24. J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Trans. Inf. Theory*, 24:530–536, 1978.

A The Simple Optimized Algorithm in Detail

```
Search (P,m,Z,n)

        /* Preprocessing */
   for (i ∈ 1...m, c ∈ Σ) B(i,c) ← i
   for (i ∈ 1...m)
      for (j ∈ i...m) B(j,P_i) ← j − i
        /* Searching */
   tpos ← 0     /* window initial position */
   rpos ← 0     /* amount of text read */
   length(0) ← 0     /* length of the blocks */
   from(0) ← 0     /* text position of the blocks */
   i ← 0     /* number of current block */
   while (true)
      readBlocks:
      if (rpos − tpos ≤ m)
         while (true)
            i ← i + 1
            if (i > n) finish the search
            obtain b_i = (j,c) from Z
            ref(i) ← j     /* referenced block */
            lastchar(i) ← c     /* char at the end */
            from(i) ← rpos
            length(i) ← length(j) + 1
            rpos ← rpos + length(i)
            if (rpos − tpos > m) break loop
            tpos ← tpos + B(rpos − tpos, c)
         /* try to shift with explicit characters */
      j ← i
      while (true)
         offset ← from(j) − tpos
         if (offset < 1) break loop
         shift ← B(offset, char(j − 1))
         if (shift > 0)
            tpos ← tpos + shift
            goto label readBlocks
         j ← j − 1
         /* unable to shift by explicit characters, unfold */
      j ← i − 1
```

```
    while (true)
      length ← length(j) − 1
      k ← j
      offset ← from(j) − tpos
      if (offset + length ≤ 0) goto label expandLast
      while (length > 0)
        k ← ref(k)
        shift ← B(offset + length, char(k))
        if (shift > 0)
          tpos ← tpos + shift
          goto label readBlocks
        length ← length − 1
        if (offset + length ≤ 0) goto label expandLast
      j ← j − 1
      /* only the last (i-th) block rests to be tested */
    expandLast:
    length ← length(i) − 1
    k ← i
    offset ← from(i) − tpos
    while (offset + length > m)
      k ← ref(k)
      length ← length − 1
    while (length > 0 ∧ offset + length > 0)
      k ← ref(k)
      shift ← B(offset + length, char(k))
      if (shift > 0)
        tpos ← tpos + shift
        goto label readBlocks
      length ← length − 1
      /* it passed all the tests, report the match */
    report a match beginning at tpos
    tpos ← tpos + 1
```

B The Algorithm that Shifts by Blocks in Detail

```
Search (P,m,Z,n)

        /* Preprocessing (O(m²), not O(m³)) */
  for (ℓ ∈ 0...m) J(0,ℓ) ← 0
  for (i ∈ 1...m) J(i,0) ← i
  for (ℓ ∈ 1...m)
    for (i ∈ 2...m)
      j ← J(i − 1, ℓ − 1)
      while (j > 0 ∧ P_{j+1} ≠ P_i) j ← J(j, ℓ − 1)
      if (j = 0 ∧ P_{j+1} ≠ P_i) j ← j − 1
      J(i,ℓ) ← j + 1
      /* Searching */
  tpos ← 0    /* window initial position */
  rpos ← 0    /* amount of text read */
```

```
length(0) ← 0    /* length of the blocks */
from(0) ← 0    /* text position of the blocks */
last(0) ← m
i ← 0    /* number of current block */
while (true)
   readBlocks:
   while (rpos − tpos ≤ m)
      i ← i + 1
      if (i > n) finish the search
      obtain b_i = (j, c) from Z
      ref(i) ← j    /* referenced block */
      lastchar(i) ← c    /* char at the end */
      from(i) ← rpos
      length(i) ← length(j) + 1
      pos ← last(j)
      if (length(j) > m) ℓ ← m else ℓ ← length(j)
      while (pos > 0 ∧ (pos = m ∨ P_{pos+1} ≠ c))
         pos ← J(pos, ℓ)
      if (pos = 0 ∧ (P_1 ≠ c)) pos ← pos − 1
      last(i) ← pos + 1
      rpos ← rpos + length(i)
      /* try to shift with complete blocks (exclude last one) */
   j ← i − 1
   while (true)
      offset ← from(j + 1) − tpos
      if (offset < 1) break loop
      pos ← last(j)
      if (pos > offset)
         ℓ ← length(j)
         if (ℓ > m) ℓ ← m
         pos ← J(pos, ℓ)
         while (pos > offset) pos ← J(pos, ℓ)
      if (pos < offset)
         tpos ← tpos + offset − pos
         goto label readBlocks
      j ← j − 1
      /* only the last (i-th) block rests to be tested */
   j ← i
   offset ← from(j) − tpos − m
   length ← length(i)
   while (offset + length > 0)
      j ← ref(j)
      length ← length − 1
   pos ← last(j)
   if (pos < m)
      tpos ← tpos + m − pos
      goto label readBlocks
      /* it passed all the tests, report the match */
   report a match beginning at tpos
   tpos ← tpos + 1
```

A Boyer–Moore Type Algorithm for Compressed Pattern Matching

Yusuke Shibata, Tetsuya Matsumoto, Masayuki Takeda,
Ayumi Shinohara, and Setsuo Arikawa

Department of Informatics, Kyushu University 33
Fukuoka 812-8581, Japan
{yusuke,tetsuya,takeda,ayumi,arikawa}@i.kyushu-u.ac.jp

Abstract. We apply the Boyer–Moore technique to compressed pattern matching for text string described in terms of collage system, which is a formal framework that captures various dictionary-based compression methods. For a subclass of collage systems that contain no truncation, our new algorithm runs in $O(\|\mathcal{D}\| + n \cdot m + m^2 + r)$ time using $O(\|\mathcal{D}\| + m^2)$ space, where $\|\mathcal{D}\|$ is the size of dictionary $\mathcal{D}$, n is the compressed text length, m is the pattern length, and r is the number of pattern occurrences. For a general collage system, the time complexity is $O(\mathit{height}(\mathcal{D}) \cdot (\|\mathcal{D}\|+n)+n \cdot m+m^2+r)$, where $\mathit{height}(\mathcal{D})$ is the maximum dependency of tokens in $\mathcal{D}$. We showed that the algorithm specialized for the so-called byte pair encoding (BPE) is very fast in practice. In fact it runs about $1.2 \sim 3.0$ times faster than the exact match routine of the software package `agrep`, known as the fastest pattern matching tool.

1 Introduction

The problem of compressed pattern matching is to find pattern occurrences in compressed text without decompression. The goal is to search in compressed files faster than a regular decompression followed by an ordinary search (**Goal 1**). This problem has been extensively studied for various compression methods by several researchers in the last decade. For recent developments, see the survey [18].

This paper, however, focuses on another aspect of compressed pattern matching. We intend to reduce the time taken to search through a text file by reducing the size of it in a special way. That is, we regard text compression as a means of speeding up pattern matching rather than of saving storage or communication costs. The research goal to this direction is to search in compressed files faster than an ordinary search in the original files (**Goal 2**). If the goal is achieved, files that are usually not compressed because they are often read, can now be compressed for a speed-up. Let t_d, t_s, and t_c be the time for a decompression, the time for searching in uncompressed files, and the time for searching in compressed files, respectively. Goal 1 aims for $t_d + t_s > t_c$ while Goal 2 for $t_s > t_c$. Thus, Goal 2 is more difficult to achieve than Goal 1.

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 181–194, 2000.

Let n and N denote the compressed text length and the original text length, respectively. Theoretically, the best compression has $n = \sqrt{N}$ for the Lempel-Ziv-Welch (LZW) encoding [22], and $n = \log N$ for LZ77 [25]. Thus an $O(n)$ time algorithm for searching directly in compressed text is considered to be better than an $O(N)$ time algorithm for searching in the original text. However, in practice n is linearly proportional to N for real text files. For this reason, an elaborate $O(n)$ time algorithm for searching in compressed text is often slower than a simple $O(N)$ time algorithm running on the original text, namely, does not achieve Goal 2. For example, it is reported in [12,11,16] that the proposed algorithms of compressed pattern matching for LZW achieved Goal 1, but did not achieve Goal 2. In order to achieve Goal 2, we shall re-estimate existing compression methods, choose a suitable one, and then develop an efficient compressed pattern matching algorithm for it. It should be emphasized that we are not particular about the traditional criteria, i.e., the compression ratio and the compression/decompression time.

As an effective tool for such a re-estimation, we introduced in [10] a *collage system*, that is a formal system to describe a string by a pair of dictionary $\mathcal{D}$ and sequence $\mathcal{S}$ of tokens defined in $\mathcal{D}$. The basic operations are concatenation, truncation, and repetition. Collage systems give us a unifying framework of various dictionary-based compression methods. We developed in [10] a general compressed pattern matching algorithm for the framework, which basically simulates the move of the Knuth-Morris-Pratt (KMP) automaton [13] on original texts. For a collage system which contains no truncation, the algorithm runs in $O(n + r)$ time after an $O(\|\mathcal{D}\| + m^2)$ time and space preprocessing, where $\|\mathcal{D}\|$ denotes the size of dictionary $\mathcal{D}$, m is the pattern length, and r is the number of pattern occurrences. For the case of LZW, it matches the same bound given in [12]. For a general collage system, which contains truncation, it runs in $O(\mathit{height}(\mathcal{D}) \cdot n + r)$ time after an $O(\mathit{height}(\mathcal{D}) \cdot \|\mathcal{D}\| + m^2)$ time preprocessing using $O(\|\mathcal{D}\| + m^2)$ space, where $\mathit{height}(\mathcal{D})$ denotes the maximum dependency of the operations in $\mathcal{D}$. These results show that the truncation slows down the compressed pattern matching to the factor $\mathit{height}(\mathcal{D})$. It coincides with the observation by Navarro and Raffinot [16] that LZ77 is not suitable for compressed pattern matching compared with LZ78/LZW compression.

In a recent work [19], we focused on the compression method called the byte pair encoding (BPE) [9]. It describes a text as a collage system with concatenation only, in which the size of $\mathcal{D}$ is restricted to at most 256 so as to encode each token of $\mathcal{S}$ into one byte. Decompression is fast and requires small work space. Moreover, partial decompression is possible. This is a big advantage of BPE compared with the Lempel-Ziv family. Despite such advantages, BPE was seldom used until now. The reason is mainly for the following two disadvantages: the compression is terribly slow and the compression ratio is not as good as those of Compress and Gzip. However, we have shown that BPE is suitable for speeding up pattern matching. The algorithm proposed in [19] runs in $O(n+r)$ time after an $O(\|\mathcal{D}\| \cdot m)$ time and space preprocessing, and it is indeed faster than such $O(N)$ time algorithms as the KMP algorithm and the Shift-Or algorithm [24,4].

Moreover, it can be extended to deal with multiple patterns. The searching time is reduced at almost the same rate as the compression ratio.

However, there are sublinear time algorithms for the usual (not compressed) pattern matching problem, such as the Boyer–Moore (BM) algorithm [5], which skip many characters of text and run faster than the $O(N)$ time algorithms on the average, although the worst-case running time is $O(mN)$. Our algorithm presented in [19] defeats the exact match routine of `agrep` [23] for highly compressible texts such as genomic data. But it is not better than `agrep` when searching for a long pattern in texts that are not highly compressible by BPE. Then a question arises: *Does text compression speed up such a sublinear time algorithm?*

In this paper, we give an affirmative answer to this question. We present a general, BM type algorithm for texts described in terms of collage system. The algorithm runs on the sequence $\mathcal{S}$, with skipping some tokens. To our best knowledge, this is the first attempt to develop such an algorithm in compressed text.[1] The token-wise processing has two advantages compared with the usual character-wise processing. One is quick detection of a mismatch at each stage of the algorithm, and the other is larger shift depending upon one token (not upon one character) to align the phrase with its occurrence within the pattern. For a general collage system, the algorithm runs in $O((\mathit{height}(\mathcal{D}) + m) \cdot n + r)$ time, after an $O(\mathit{height}(\mathcal{D}) \cdot \|\mathcal{D}\| + m^2)$ time preprocessing with $O(\|\mathcal{D}\| + m^2)$ space. In the case of no truncation, it runs in $O(mn + r)$ time, after an $O(\|\mathcal{D}\| + m^2)$ time and space preprocessing. However, we cannot shift the pattern without knowing the total length of phrases corresponding to skipped tokens. This slows down the algorithm in practice. To do without such information, we assume that the skipped phrases are all of length C, the maximum phrase length in $\mathcal{D}$, and divide the shift value by C. The value of C is crucial in this approach. For the BPE compression, we observed that putting a restriction on C makes no great sacrifice of compression ratio even for $C = 3, 4$. Experimental results show that the proposed algorithm searching in BPE compressed files is about $1.2 \sim 3.0$ times faster than `agrep` on the original files.

There are a few researches that aims Goal 2. The first attempt was made by Manber [14]. The compression scheme used is similar to but simpler than BPE, in which the maximum phrase length C is restricted to 2. The approach is to encode a given pattern and to apply any search routine in order to find the encoded pattern within compressed files. The problem in this approach is that the pattern may have more than one encoding. The solution given in [14] is to devise a way to restrict the number of possible encodings for any string with sacrifices in compression ratio. Thus the reductions in file size and searching time are only about 30%. Miyazaki et al. [15] presented an efficient realization of pattern matching machine for searching directly in a Huffman encoded text. The reduction in searching time is almost the same as that in file size. Moura et al. [8] proposed a compression scheme that uses a word-based Huffman encoding with

[1] However, Navarro et al. [17] in this conference gives a similar algorithm, which is restricted to the LZ78/LZW format.

a byte-oriented code, which allows a search twice faster than `agrep`. However, the compression method is not applicable to such texts as genomic sequence data since they cannot be segmented into words. Our previous and new algorithms can deal with such texts.

2 Preliminaries

Let Σ be a finite alphabet. An element of Σ^* is called a *string*. Strings x, y, and z are said to be a *prefix*, *factor*, and *suffix* of the string $u = xyz$, respectively. A prefix, factor, and suffix of a string u is said to be *proper* if it is not u. The length of a string u is denoted by $|u|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. The ith symbol of a string u is denoted by $u[i]$ for $1 \leq i \leq |u|$, and the factor of a string u that begins at position i and ends at position j is denoted by $u[i:j]$ for $1 \leq i \leq j \leq |u|$. For convenience, let $u[i:j] = \varepsilon$ for $j < i$. For a string u and a non-negative integer i, the string obtained by removing the length i prefix (resp. suffix) from u is denoted by ${}^{[i]}u$ (resp. $u^{[i]}$). That is, ${}^{[i]}u = u[i+1:|u|]$ and $u^{[i]} = u[1:|u|-i]$.

3 A Unifying Framework for Compressed Pattern Matching

In a dictionary-based compression, a text string is described by a pair of a *dictionary* and a sequence of *tokens*, each of which represents a phrase defined in the dictionary. Kida et al. [10] introduced a unifying framework, named *collage system*, which abstracts various dictionary-based methods, such as the Lempel-Ziv family and the static dictionary methods. In [10] they presented a general compressed pattern matching algorithm for the framework. Consequently, any compression method within the framework has a compressed pattern matching algorithm as an instance.

3.1 Collage System

A *collage system* is a pair $\langle \mathcal{D}, \mathcal{S} \rangle$ defined as follows: $\mathcal{D}$ is a sequence of assignments $X_1 = expr_1;\ X_2 = expr_2;\ \cdots;\ X_\ell = expr_\ell$, where each X_k is a token (or a variable) and $expr_k$ is any of the form

a	for $a \in \Sigma \cup \{\varepsilon\}$,	(*primitive assignment*)
$X_i X_j$	for $i, j < k$,	(*concatenation*)
${}^{[j]}X_i$	for $i < k$ and an integer j,	(*prefix truncation*)
$X_i^{[j]}$	for $i < k$ and an integer j,	(*suffix truncation*)
$(X_i)^j$	for $i < k$ and an integer j.	(j *times repetition*)

Each token represents a string obtained by evaluating the expression as it implies. The strings represented by tokens are called *phrases*. Denote by $X.u$ the phrase represented by a token X. The *size* of $\mathcal{D}$ is the number n of assignments and

denoted by $\|\mathcal{D}\|$. Define the *height* of a token X to be the height of the syntax tree whose root is X. The *height* of $\mathcal{D}$ is defined by $height(\mathcal{D}) = \max\{height(X) \mid X \text{ in } \mathcal{D}\}$. It expresses the maximum dependency of the tokens in $\mathcal{D}$. On the other hand, $\mathcal{S} = X_{i_1}, X_{i_2}, \ldots, X_{i_n}$ is a sequence of tokens defined in $\mathcal{D}$. The collage system represents a string obtained by concatenating the phrases represented by $X_{i_1}, X_{i_2}, \ldots, X_{i_n}$.

3.2 Pattern Matching in Collage Systems

Our problem is defined as follows.

> Given a pattern $\pi = \pi[1 : m]$ and a collage system $\langle \mathcal{D}, \mathcal{S} \rangle$ with $\mathcal{S} = \mathcal{S}[1 : n]$, find all locations at which π occurs within the original text $\mathcal{S}[1].u \cdot \mathcal{S}[2].u \cdots \mathcal{S}[n].u$.

Kida et al. [10] presented an algorithm solving the above problem. Figure 1 gives an overview of the algorithm, which processes $\mathcal{S}$ token-by-token. The algorithm

```
Input:   Pattern π and collage system consisting of D and S = S[1 : n].
Output:  All occurrences of π in the original text.
begin
/* Preprocessing for computing Jump_KMP and Output_KMP. */
    Preprocess the pattern π and the dictionary D;
/* Main routine */
    state := 0;   ℓ := 0;
    for i := 1 to n do begin
        for each d ∈ Output_KMP(state, S[i]) do
            Report a pattern occurrence that ends at position ℓ + d;
        state := Jump_KMP(state, S[i]);   ℓ := ℓ + |S[i].u|
    end
end.
```

Fig. 1. General algorithm for searching in a collage system.

simulates the move of the KMP automaton running on the original text, by using two functions $Jump_{\mathrm{KMP}}$ and $Output_{\mathrm{KMP}}$, both take as input a state and a token. The former is used to substitute just one state transition for the consecutive state transitions of the KMP automaton caused by each of the phrases, and the latter is used to report all pattern occurrences found during the state transitions. Thus the definitions of the two functions are as follows.

$$Jump_{\mathrm{KMP}}(j, t) = \delta(j, t.u),$$

$$Output_{\mathrm{KMP}}(j, t) = \left\{ |v| \;\middle|\; \begin{array}{l} v \text{ is a non-empty prefix of } t.u \\ \text{such that } \delta(j, v) \text{ is the final state} \end{array} \right\},$$

where δ is the state transition function of the KMP automaton.

Theorem 1 (Kida et al. [10]). *The algorithm of Fig. 1 runs in $O(height(\mathcal{D}) \cdot n+r)$ time after an $O(height(\mathcal{D}) \cdot \|\mathcal{D}\|+m^2)$ time preprocessing using $O(\|\mathcal{D}\|+m^2)$ space, where r is the number of pattern occurrences. The factor $height(\mathcal{D})$ can be dropped if $\mathcal{D}$ contains no truncation.*

This idea is a generalization of the algorithm due to Amir et al. [3], which is restricted to LZW compressed texts. Shibata et al. [20] applied a similar technique to the case of the compression using anti-dictionaries [6]. An extension of [3] to multiple pattern searching was presented by Kida et al. [12], which is based on the Aho-Corasick (AC) pattern matching algorithm [1]. The technique of [12] was then generalized to multiple pattern searching in collage systems which contain concatenation only [10]. Bit-parallel realization of [3] was independently proposed in [11,16] and proved to be fast in practice for a short pattern ($m \leq 32$).

3.3 Practical Aspects

Theorem 1 suggests that a compression method which describes a text as a collage system with no truncation might be suitable for the compressed pattern matching. For example, the collage systems for LZW contain no truncation but those for LZ77 have truncation. It implies that LZW is suitable compared with LZ77. This coincides with the observation by Navarro and Raffinot [16] that the compressed pattern matching for LZ77 achieves none of Goal 1 and Goal 2, but that for LZW achieves Goal 1. However, it was observed that the compressed pattern matching for LZW is too slow to achieve Goal 2. We have two reasons. One is that in LZW the dictionary $\mathcal{D}$ is not encoded explicitly: it will be incrementally re-built from $\mathcal{S}$. The preprocessing of $\mathcal{D}$ is therefore merged into the main routine (see Fig. 1 again). The other reason is as follows. Although $Jump_{\mathrm{KMP}}$ can be realized using only $O(\|\mathcal{D}\| + m^2)$ space so that it answers in constant time, the constant factor is relatively large. The two-dimensional array realization of $Jump_{\mathrm{KMP}}$ would improve this, but it requires $O(\|\mathcal{D}\| \cdot m)$ space, which is unrealistic because $\|\mathcal{D}\|$ is linear in n in the case of LZW.

From the above observations the desirable properties for compressed pattern matching can be summarized as follows.

- The dictionary $\mathcal{D}$ contains no truncation.
- The dictionary $\mathcal{D}$ is encoded separately from the sequence $\mathcal{S}$.
- The size of $\mathcal{D}$ is small enough.
- The tokens of $\mathcal{S}$ are encoded using a fixed length code.

The BPE compression [9] is the one which satisfies all of the properties. The collage systems for BPE have concatenation only, and $\|\mathcal{D}\|$ is restricted to at most 256 so as to encode each token of $\mathcal{S}$ into one byte. By using the two-dimensional array implementation, we presented in [19] an algorithm for searching in BPE compressed files, which runs in $O(n+r)$ time after an $O(\|D\| \cdot m)$ time and space preprocessing. This algorithm defeats the BM algorithm for highly compressible text files such as biological data. However, it is not better

```
T[0] := $; /* $ is a character that never occurs in pattern */
i := m;
while i ≤ N do begin
    state := 0;    ℓ := 0;
    while g(state, T[i − ℓ]) is defined do begin
        state := g(state, T[i − ℓ]);        ℓ := ℓ + 1
    end;
    if state = m then report a pattern occurrence;
    i := i + σ(state, T[i − ℓ])
end
```

Fig. 2. BM algorithm on uncompressed text.

than the BM algorithm in the case of searching for a long pattern in text files that are not highly compressible by BPE. For this reason, we try to devise a BM type algorithm for searching in BPE compressed files.

4 BM Type Algorithm for Compressed Pattern Matching

We first briefly sketches the BM algorithm, and show a general, BM type algorithm for searching in collage systems. Then, we discuss searching in BPE compressed files from the practical viewpoints.

4.1 BM Algorithm on Uncompressed Text

The BM algorithm performs the character comparisons in the right-to-left direction, and slides the pattern to the right using the so-called shift function when a mismatch occurs. The algorithm for searching in text $T[1 : N]$ is shown in Fig. 2. Note that the function g is the state transition function of the (partial) automaton that accepts the reversed pattern, in which state j represents the length j suffix of the pattern ($0 \leq j \leq m$).

Although there are many variations of the shift function, they are basically designed to shift the pattern to the right so as to align a text substring with its rightmost occurrence within the pattern. Let

$$rightmost\_occ(w) = \min \left\{ \ell > 0 \;\middle|\; \begin{array}{l} \pi[m-\ell-|w|+1 : m-\ell] = w, \text{ or} \\ \pi[1 : m-\ell] \text{ is a suffix of } w \end{array} \right\}.$$

The following definition, given by Uratani and Takeda [21] (for multiple pattern case), is the one which utilizes all information gathered in one execution of the inner-while-loop in the algorithm of Fig. 2.

$$\sigma(j, a) = rightmost\_occ(a \cdot \pi[m-j+1 : m]).$$

The two-dimensional array realization of this function requires $O(|\Sigma| \cdot m)$ memory, but it becomes realistic due to recent progress in computer technology. Moreover, the array can be shared with the goto function g. This saves not only memory requirement but also the number of table references.

4.2 BM Type Algorithm for Collage System

Now, we show a BM type algorithm for searching in collage systems. Figure 3 gives an overview of our algorithm. For each iteration of the while-loop, we report

```
/*Preprocessing for computing Jump_BM(j,t), Output_BM(j,t), and Occ(t) */
    Preprocess the pattern π and the dictionary D;
/* Main routine */
    focus := an appropriate value;
    while focus ≤ n do begin
Step 1: Report all pattern occurrences that are contained in the phrase S[focus].u
          by using Occ(t);
Step 2: Find all pattern occurrences that end within the phrase S[focus].u
          by using Jump_BM(j,t) and Output_BM(j,t);
Step 3: Compute a possible shift Δ based on information gathered in Step 2;
        focus := focus + Δ
    end
```

Fig. 3. Overview of BM type compressed pattern matching algorithm.

in Step 1 all the pattern occurrences that are contained in the phrase represented by the token we focus on, determine in Step 2 the pattern occurrences that end within the phrase, and then shift our focus to the right by Δ obtained in Step 3. Let us call the token we focus on the *focused token*, and the phrase it represents the *focused phrase*. For step 1, we shall compute during the preprocessing, for every token t, the set $Occ(t)$ of all pattern occurrences contained in the phrase $t.u$. The time and space complexities of this computation are as follows.

Lemma 1 (Kida et al. 1999). *We can build in $O(height(\mathcal{D}) \cdot \|\mathcal{D}\| + m^2)$ time using $O(\|\mathcal{D}\| + m^2)$ space a data structure by which the enumeration of the set $Occ(t)$ is performed in $O(height(t) + \ell)$ time, where $\ell = |Occ(t)|$. If $\mathcal{D}$ contains no truncation, it can be built in $O(\|\mathcal{D}\| + m^2)$ time and space, and the enumeration requires only $O(\ell)$ time.*

In the following we discuss how to realize Step 2 and Step 3.

Figure 4 illustrates pattern occurrences that end within the focused phrase. A candidate for pattern occurrence is a non-empty prefix of the focused phrase that is also a proper suffix of the pattern. There may be more than one candidate to be checked. One naive method is to check all of them independently, but we take here another approach. We shall start with the longest one. For the case of uncompressed text, we can do it by using the partial automaton for the reversed pattern stated in Section 4.1. When a mismatch occurs, we change the state by using the failure function and try to proceed into the left direction. The process is repeated until the pattern does not have an overlap with the focused phrase. In order to perform such processing over compressed text, we use the two functions $Jump_{\mathrm{BM}}$ and $Output_{\mathrm{BM}}$ defined in the sequel.

Let $lpps(w)$ denote the longest prefix of a string w that is also a proper suffix of the pattern π. Extend the function g into the domain $\{0,\ldots,m\} \times \Sigma^*$ by $g(j, aw) = g(g(j, w), a)$, if $g(j, w)$ is defined and otherwise, $g(j, aw)$ is undefined, where $w \in \Sigma^*$ and $a \in \Sigma$. Let $f(j)$ be the largest integer k $(k < j)$ such that the length k suffix of the pattern is a prefix of the length j suffix of the pattern. Note that f is the same as the failure function of the KMP automaton. Define the functions $Jump_{\mathrm{BM}}$ and $Output_{\mathrm{BM}}$ by

$$Jump_{\mathrm{BM}}(j,t) = \begin{cases} g(j, t.u), & \text{if } j \neq 0; \\ g(j, lpps(t.u)), & \text{if } j = 0 \text{ and } lpps(t.u) \neq \varepsilon; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

$$Output_{\mathrm{BM}}(j,t) = \begin{cases} true, & \text{if } g(j,w) = m \text{ and } w \text{ is a proper suffix of } t.u; \\ false, & \text{otherwise.} \end{cases}$$

The procedure for Step 2 is shown in Fig. 5.

We now discuss how to compute the possible shift Δ of the focus. Let

$$Shift(j,t) = rightmost\_occ(t.u \cdot \pi[m-j+1:m]).$$

Assume that starting at the token $\mathcal{S}[focus]$, we encounter a mismatch against a token t in state j. Find the minimum integer $k > 0$ such that

$$Shift(0, \mathcal{S}[focus]) \leq \sum_{i=1}^{k} \Big| \mathcal{S}[focus+i].u \Big|, \quad \text{or} \tag{1}$$

$$Shift(j,t) \leq \sum_{i=0}^{k} \Big| \mathcal{S}[focus+i].u \Big| - \Big| lpps(\mathcal{S}[focus].u) \Big|. \tag{2}$$

Note that the shift due to Eq. (1) is possible independently of the result of the procedure of Fig. 5. When returning at the first if-then statement of the procedure in Fig. 5, we can shift the focus by the amount due to Eq. (1). Otherwise, we shift the focus by the amount due to both Eq. (1) and Eq. (2) for $j = state$ and $t = \mathcal{S}[focus-\ell]$ just after the execution of the while-loop at the first iteration of the repeat-until loop.

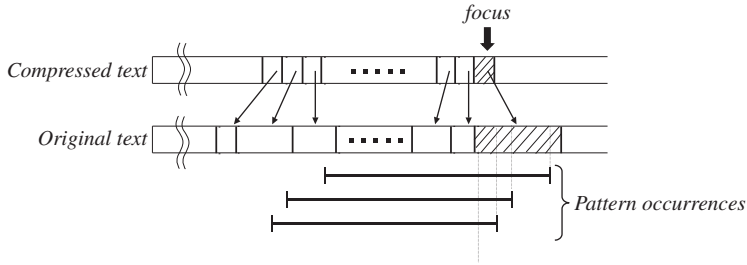


Fig. 4. Pattern occurrences.

.

```
procedure Find_pattern_occurrences(focus : integer);
begin
    if Jump_BM(0, S[focus]) is undefined then return;
    state := Jump_BM(0, S[focus]);    d := state;    ℓ := 1;
    repeat
        while Jump_BM(state, S[focus − ℓ]) is defined do begin
            state := Jump_BM(state, S[focus − ℓ]);    ℓ := ℓ + 1
        end;
        if Output_BM(state, S[focus − ℓ]) = true then report a pattern occurrence;
        d := d − (state − f(state));    state := f(state)
    until d ≤ 0
end;
```

Fig. 5. Finding pattern occurrences in Step 2.

Lemma 2. *The functions $Jump_{\mathrm{BM}}$, $Output_{\mathrm{BM}}$, and Shift can be built in $O(height(\mathcal{D}) \cdot \|\mathcal{D}\| + m^2)$ time and $O(\|\mathcal{D}\| + m^2)$ space, so that they answer in $O(1)$ time. The factor $height(\mathcal{D})$ can be dropped if $\mathcal{D}$ contains no truncation.*

Proof. We can prove the lemma by using techniques similar to those in [10], but the proof is omitted for lack of space. □

Theorem 2. *The algorithm of Fig. 3 runs in $O(height(\mathcal{D}) \cdot (\|\mathcal{D}\| + n) + n \cdot m + m^2 + r)$ time, using $O(\|\mathcal{D}\| + m^2)$ space. If $\mathcal{D}$ contains no truncation, the time complexity becomes $O(\|\mathcal{D}\| + n \cdot m + m^2 + r)$.*

4.3 Searching in BPE Compressed Files

We again take the two-dimensional array implementation for the functions $Jump_{\mathrm{BM}}$, $Output_{\mathrm{BM}}$, and *Shift*. Since the collage systems for BPE contain no truncation, the construction of the functions seems to require $O(\|\mathcal{D}\| + m^2)$ time and space in addition to $O(\|\mathcal{D}\| \cdot m)$ time and space. However, we can build them in another way, and have the following result.

Theorem 3. *The tables storing $Jump_{\mathrm{BM}}$, $Output_{\mathrm{BM}}$, and Shift can be built in $O(\|\mathcal{D}\| \cdot m)$ time and space, if $\mathcal{D}$ contains concatenation only.*

Proof. We can fill the entries of the tables in a bottom-up manner by using the directed acyclic word graph [7] for the reversed pattern. □

The computation of Δ stated in Section 4.2 requires knowing the lengths of the phrases represented by the skipped tokens. This slows down the searching speed. To do without such information, we assume they are all of length C, where C is the maximum phrase length, and let

$$\Delta(j, t) = \max\Big(\lceil \mathit{Shift}(0, t)/C \rceil, \lfloor \mathit{Shift}(j, t)/C \rfloor\Big).$$

The value of C is a crucial factor in such an approach, We estimated the change of compression ratios depending on C. The text files we used are:

Medline. A clinically-oriented subset of Medline, consisting of 348,566 references. The file size is 60.3 Mbyte and the entropy is 4.9647.

Genbank. The file consisting only of accession numbers and nucleotide sequences taken from a data set in Genbank. The file size is 17.1 Mbyte and the entropy is 2.6018.

Table 1 shows the compression ratios of these texts for BPE, together with those for the Huffman encoding, `gzip`, and `compress`, where the last two are well-known compression tools based on LZ77 and LZW, respectively. Remark that the change of compression ratios depending on C is non-monotonic. The reason for this is that the BPE compression routine we used builds a dictionary $\mathcal{D}$ in a greedy manner only from the first block of a text file. It is observed that we can restrict C with no great sacrifice of compression ratio. Thus we decided to use the BPE compressed file of Medline for $C = 3$, and that of Genbank for $C = 4$ in our experiment in the next section.

Table 1. Compression ratios (%).

	Huffman	BPE							`compress`	`gzip`
		$C = 3$	$C = 4$	$C = 5$	$C = 6$	$C = 7$	$C = 8$	unlimit.		
Medline	62.41	59.44	58.46	58.44	58.53	58.47	58.58	59.07	42.34	33.35
Genbank	33.37	36.93	32.84	32.63	32.63	32.34	32.28	32.50	26.80	23.15

5 Experimental Results

We estimated the performances of the following programs:

(A) *Decompression followed by ordinary search.*
We tested this approach with the KMP algorithm for the compression methods: `gzip`, `compress`, and BPE. We did not combine the decompression programs and the KMP search program using the Unix 'pipe' because it is slow. Instead, we embedded the KMP routine in the decompression programs, so that the KMP automaton processes the decoded characters 'on the fly'. The programs are abbreviated as `gunzip`+KMP, `uncompress`+KMP, and `unBPE`+KMP, respectively.

(B) *Ordinary search in original text.*
KMP, UT (the Uratani-Takeda variant [21] of BM), and `agrep`.

(C) *Compressed pattern matching.*
AC on LZW [12], Shift-Or on LZW [11], AC on Huffman [15], AC on BPE [19], and BM on BPE (the algorithm proposed in this paper).

The automata in KMP, UT, AC on Huffman, AC on BPE, and BM on BPE were realized as two-dimensional arrays of size $\ell \times 256$, where ℓ is the number of states. The texts used are Medline and Genbank mentioned in Section 4.3, and

the patterns searched for are text substrings randomly gathered from them. Our experiment was carried out on an AlphaStation XP1000 with an Alpha21264 processor at 667MHz running Tru64 UNIX operating system V4.0F. Figure 6 shows the running times (CPU time). We excluded the preprocessing times since they are negligible compared with the running times. We observed the following facts.

- The differences between the running times of KMP and the three programs of (A) correspond to the decompression times. Decompression tasks for LZ77 and LZW are thus time-consuming compared with pattern matching task. Even when we use the BM algorithm instead of KMP, the approach (A) is still slow for LZ77 and LZW.
- The proposed algorithm (BM on BPE) is faster than all the others. Especially, it runs about 1.2 times faster than `agrep` for Medline, and about 3 times faster for Genbank.

6 Conclusion

We have presented a BM type algorithm for compressed pattern matching in collage system, and shown that an instance of the algorithm searches in BPE compressed texts $1.2 \sim 3.0$ faster than `agrep` does in the original texts. For searching a very long pattern (e.g., $m > 30$), a simplified version of the backward-dawg-matching algorithm [7] is very fast as reported in [2]. To develop its compressed matching version will be our future work.

References

1. A. V. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Comm. ACM*, 18(6):333–340, 1975.
2. C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle, suffix oracle. Technical Report IGM-99-08, Institut Gaspard-Monge, 1999.
3. A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52:299–307, 1996.
4. R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Comm. ACM*, 35(10):74–82, 1992.
5. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20(10):62–72, 1977.
6. M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi. Text compression using antidictionaries. In *Proc. 26th Internationial Colloquim on Automata, Languages and Programming*, pages 261–270. Springer-Verlag, 1999.
7. M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, New York, 1994.
8. E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Direct pattern matching on compressed text. In *Proc. 5th International Symp. on String Processing and Information Retrieval*, pages 90–95. IEEE Computer Society, 1998.
9. P. Gage. A new algorithm for data compression. *The C Users Journal*, 12(2), 1994.

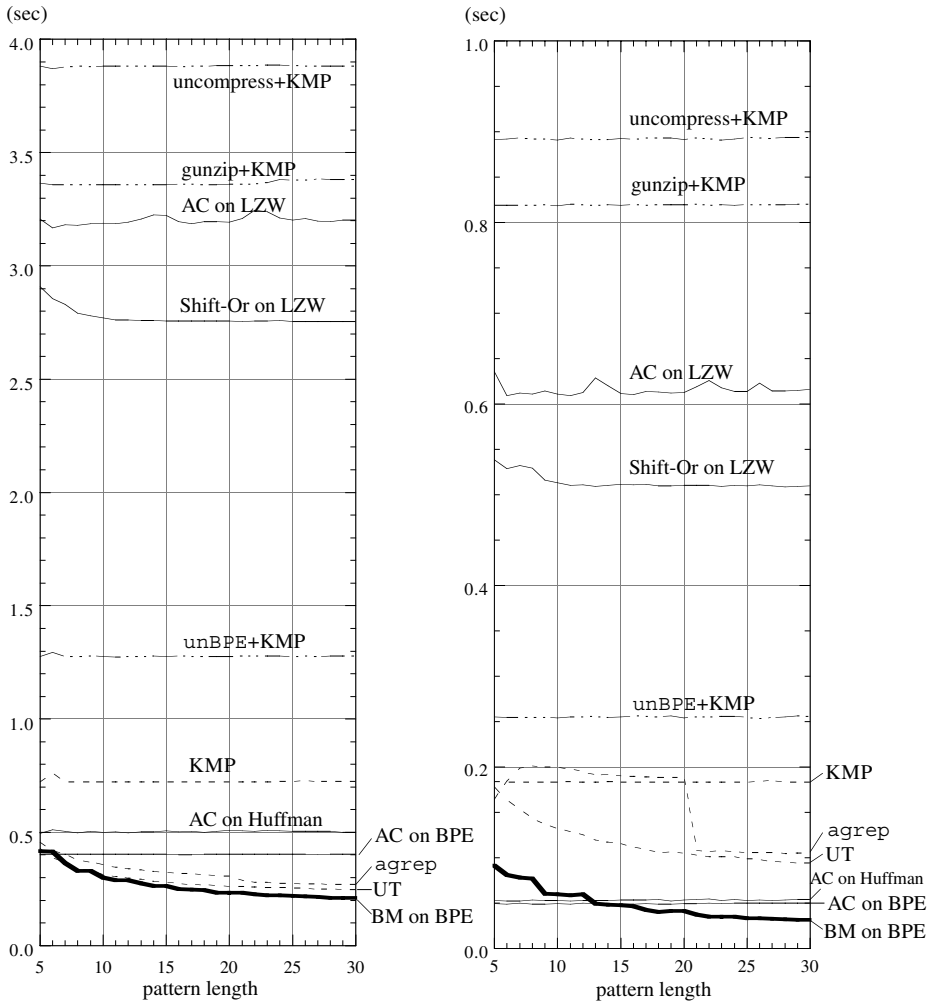


Fig. 6. Running times.

10. T. Kida, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. A unifying framework for compressed pattern matching. In *Proc. 6th International Symp. on String Processing and Information Retrieval*, pages 89–96. IEEE Computer Society, 1999.

11. T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Shift-And approach to pattern matching in LZW compressed text. In *Proc. 10th Ann. Symp. on Combinatorial Pattern Matching*, pages 1–13. Springer-Verlag, 1999.

12. T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In *Proc. Data Compression Conference (DCC'98)*, pages 103–112. IEEE Computer Society, 1998.

13. D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput*, 6(2):323–350, 1977.
14. U. Manber. A text compression scheme that allows fast searching directly in the compressed file. In *Proc. 5th Ann. Symp. on Combinatorial Pattern Matching*, pages 113–124. Springer-Verlag, 1994.
15. M. Miyazaki, S. Fukamachi, M. Takeda, and T. Shinohara. Speeding up the pattern matching machine for compressed texts. *Transactions of Information Processing Society of Japan*, 39(9):2638–2648, 1998. (in Japanese).
16. G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In *Proc. 10th Ann. Symp. on Combinatorial Pattern Matching*, pages 14–36. Springer-Verlag, 1999.
17. G. Navarro and J. Tarhio. Boyer–Moore string matching over Ziv-Lempel compressed text. In *Proc. 11th Ann. Symp. on Combinatorial Pattern Matching*. Springer-Verlag, 2000. to appear.
18. W. Rytter. Algorithms on compressed strings and arrays. In *Proc. 26th Ann. Conf. on Current Trends in Theory and Practice of Infomatics*. Springer-Verlag, 1999.
19. Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Speeding up pattern matching by text compression. In *Proc. 4th Italian Conference on Algorithms and Complexity*, pages 306–315. Springer-Verlag, 2000.
20. Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Pattern matching in text compressed by using antidictionaries. In *Proc. 10th Ann. Symp. on Combinatorial Pattern Matching*, pages 37–49. Springer-Verlag, 1999.
21. N. Uratani and M. Takeda. A fast string-searching algorithm for multiple patterns. *Information Processing & Management*, 29(6):775–791, 1993.
22. T. A. Welch. A technique for high performance data compression. *IEEE Comput.*, 17:8–19, June 1984.
23. S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Usenix Winter 1992 Technical Conference*, pages 153–162, 1992.
24. S. Wu and U. Manber. Fast text searching allowing errors. *Comm. ACM*, 35(10):83–91, October 1992.
25. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Inform. Theory*, IT-23(3):337–349, May 1977.

Approximate String Matching over Ziv–Lempel Compressed Text

Juha Kärkkäinen[1], Gonzalo Navarro[2⋆], and Esko Ukkonen[1]

[1] Dept. of Computer Science, University of Helsinki, Finland
{tpkarkka,ukkonen}@cs.helsinki.fi
[2] Dept. of Computer Science, University of Chile
gnavarro@dcc.uchile.cl

Abstract. We present a solution to the problem of performing approximate pattern matching on compressed text. The format we choose is the Ziv–Lempel family, specifically the LZ78 and LZW variants. Given a text of length u compressed into length n, and a pattern of length m, we report all the R occurrences of the pattern in the text allowing up to k insertions, deletions and substitutions, in $O(mkn + R)$ time. The existence problem needs $O(mkn)$ time. We also show that the algorithm can be adapted to run in $O(k^2 n + \min(mkn, m^2(m\sigma)^k) + R)$ average time, where σ is the alphabet size. The experimental results show a speedup over the basic approach for moderate m and small k.

1 Introduction

The *string matching problem* is defined as follows: given a pattern $P = p_1 \dots p_m$ and a text $T = t_1 \dots t_u$, find all the occurrences of P in T, i.e. return the set $\{|x|,\ T = xPy\}$. The complexity of this problem is $O(u)$ in the worst case and $O(u \log_\sigma(m)/m)$ on average (where σ is the alphabet size), and there exist algorithms achieving both time complexities using $O(m)$ extra space [8,3].

A generalization of the basic string matching problem is *approximate string matching*: an error threshold $k < m$ is also given as input, and we want to report all the ending positions of text substrings which match the pattern after performing up to k character insertions, deletions and replacements on them. Formally, we have to return the set $\{|xP'|,\ T = xP'y \text{ and } ed(P, P') \leq k\}$, where $ed(P, P')$ is the "edit distance" between both strings, i.e. the minimum number of character insertions, deletions and replacements needed to make them equal. The complexity of this problem is $O(u)$ in the worst case and $O(u(k + \log_\sigma(m))/m)$ on average. Both complexities have been achieved, despite that the space and preprocessing cost is exponential in m and k in the first case and polynomial in m in the second case. The best known worst case time complexity is $O(ku)$ if the space has to be polynomial in m (see [14] for a survey).

⋆ Work developed during postdoctoral stay at the University of Helsinki, partially supported by the Academy of Finland (grant (grant 44449) and Fundación Andes. Also supported by Fondecyt grant 1-990627.

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 195–209, 2000.

A particularly interesting case of string matching is related to text compression. Text compression [5] tries to exploit the redundancies of the text to represent it using less space. There are many different compression schemes, among which the Ziv–Lempel family [23,24] is one of the best in practice because of their good compression ratios combined with efficient compression and decompression time.

The *compressed matching problem* was first defined in the work of Amir and Benson [1] as the task of performing string matching in a compressed text without decompressing it. Given a text T, a corresponding compressed string $Z = z_1 \dots z_n$, and a pattern P, the compressed matching problem consists in finding all occurrences of P in T, using only P and Z. A naive algorithm, which first decompresses the string Z and then performs standard string matching, takes time $O(m+u)$. An optimal algorithm takes worst-case time $O(m+n+R)$, where R is the number of matches (note that it could be that $R = u > n$).

The compressed matching problem is important in practice. Today's textual databases are an excellent example of applications where both problems are crucial: the texts should be kept compressed to save space and I/O time, and they should be efficiently searched. Surprisingly, these two combined requirements are not easy to achieve together, as the only solution before the 90's was to process queries by uncompressing the texts and then searching into them. In particular, *approximate* searching on compressed text was advocated in [1] as an open problem.

This is the problem we solve in this paper: we present the first solution for compressed approximate string matching. The format we choose is the Ziv–Lempel family, focusing in the LZ78 and LZW variants. By modifying the basic dynamic programming algorithm, we achieve a time complexity of $O(mkn + R)$ and a space complexity of $O(n(mk + \log n))$ bits (i.e. $O(1 + mk/\log n)$ times the memory necessary to decompress). The existence problem needs $O(mkn)$ time and space. We show that the algorithm can be adapted to run in $O(k^2 n + \min(mkn, m^2(m\sigma)^k) + R)$ average time, where σ is the alphabet size.

Some experiments have been conducted to assess the practical interest of our approach. We have developed a variant of LZ78 which is faster to decompress in exchange for somewhat worse compression ratios. Using this compression format our technique can take less than 70% of the time needed by decompressing and searching on the fly with basic dynamic programming for moderate m and small k values. Dynamic programming is considered as the most flexible technique to cope with diverse variants of the problem. However, decompression followed by faster search algorithms specifically designed for the edit distance still outperforms our technique, albeit those algorithms are less flexible to cope with other variants of the problem.

2 Related Work

We consider in this work Ziv-Lempel compression, which is based on finding repetitions in the text and replacing them with references to similar strings

previously appeared. LZ77 [23] is able to reference any substring of the text already processed, while LZ78 [24] and LZW [21] reference only a single previous reference plus a new letter that is added. The first algorithm for exact searching is from 1994 [2], which searches in LZ78 needing time and space $O(m^2 + n)$.

The only search technique for LZ77 [9] is a randomized algorithm to determine in time $O(m + n \log^2(u/n))$ whether a pattern is present or not in the text (it seems that with $O(R)$ extra time they could find all the pattern occurrences).

An extension of [2] to multipattern searching was presented in [11], together with the first experimental results in this area. They achieve $O(m^2 + n)$ time and space, although this time m is the total length of all the patterns.

New practical results appeared in [16], who presented a general scheme to search on Ziv–Lempel compressed texts (simple and extended patterns) and specialized it for the particular cases of LZ77, LZ78 and a new variant proposed which was competitive and convenient for search purposes. A similar result, restricted to the LZW format, was independently found and presented in [12]. In [17] a new, faster, algorithm was presented based on Boyer-Moore.

The aim of this paper is to present a general solution to the approximate string matching problem on compressed text in the LZ78 and LZW formats.

3 Approximate String Matching by Dynamic Programming

We introduce some notation for the rest of the paper. A string S is a sequence of characters over an alphabet Σ. If the alphabet is finite we call σ its size. The length of S is denoted as $|S|$, therefore $S = s_1 \dots s_{|S|}$ where $s_i \in \Sigma$. A substring of S is denoted as $S_{i\dots j} = s_i s_{i+1} \dots s_j$, and if $i > j$, $S_{i\dots j} = \varepsilon$, the empty string of length zero. In particular, $S_i = s_i$. The pattern and the text, P and T, are strings of length m and u, respectively.

We recall that $ed(A, B)$, the edit distance between A and B, is the minimum number of characters insertions, deletions and replacements needed to convert A into B or vice versa. The basic algorithm to compute the edit distance between two strings A and B was discovered many times in the past, e.g. [18]. This was converted into a search algorithm much later [19]. We first show how to compute the edit distance between two strings A and B. Later, we extend that algorithm to search a pattern in a text allowing errors.

To compute $ed(A, B)$, a matrix $C_{0\dots|A|,0\dots|B|}$ is filled, where $C_{i,j}$ represents the minimum number of operations needed to convert $A_{1\dots i}$ to $B_{1\dots j}$. This is computed as $C_{i,0} = i$, $C_{0,j} = j$, and

$$C_{i,j} = \text{if } (A_i = B_j) \text{ then } C_{i-1,j-1} \text{ else } 1 + \min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1})$$

where at the end $C_{|A|,|B|} = ed(A, B)$.

We consider the text searching problem now. The algorithm is basically the same, with $A = P$ and $B = T$ (computing C column-wise so that $O(m)$ space is required). The only difference is that we must allow that any text position

is the potential start of a match. This is achieved by setting $C_{0,j} = 0$ for all $j \in 0 \ldots u$. That is, the empty pattern matches with zero errors at any text position (because it matches with a text substring of length zero).

The algorithm then initializes its column $\mathcal{C}_{0\ldots m}$ with the values $\mathcal{C}_i = i$, and processes the text character by character. At each new text character T_j, its column vector is updated to $\mathcal{C}'_{0\ldots m}$. The update formula is

$$\mathcal{C}'_i \quad = \quad \text{if } (P_i = T_j) \text{ then } \mathcal{C}_{i-1} \text{ else } 1 + \min(\mathcal{C}'_{i-1}, \mathcal{C}_i, \mathcal{C}_{i-1})$$

With this formula the invariant that holds after processing text position j is $\mathcal{C}_i = led(P_{1\ldots i}, T_{1\ldots j})$, where

$$led(A, B) \quad = \quad \min_{i \in 1\ldots|B|} ed(A, B_{i\ldots|B|})$$

that is, $\mathcal{C}_i$ is the minimum edit distance between $P_{1\ldots i}$ and a suffix of the text already seen. Hence, all the text positions where $\mathcal{C}_m \leq k$ are reported as ending points of occurrences.

The search time of this algorithm is $O(mu)$ and it needs $O(m)$ space.

The dynamic programming matrix has a number of properties that have been used to derive better algorithms. We are interested in two of them.

Property 1. *Let A and B be two strings such that $A = A_1A_2$. Then there exist strings B_1 and B_2 such that $B = B_1B_2$ and $ed(A, B) = ed(A_1, B_1) + ed(A_2, B_2)$.*

That is, there must be some point inside B where its optimal comparison against A can be divided at any arbitrary point in A. This is easily seen by considering an optimal path on the C matrix that converts A into B. The path must have at least one node in each row (and column), and therefore it can be split in a path leading to the cell $(|A_1|, r)$, for some r, and a path leading from that cell to $(|A|, |B|)$. Thus, $r = |B_1|$, which determines B_1. For example $ed(\text{"}\texttt{survey}\text{"}, \text{"}\texttt{surgery}\text{"}) = ed(\text{"}\texttt{surv}\text{"}, \text{"}\texttt{surg}\text{"}) + ed(\text{"}\texttt{ey}\text{"}, \text{"}\texttt{ery}\text{"})$.

The second property refers to the so-called *active* cells of the $\mathcal{C}$ vector when searching P allowing k errors. All the cells before and including the last one with value $\leq k$ are called "active". As noted in [20]:

Property 2. *The output of the search depends only on the active cells, and the rest can be assumed to be $k + 1$.*

Between consecutive iterations of the dynamic programming algorithm, the last active cell can be incremented at most in 1 (because neighboring cells of the C matrix differ at most in 1). Hence the last active cell can be maintained at $O(1)$ amortized time per iteration. The search algorithm needs to work only on the active cells. As conjectured in [20] and proved in [6,4], there are $O(k)$ active cells per column on average and therefore the dynamic programming takes $O(ku)$ time on average.

Considering Property 2, we use a modified version of *ed* in this paper. When we use $ed(A, B)$ we mean the exact edit distance between A and B if it is $\leq k$, otherwise any number larger than k can be returned. It is clear that the output of an algorithm using this definition is the same as with the original one.

4 A General Search Approach

We present now a general approach for approximate pattern matching over a text $Z = b_1 \dots b_n$, that is expressed as a sequence of n *blocks*. Each block b_r represents a substring B_r of T, such that $B_1 \dots B_n = T$. Moreover, each block B_r is formed by a concatenation of a previously seen blocks and an explicit letter. This comprises the LZ78 and LZW formats. Our goal is to find the positions in T where occurrences of P end with at most k errors, using Z.

Our approach is to adapt an algorithm designed to process T character by character so that it processes T block by block, using the fact that blocks are built from previous blocks and explicit letters. In this section we show how have we adapted the classical dynamic programming algorithm of Section 3. We show later that the $O(ku)$ algorithm based on active cells can be adapted as well.

We need a little more notation before explaining the algorithm. Each match is defined as either *overlapping* or *internal*. A match j is internal if there is an occurrence of P ending at j totally contained in some block B_r (i.e. if the block repeats the occurrence surely repeats). Otherwise it is an overlapping match. We also define $b^{(r)}$, for a block b and a natural number r, as follows: $b^{(0)} = b$ and $b^{(r+1)} = (b')^{(r)}$, where b' is the block referenced by b. That is, $b^{(r)}$ is the block obtained by going r steps in the backward referencing chain of b.

The general mechanism of the search is as follows: we read the blocks b_r one by one. For each new block b read, representing a string B, and where we have already processed $T_{1\dots j}$, we update the state of the search so that after working on the block we have processed $T_{1\dots j+|B|} = T_{1\dots j}B$. To process each block, three steps are carried out: (1) its *description* (to be specified shortly) is computed, (2) the occurrences ending inside the block B are reported, and (3) the state of the search is updated. The state of the search consists of two elements

- The last text position considered, j (initially 0).
- A vector $\mathcal{C}_i$, for $i \in 0 \dots m$, where $\mathcal{C}_i = led(P_{1\dots i}, T_{1\dots j})$. Initially, $\mathcal{C}_i = i$. This vector is the same as for plain dynamic programming.

The description of all the blocks already seen is maintained. Say that block b represents the text substring B. Then the description of b is formed by the length $len(b) = |B|$, the referenced block $ref(b) = b^{(1)}$ and some vectors indexed by $i \in 1 \dots m$ (their values are assumed to be $k+1$ if accessed outside bounds).

- $\mathcal{I}_{i,i'}(b) = ed(P_{i\dots i'}, B)$, for $i \in 1 \dots m$, $i' \in \max(i+|B|-k-1, i-1) \dots \min(i+|B|+k-1, m)$, which at each point gives the edit distance between B and $P_{i\dots i'}$. Note that $\mathcal{I}$ has $O(mk)$ entries per block. In particular, the set of possible i' values is empty if $i > m+k+1-|B|$, in which case $\mathcal{I}_{i,i'}(b) = k+1$.
- $\mathcal{P}_i(b) = led(P_{1\dots i}, B)$, for $i \in 1 \dots m$, gives the edit distance between the prefix of length i of P and a suffix of B. $\mathcal{P}$ has $O(m)$ entries per block.
- $\mathcal{M}(b) = b^{(r)}$, where $r = \min\{r' \geq 0,\ \mathcal{P}_m(b^{(r')}) \leq k\}$. That is, $\mathcal{M}(b)$ is the last block in the referencing chain for b that finishes with an internal match of P. Its value is -1 if no such block exists.

Figure 1 illustrates the $\mathcal{I}$ matrix and how is it filled under different situations.

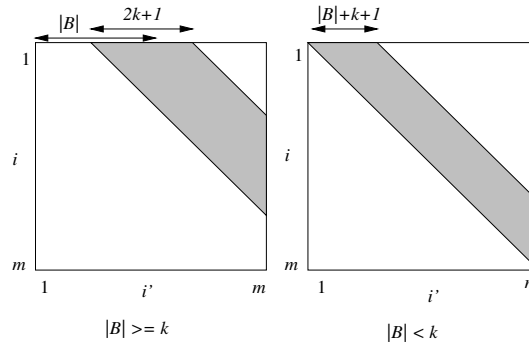


Fig. 1. The $\mathcal{I}$ matrix for a block b representing a string B.

5 Computing Block Descriptions

We show how to compute the description of a new block b' that represents $B' = Ba$, where B is the string represented by a previous block b and a is an explicit letter. The procedure is almost the same as for LZW so we omit it here and concentrate on LZ78 only. An initial block b_0 represents the string ε, and its description is: $len(b_0) = 0$; $\mathcal{I}_{i,i'}(b_0) = i' - i + 1$, $i \in 1 \ldots m, i' \in i - 1 \ldots \min(i + k - 1, m)$; $\mathcal{P}_i(b_0) = i$, $i \in 1 \ldots m$; and $\mathcal{M}(b_0) = -1$.

We give now the update formulas for the case when a new letter a is added to B in order to form B'. These can be seen as special cases of dynamic programming matrices between B and parts of P.

- $len(b') = len(b) + 1$.
- $ref(b') = b$.
- $\mathcal{I}_{i,i'}(b') = \mathcal{I}_{i,i'-1}(b)$ if $a = P_{i'}$, and $1 + \min(\mathcal{I}_{i,i'}(b), \mathcal{I}_{i,i'-1}(b'), \mathcal{I}_{i,i'-1}(b))$ otherwise. We start with[1] $\mathcal{I}_{i,\max(i-1,i+|B'|-k-2)}(b') = \min(|B'|, k+1)$, and compute the values for increasing i'. This corresponds to filling a dynamic programming matrix where the characters of $P_{i\ldots}$ are the columns and the characters of B are the rows. Adding a to B is equivalent to adding a new row to the matrix, and we store at each block only the row of the matrix corresponding to its last letter (the rest could be retrieved by going back in the references). For each i, there are $2k + 1$ such columns stored at each block B, corresponding to the interesting i' values. Figure 2 illustrates. To relate this to the matrix of $\mathcal{I}$ in Figure 1 one needs to consider that there is a three dimensional matrix indexed by i, i' and $|B|$. Figure 1 shows the plane stored at each block B, corresponding to its last letter. Figure 2 shows a plane obtained by fixing i.
- $\mathcal{P}_i(b') = \mathcal{P}_{i-1}(b)$ if $a = P_i$ and $1 + \min(\mathcal{P}_i(b), \mathcal{P}_{i-1}(b'), \mathcal{P}_{i-1}(b))$ otherwise. We assume that $\mathcal{P}_0(b') = 0$ and compute the values for increasing i. This

[1] Note that it may be that this initial value cannot be placed in the matrix because its position would be outside bounds.

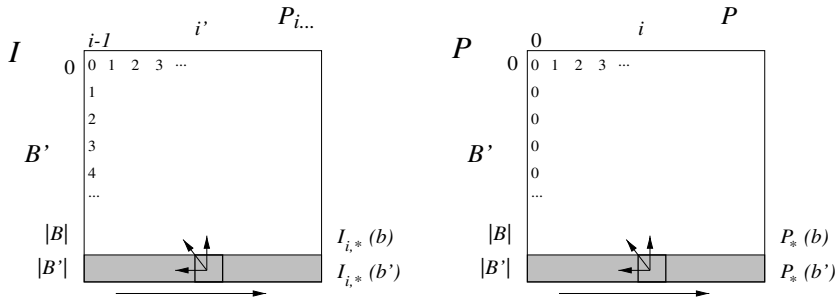


Fig. 2. The virtual dynamic programming matrices. On the left, between B and $P_{i...}$, to compute $\mathcal{I}$. On the right, between B and P, to compute $\mathcal{P}$.

corresponds again to filling a dynamic programming matrix where the characters of P are the columns, while the characters of B are the rows. The (virtual) matrix has i at the i-th column of the first row and zeros in the first column. Figure 2 illustrates.

– $\mathcal{M}(b') = \mathcal{M}(b)$ if $\mathcal{P}_m(b') > k$, and b' otherwise. That is, if there is a new internal match ending at $|B'|$ then b' is added to the list. This takes constant time.

6 Updating the Search State

We specify now how to report the matches and update the state of the search once the description of a new block b has been computed. Three actions are carried out, in this order.

Reporting the Overlapping Matches. An overlapping match ending inside the new block B corresponds to an occurrence that spans a suffix of the text already seen $T_{1...j}$ and a prefix of B. From Property 1, we know that if such an occurrence matches P with k errors (or less) then it must be possible to split P in $P_{1...i}$ and $P_{i+1...m}$, such that the text suffix matches the first half and the prefix of B matches the second half. Figure 3 illustrates.

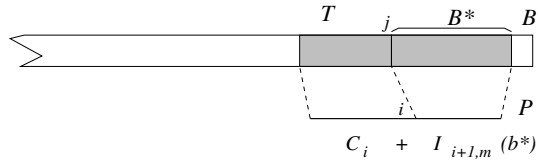


Fig. 3. Splitting of an overlapping match (grayed), where $b^* = b^{(|B|-i')}$.

Therefore, all the possible overlapping matches are found by considering all the possible positions i in the pattern. The check for a match ending at text

position $j + i'$ is then split into two parts. A first condition states that $P_{1...i}$ matches a suffix of $T_{1...j}$ with k_1 errors, which can be checked using the $\mathcal{C}$ vector. A second condition states that $P_{i+1...m}$ matches $B_{1...i'}$ with k_2 errors, which can be checked using the $\mathcal{I}$ matrix of previous referenced blocks. Finally, $k_1 + k_2$ must be $\leq k$.

Summarizing, the text position $j + i'$ (for $i' \in 1 \ldots \min(m + k - 1, |B|)$) is reported if

$$\min_{i=\min(1,m-i'-k)}^{\max(m-1,m-i'+k)} (\mathcal{C}_i + \mathcal{I}_{i+1,m}(b^{(|B|-i')})) \quad \leq \quad k \tag{1}$$

where we need $ref(\cdot)$ to compute $b^{(|B|-i')}$. Additionally, we have to report the positions $j + i'$ such that $\mathcal{C}_m + i' \leq k$ (for $i' \in 1 \ldots k$). This corresponds to $\mathcal{I}_{m+1,m}(b^{(|B|-i')}) = i'$, which is not stored in that matrix.

Note that if we check the i' positions in decreasing order then the backward reference chain has to be traversed only once. So the total cost for this check is $O(mk)$. The occurrences are not immediately reported but stored in decreasing order of i' in an auxiliary array (of size at most $m + k$), because they can mix and collide with internal matches.

Reporting the Internal Matches. These are matches totally contained inside B. Their offsets can be retrieved by following the $\mathcal{M}(b)$ list until reaching the value -1. That is, we start with $b' \leftarrow \mathcal{M}(b)$ and while $b' \neq -1$ report the positions $j+len(b')$ and update $b' \leftarrow \mathcal{M}(b')$. This retrieves all the internal matches in time proportional to their amount, in reverse order. These matches may collide and intermingle with the overlapping matches. We merge both chains of matches and report them in increasing order and without repetitions. All this can be done in time proportional to the number of matches reported (which adds up $O(R)$ across all the search).

Updating the $\mathcal{C}$ Vector and j. To update $\mathcal{C}$ we need to determine the best edit distance between $P_{1...i}$ and a suffix of the new text $T_{1...j+|B|} = T_{1...j}B$. Two choices exist for such a suffix: either it is totally inside B or it spans a suffix of $T_{1...j}$ and the whole B. Figure 4 illustrates the two alternatives. The first case corresponds to a match of $P_{1...i}$ against a suffix of B, which is computed in $\mathcal{P}$. For the second case we can use Property 1 again to see that such an occurrence is formed by matching $P_{1...i'}$ against some suffix of $T_{1...j}$ and $P_{i'+1...i}$ against the whole B. This can be solved by combining $\mathcal{C}$ and $\mathcal{I}$.

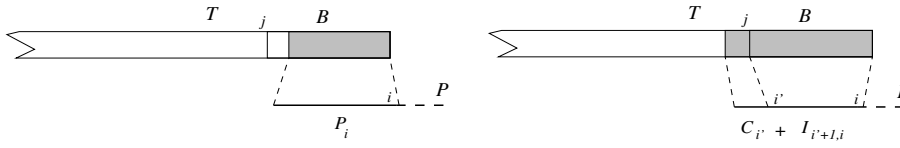


Fig. 4. Two choices to update the $\mathcal{C}$ vectors.

The formula to update $\mathcal{C}$ to a new $\mathcal{C}'$ is therefore

$$\mathcal{C}'_i \leftarrow \min(\mathcal{P}_i(b), \min_{i'=\max(1,i-|B|-k)}^{\min(i-1,i-|B|+k)}(\mathcal{C}_{i'} + \mathcal{I}_{i'+1,i}(b))) \quad (2)$$

which finds the correct value if it is not larger than k, and gives something larger than k otherwise (this is in accordance to our modified definition of ed). Since there are m cells to compute and each one searches over at most $2k+1$ values, the total cost to update $\mathcal{C}$ is $O(mk)$.

Finally, j is easily updated by adding $|B|$ to it.

7 Complexity

The space requirement for the algorithm is basically that to store the block descriptions. The lengths $len(\cdot)$ add up u, so in the worst case $n\log(u/n) = O(n\log n)$ bits are necessary to store them. The references to the previous blocks $ref(\cdot)$ and $\mathcal{M}(\cdot)$ need $O(\log n)$ bits per entry, for a total of $O(n\log n)$ bits. For the matrices, we observe that each element of those arrays differs from the previous one in at most 1, that is $|\mathcal{I}_{i,i'+1}(b) - \mathcal{I}_{i,i'}(b)| \leq 1$ and $|\mathcal{P}_{i+1}(b) - \mathcal{P}_i(b)| \leq 1$. Their first value is trivial and does not need to be stored. Therefore, each such cell can be represented only with 2 bits, for a total space requirement of $O(mkn)$ bits.

With respect to time complexity, all the processes described take $O(mkn)$ time for the existence problem and $O(mkn+R)$ time to report the R matches of P. The update of $\mathcal{P}$ costs $O(m)$ per block, but that of $\mathcal{I}$ takes $O(mk)$. The same happens to finding the overlapping matches and the update of $\mathcal{C}$. The handling of internal matches and merging with overlapping matches add up $O(R)$ along the total process.

8 A Faster Algorithm on Average

A simple form to speed up the dynamic programming algorithm on compressed text is based on Property 2. That is, we try to work only on the active cells of $\mathcal{C}$, $\mathcal{I}$ and $\mathcal{P}$.

We update only the active cells of $\mathcal{C}$ and $\mathcal{P}$. If we assume a random pattern P, then the property that says that there are on average $O(k)$ active cells in $\mathcal{C}$ at a random text position holds also when those text positions are the endpoints of blocks in the text. The same happens to the $\mathcal{P}$ values, since $\mathcal{P}_i(b) \geq \mathcal{C}_i$ after processing block b.

We recall the minimization formula (2) to update $\mathcal{C}$, and note that the $\mathcal{C}_{i'}$ are on average active only for $i' = O(k)$. Therefore only the values $i \in |B| \pm O(k)$ have a chance of being $\leq k$. The minimization with $\mathcal{P}_i$ does not change things because this vector has also $O(k)$ active values on average.

Therefore, updating $\mathcal{C}$ costs $O(k^2)$ per block on average. Computing $\mathcal{P}$ takes $O(k)$ time since only the active part of the vector needs to be traversed.

A more challenging problem appears when trying to apply the technique to $\mathcal{I}_{i,i'}(b)$. They key idea in this case comes from considering that $ed(P_{i...i'}, B) > k$ if $|B| - (i' - i + 1) > k$, and therefore any block B such that $|B| > m + k$ cannot have any active value in $\mathcal{I}$. Since there are at most $O(\sigma^{m+k})$ different blocks of length at most $m+k$ (recall that σ is the alphabet size of the text), we can work $O(mk\sigma^{m+k})$ in total in computing $\mathcal{I}$ values. This is obtained by marking the blocks that do not have any active value in their $\mathcal{I}$ matrix, so that the $\mathcal{I}$ matrix of the blocks referencing them do not need to be computed either (moreover, the computation of $\mathcal{C}$ and overlapping matches can be simplified).

However, this bound can be improved. The set of different strings matching a pattern P with at most k errors, called $U_k(P) = \{P',\ ed(P, P') \leq k\}$, is finite. More specifically, it is shown in [20] that if $|P| = m$, then $|U_k(P)| = O((m\sigma)^k)$ at most. This limits the total number of different blocks B that can be preprocessed for a given pattern substring $P_{i...i'}$. Summing over all the possible substrings $P_{i...i'}$, considering that computing each such entry for each block takes $O(1)$ time, we have a total cost of $O(m^2(m\sigma)^k)$. Note that this is a worst case result, not only average case. Another limit for the total amount of work is still $O(mkn)$, so the cost is $O(\min(mkn, m^2(m\sigma)^k))$.

Finally, we have to consider the cost of processing the matches. This is $O(R)$ plus the cost to search the overlapping matches. We recall the formula (1) to find them, which can be seen to cost $O(k^2)$ only, since there are $O(k)$ active values in $\mathcal{C}$ on average and therefore $i' \in m \pm O(k)$ is also limited to $O(k)$ different values.

Summarizing, we can solve the problem on LZ78 and LZW in $O(k^2 n + \min(mkn, m^2(m\sigma)^k) + R)$ average time. Note in particular that the middle term is asymptotically independent on n. Moreover, the space required is $O(k^2 n + \min(mkn, m^2(m\sigma)^k) + n \log n)$ bits because only the relevant parts of the matrices need to be stored.

9 Significance of the Results

9.1 Memory Requirements

First consider the space requirements. In the worst case we need $O(n(mk+\log n))$ bits. Despite that this may seem impractical, this is not so. A first consideration is that normal compressors use only a suffix ("window") of the text already seen, in order to use bounded memory independent of n. The normal mechanism is that when the number of nodes in the LZ78 trie reaches a given amount N, the trie is deleted and the compression starts again from scratch for the rest of the file. A special mark is left in the compressed file to let the decompressor know of this fact.

Our search mechanism can use the same mark to start reusing its allocated memory from zero as well, since no node seen in the past will be referenced again. This technique can be adapted to more complex ways of reusing the memory under various LZ78-like compression schemes [5].

If a compressor is limited to use N nodes, the decompression needs at the very least $O(N \log N)$ bits of memory. Since the search algorithm can be restarted after reading N blocks, it requires only $O(N(mk + \log N))$ bits. Hence the amount of memory required to search is $O(1 + mk/\log N) \times$ memory for decompression, and we recall that this can be lowered in the average case. Moreover, reasonably fast decompression needs to keep the decompressed text in memory, which increases its space requirements.

9.2 Time Complexity

Despite that ours is the first algorithm for approximate searching on compressed text, there exist also alternative approaches, some of them trivial and others not specifically designed for approximate searching.

The first alternative approach is DS, a trivial decompress-then-search algorithm. This yields, for the worst case, $O(ku)$ [10] or $O(m|U_k(P)| + u)$ [20] time, where we recall that $|U_k(P)|$ is $O((m\sigma)^k)$. For the average case, the best result in theory is $O(u + (k + \log_\sigma m)u/m) = O(u)$ [7]. This is competitive when u/n is not large, and it needs much memory for fast decompression.

A second alternative approach, OM, considers that all the overlapping matches can be obtained by decompressing the first and last $m + k$ characters of each block, and using any search algorithm on that decompressed text. The internal matches are obtained by copying previous results. The total amount of text to process is $O(mn)$. Using the previous algorithms, this yields worst case times of $O(kmn + R)$ and $O(m|U_k(P)| + mn + R)$ in the worst case, and $O((k + \log_\sigma m)n + mn + R) = O(mn + R)$ on average. Except for large u/m, it is normally impractical to decompress the first and last $m + k$ characters of each block.

Yet a third alternative, MP, is to reduce the problem to multipattern searching of all the strings in $U_k(P)$. As shown in [11], a set of strings of *total* length M can be searched in $O(M^2 + n)$ time and space in LZ78 and LZW compressed text. This yields an $O(m^2|U_k(P)|^2 + n + R)$ worst case time algorithm, which for our case is normally impractical due to the huge preprocessing cost.

Table 1 compares the complexities. As can be seen, our algorithm yields the best average case complexity for

$$k = O(\sqrt{u/n}) \quad \wedge \quad k = O(\sqrt{m}) \quad \wedge \quad \frac{\log_\sigma n}{2(1 + \log_\sigma m)} \leq k + O(1) \leq \frac{\log_\sigma n}{1 + \log_\sigma m}$$

where essentially the first condition states that the compressed text should be reasonably small compared to the uncompressed text (this excludes DS), the second condition states that the number of errors should be small compared to the pattern length (this excludes OM) and the third condition states that n should be large enough to make $|U_k(P)|$ not significant but small enough to make $|U_k(P)|^2$ significant (this excludes MP). This means in practice that our approach is the fastest for short and medium patterns and low error levels.

Algorithm	Worst case time	Average case time
DS	ku $m\|U_k(P)\| + u$	u
OM	$kmn + R$ $m\|U_k(P)\| + mn + R$	$mn + R$
MP	$m^2\|U_k(P)\|^2 + n$	$m^2\|U_k(P)\|^2 + n + R$
Ours	$kmn + R$	$k^2n + \min(mkn, m^2\|U_k(P)\|) + R$

Table 1. Worst and average case time for different approaches.

9.3 Experimental Results

We have implemented our algorithm in order to determine its practical value. Our implementation does not store the matrix values using 2 bit deltas, but their full values are stored in whole bytes (this works for $k < 255$). The space is further reduced by not storing the information on blocks that are not to be referenced later. In LZ78 this discards all the leaves of the trie. Of course a second compression pass is necessary to add this bit to each compressed code. Now, if this is done then we can even not assign a number to those nodes (i.e. the original nodes are renumbered) and thus reduce the number of bits of the backward pointers. This can reduce the effect of the extra bit and reduces the memory necessary for decompression as well.

We ran our experiments on a Sun UltraSparc-1 of 167 MHz and 64 Mb of RAM. We have compressed two texts: WSJ (10 Mb of Wall Street Journal articles) and DNA (10 Mb of DNA text with lines cut every 60 characters). We use an ad-hoc LZ78 compressor which stores the pair (s, a) corresponding to the backward reference and new character in the following form: s is stored as a sequence of bytes where the last bit is used to signal the end of the code; and a is coded as a whole byte. Compression could be further reduced by better coding but this would require more time to read the compressed file. The extra bit indicating whether each node is going to be used again or not is added to s, i.e. we code $2s$ or $2s + 1$ to distinguish among the two possibilities.

Using the plain LZ78 format, WSJ was reduced to 45.02% of its original size, while adding the extra bit to signal not referenced blocks raised this percentage to 45.46%, i.e. less than 1% of increment. The figures for DNA were 39.69% and 40.02%. As a comparison, Unix *Compress* program, an LZW compressor that uses bit coding, obtained 38.75% and 27.91%, respectively.

We have compared our algorithm against a more practical version of DS, which decompresses the text on the fly and searches over it, instead of writing it to a new decompressed file and then reading it again to search. The search algorithm used is that based on active columns (the one we adapted). This gives us a measure of the improvement obtained over the algorithm we are transforming.

It is also interesting to compare our technique against decompression plus searching using the best available algorithm. For this alternative (which we call "Best" in the experiments) we still use our compression format, because it de-

compresses faster than Gnu *gzip* and Unix *compress*. Our decompression times are 2.09 seconds for WSJ and 1.80 for DNA. The search algorithms used are those of [15,4,13], which were the fastest for different m and k values in our texts.

On the other hand, the OM-type algorithms are unpractical for typical compression ratios (i.e. u/n at most 10) because of their need to keep count of the $m + k$ first and last characters of each block. The MP approach does not seem practical either, since for $m = 10$ and $k = 1$ it has to generate an automaton of more than one million states at the very least. We tested the code of [11] on our text and it took 5.50 seconds for just one pattern of $m = 10$, which outrules it in our cases of interest.

We tested $m = 10$, 20 and 30, and $k = 1$, 2 and 3. For each pattern length, we selected 100 random patterns from the text and used the same patterns for all the algorithms. Table 2 shows the results.

WSJ									
k	Ours $m = 10$	DS $m = 10$	Best $m = 10$	Ours $m = 20$	DS $m = 20$	Best $m = 20$	Ours $m = 30$	DS $m = 30$	Best $m = 30$
1	3.77	4.72	2.40	3.23	4.64	2.28	3.08	4.62	2.27
2	5.63	5.62	2.74	4.72	5.46	2.42	6.05	5.42	2.33
3	11.60	6.43	3.64	9.17	6.29	2.75	13.56	6.22	2.44
DNA									
k	Ours $m = 10$	DS $m = 10$	Best $m = 10$	Ours $m = 20$	DS $m = 20$	Best $m = 20$	Ours $m = 30$	DS $m = 30$	Best $m = 30$
1	3.91	5.21	2.66	2.49	5.08	2.46	2.57	5.06	2.51
2	6.98	6.49	2.88	3.81	6.31	2.91	5.02	6.28	2.71
3	11.51	8.91	3.08	9.28	7.51	3.24	15.35	7.50	3.18

Table 2. CPU times to search over the WSJ and DNA files.

As the table shows, we can actually improve over the decompression of the text and the application of the *same* search algorithm. In practical terms, we can search the original file at about 2.6 . . . 4.0 Mb/sec when $k = 1$, while the time keeps reasonable and competitive for $k = 2$ as well. Moreover, DS needs for fast decompression to store the uncompressed file in main memory, which could pose a problem in practice.

On the other hand, the "Best" option is faster than our algorithm, but we recall that this is an algorithm specialized for edit distance. Dynamic programming is unbeaten in its flexibility to accommodate other variants of the approximate string matching problem.

10 Conclusions

We have presented the first solution to the open problem of approximate pattern matching over Ziv–Lempel compressed text. Our algorithm can find the R occurrences of a pattern of length m allowing k errors over a text compressed by LZ78 or LZW into n blocks in $O(kmn + R)$ worst-case time and $O(k^2 n + \min(mkn, m^2(m\sigma)^k) + R)$ average case time. We have shown that this is of theoretical and practical interest for small k and moderate m values.

Many theoretical and practical questions remain open. A first one is whether we can adapt an $O(ku)$ worst case time algorithm (where u is the size of the uncompressed text) instead of the dynamic programming algorithm we have selected, which is $O(mu)$ time. This could yield an $O(k^2 n + R)$ worst-case time algorithm. Our efforts to adapt one of these algorithms [10] yielded the same $O(mkn + R)$ time we already have.

A second open question is how can we improve the search time in practice. For instance, we have not implemented the version that stores 2 bits per number, which could reduce the space. The updates to $\mathcal{P}$ and $\mathcal{I}$ could be done using bit-parallelism by adapting [13]. We believe that this could yield improvements for larger k values. On the other hand, we have not devised a bit-parallel technique to update $\mathcal{C}$ and to detect overlapping matches. Another idea is to map all the characters not belonging to the pattern to a unique symbol at search time, to avoid recomputing similar states. This, however, requires a finer tracking of the trie of blocks to detect also descendants of similar states. This yields a higher space requirement.

A third question is if faster filtration algorithms can be adapted to this problem without decompressing all the text. For example, the filter based in splitting the pattern in $k + 1$ pieces, searching the pieces without errors and running dynamic programming on the text surrounding the occurrences [22] could be applied by using the multipattern search algorithm of [11]. In theory the complexity is $O(m^2 + n + ukm^2/\sigma^{\lfloor m/(k+1) \rfloor})$, which is competitive for $k < m/\Theta(\log_\sigma(u/n) + \log_\sigma m)$.

References

1. A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proc. DCC'92*, pages 279–288, 1992.
2. A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *J. of Comp. and Sys. Sciences*, 52(2):299–307, 1996. Earlier version in *Proc. SODA'94*.
3. A. Apostolico and Z. Galil. *Pattern Matching Algorithms*. Oxford University Press, Oxford, UK, 1997.
4. R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.
5. T. Bell, J. Cleary, and I. Witten. *Text Compression*. Prentice Hall, 1990.
6. W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. CPM'92*, LNCS 644, pages 172–181, 1992.

7. W. Chang and T. Marr. Approximate string matching and local similarity. In *Proc. CPM'94*, LNCS 807, pages 259–273, 1994.
8. M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, Oxford, UK, 1994.
9. M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20:388–404, 1998. Previous version in *STOC'95*.
10. Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM J. on Computing*, 19(6):989–999, 1990.
11. T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In *Proc. DCC'98*, pages 103–112, 1998.
12. T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Shift-And approach to pattern matching in LZW compressed text. In *Proc. CPM'99*, LNCS 1645, pages 1–13, 1999.
13. G. Myers. A fast bit-vector algorithm for approximate pattern matching based on dynamic progamming. In *Proc. CPM'98*, LNCS 1448, pages 1–13, 1998.
14. G. Navarro. A guided tour to approximate string matching. Technical Report TR/DCC-99-5, Dept. of Computer Science, Univ. of Chile, 1999. To appear in *ACM Computing Surveys*.
`ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/survasm.ps.gz`.
15. G. Navarro and R. Baeza-Yates. Very fast and simple approximate string matching. *Information Processing Letters*, 72:65–70, 1999.
16. G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In *Proc. CPM'99*, LNCS 1645, pages 14–36, 1999.
17. G. Navarro and J. Tarhio. Boyer-Moore string matching over Ziv-Lempel compressed text. In *Proc. CPM'2000*, LNCS 1848, 2000, pp. 166–180. In this same volume.
18. S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. of Molecular Biology*, 48:444–453, 1970.
19. P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.
20. E. Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.
21. T. A. Welch. A technique for high performance data compression. *IEEE Computer Magazine*, 17(6):8–19, June 1984.
22. S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, 1992.
23. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23:337–343, 1977.
24. J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Trans. Inf. Theory*, 24:530–536, 1978.

Improving Static Compression Schemes by Alphabet Extension

Shmuel T. Klein

Department of Mathematics and Computer Science
Bar Ilan University
Ramat-Gan 52900, Israel
tomi@cs.biu.ac.il

Abstract. The performance of data compression on a large static text may be improved if certain variable-length strings are included in the character set for which a code is generated. A new method for extending the alphabet is presented, based on a reduction to a graph-theoretic problem. A related optimization problem is shown to be NP-complete, a fast heuristic is suggested, and experimental results are presented.

1 Introduction and Background

Compression methods may be classified according to various criteria, such as compression efficiency, speed, necessary space in RAM, etc (see [17] or [3] for an overview). The present work focuses on a static compression scheme, for which we assume that the time restrictions on the encoding and decoding processes are not symmetrical: we shall put no constraint on compression time, but require very fast decompression. The corresponding scenario is that of a large full-text information retrieval (IR) system, for which compression is usually performed only once or periodically (but rarely), but which may be accessed frequently, so that (possibly many) small parts of it need be decompressed while the user is waiting for responses to his query. The text of the IR system may be static, like for the Encyclopædia Britannica, the *Responsa Retrieval Project* [8], the *Trésor de la Langue Française* [6], etc., or it might be dynamically increasing over time, like collections of news wire messages, newspapers, and ultimately, the whole World Wide Web.

Statistical compression methods, like Huffman or arithmetic coding, assume that the text at hand is a sequence of data items, that will be encoded according to their occurrence frequencies. These data items are generally single characters, but sometimes more sophisticated models are used for which a character pair, a word, or more generally, certain character strings, are considered to be an item. *Dictionary* compression methods work by replacing the occurrence of certain strings in the text by (shorter) pointers to some dictionary, which again may be either static, fixed in advance (e.g., the most frequent words in English),

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 210–221, 2000.

or dynamically adapted to the given text, like the various Lempel-Ziv methods and their variants. The statistical methods working only on single characters are often inferior to the dictionary methods from the compression point of view. Nevertheless, they may be the preferred choice in IR applications, as they do not require a sequential scan of the compressed file.

To improve the performance of the statistical methods, we shall follow the approach in [4] and [5], and construct a *meta-alphabet*, which extends the standard alphabet by including also frequent variable-length strings. The idea is that even if we assume a quite involved model for the text generation process, there will always be strings deviating from this model. One may thus improve the accuracy of the model by including such strings as indivisible items, called below *meta-characters,* into the extended alphabet.

Obviously, the larger the meta-alphabet, the better compression we may expect. The problem then is that of an optimal exploitation of a given amount of RAM, which puts a bound on the size of the dictionary. In addition, there is a problem with the selection of the set of meta-characters, because the potential strings are overlapping. A similar problem has been shown to be NP-complete under certain constraints [9]. We shall thus proceed as follows: in the next section, we present a novel approach to alphabet extension and show that a related optimization problem is NP-complete. Section 3 deals with implementation issues and refinements and suggests a fast heuristic. Finally, we bring some preliminary experimental results in Section 4.

2 Definition of Meta-alphabet

The criterion for including a string of characters as a new meta-character into the meta-alphabet is the expected savings we would incur by its inclusion. The exact value is hard to evaluate, since the savings depend ultimately on the encoding method, and for Huffman codes, say, the length of each codeword may depend on all the others. Assume for simplicity in a first stage, that we shall use a fixed length code to refer to any element of the meta-alphabet $\mathcal{A}$, which contains both single characters and the meta-characters. If $|\mathcal{A}| = D$, any element can be encoded by $\lceil \log_2(D) \rceil$ bits.

Our approach starts by a reduction to a graph-theoretic problem in the following way. There will be a vertex for every possible character string in the text, and vertices are connected by an edge if the corresponding strings are overlapping. Both vertices and edges are assigned weights. The weight $w(x)$ of a vertex x will be the savings, measured in number of characters, obtained by including x in $\mathcal{A}$. The weight $w(x, y)$ of an edge (x, y) will be the loss of such savings due to the overlap between x and y. We are thus interested in a subset V' of the vertices, not larger than some predetermined constant K, which maximizes the overall savings

$$\sum_{x \in V'} w(x) \; - \sum_{x,y \in V'} w(x, y). \tag{1}$$

Formally, we are given a text $T = t_1 t_2 \cdots t_n$, where each t_i belongs to a finite alphabet Σ; define a directed graph $G = (V, E)$, where V is the set of all the substrings of T, i.e., the strings $t_i \cdots t_j$ for $1 \leq i \leq j \leq n$, and there is a directed edge from $x = x_1 \cdots x_k$ to $y = y_1 \cdots y_\ell$ if some suffix of x is a prefix of y, i.e., there is a $t \geq 1$ such that $x_{k-t+j} = y_j$ for all $1 \leq j \leq t$. For example, there will be a directed edge from the string `element` to `mention`, corresponding to $t = 4$. The weight of a vertex x is defined as

$$w(x) = freq(x)(|x| - 1),$$

where $freq(x)$ is the number of occurrences of the string x in T, $|a|$ denotes the length (number of characters) of a string a, and the -1 accounts for the fact that for each occurrence of x, $|x|$ characters are saved, but one meta-character will be used, so we save only $|x| - 1$ characters (recall that we assume a fixed-length code, so that the characters and meta-characters are encoded by the same number of bits).

For strings x and y like above, define the super-string, denoted $\overline{xy}$, as the (shortest) concatenation of x with y, but without repeating the overlapping part, i.e., $\overline{xy} = x_1 \cdots x_k y_{t+1} \cdots y_\ell$, where t has been chosen as largest among all possible t's. For example, if x is `element` and y is `mention`, then $\overline{xy}$ is `elemention`. The weight of the directed edge from x to y is defined as

$$w(x, y) = freq(\overline{xy})(|y| - 1).$$

The reason for this definition is that if the text will be parsed by a greedy method, it may happen that $freq(\overline{xy})$ of the $freq(y)$ occurrences of y will not be detected, because for these occurrences, the first few characters of y will be parsed as part of x.

In fact, assuming that the parsing of $\overline{xy}$ will always start with x in a greedy method is an approximation. For it may happen that certain occurrences of x will stay undetected because of another preceding string z, that has a non-empty suffix overlapping with a prefix of x. To continue the example, suppose z is `steel`, then $\overline{z\overline{xy}}$ is `steelemention`, which could be parsed as z`e`y. Moreover, when the overlapping part itself has a prefix which is also a suffix, then choosing the *shortest* concatenation in the definition of $\overline{xy}$ does not always correspond to the only possible sequence of characters in the text. For example, if x is `managing` and y is `ginger`, $\overline{xy}$ would be `managinger`, but the text could include a string like `managinginger`. We shall however ignore such cases and keep the above definition of $w(x, y)$.

A first simplification results from the fact that we seek the subgraph induced by the set of vertices V', so that either both directed edges (x, y) and (y, x) will be included, if they both exist, or none of them. We can therefore consider an equivalent undirected graph, defining the label on the (undirected) edge (x, y) as

$$w(x, y) \;=\; freq(\overline{xy})(|y| - 1) \;+\; freq(\overline{yx})(|x| - 1).$$

For a text of n characters, the resulting graph has $\Theta(n^2)$ vertices, and may thus have $\Theta(n^4)$ edges, which is prohibitive, even for small n. We shall thus try

to exclude a priori strings that will probably not be chosen as meta-characters. The excluded strings are:

1. a string of length 1 (they are included anyway in $\mathcal{A}$);
2. a string that appears only once in the text (no savings can result from these).

For example, consider the text

`the-car-on-the-left-hit-the-car-I-left-on-the-road`.

Using the above criteria reduces the set of potential strings to: {`the-car-`, `-on-the-`, `-left-`}, and all their substrings of length > 1. If this seems still too large, we might wish to exclude also

3. a string x that always appears as a substring of the same string y.

This would then purge the proper substrings from the above set, except the string `the-`, which appears as substring of *different* strings. The rationale for this last criterion is that it is generally preferable to include the longer string y into the extended alphabet. This is, however, not always true, because a longer string has potentially more overlaps with other strings, which might result in an overall loss.

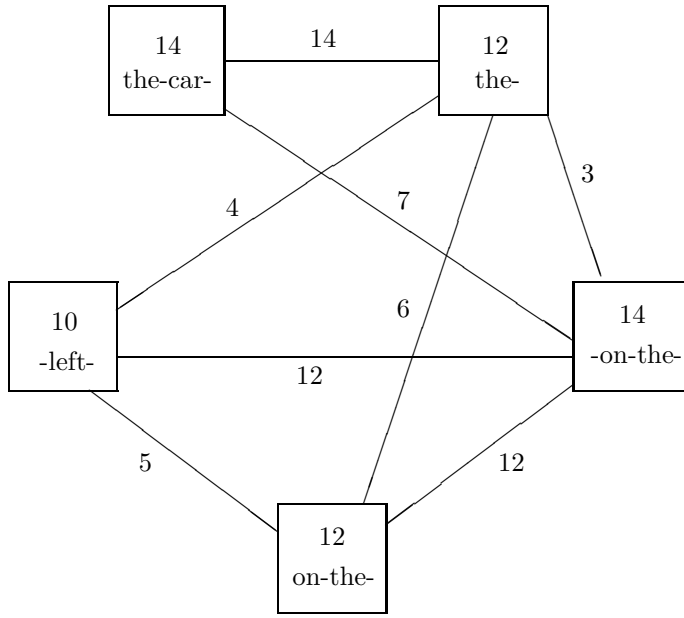


FIGURE 1: *Graph for text*
`the-car-on-the-left-hit-the-car-i-left-on-the-road`

Figure 1 depicts the graph of the above example, but to which we have added `on-the-` as fifth string to the alphabet (in spite of it appearing always as substring of `-on-the-`) because it will yield below an example showing a deficiency of criterion 3. Edges with weight 0 have been omitted, e.g., there is no edge between `-left-` and `the-car-`, because the super-string `the-car-left-` does not appear; similarly, there is no self-loop on the vertex $x =$`-left-`: even though the super-string $\overline{xx} =$`-left-left-` is defined, its frequency is zero. If we are looking for a set of 3 meta-characters, the best of the $\binom{5}{3} = 10$ triplets is `the-car-`, `-left-`, `on-the-`, yielding savings of $14 + 10 + 12 - 5 = 31$ characters. Indeed, replacing all occurrences of the meta-characters in the text reduces the number of elements in the parsing from 50 to 19, of which 14 are single characters and 5 are meta-characters. However, one can see here that the criterion of excluding a string x if it always appears as substring of y, does not assure optimality: if we would choose the string `on-the` instead of `on-the-` as fifth vertex, the text could be parsed as 17 instead of 19 elements.

There is, however, still a problem with the complexity of the algorithm. A similar problem has been shown to be NP-complete in [16], but we bring here a direct proof:

Theorem 1. *The problem of finding a subgraph maximizing (1) is NP-complete.*

Proof: One actually has to deal with the corresponding decision problem, which we denote SDP (Substring Dictionary Problem). Given is a graph $G = (V, E)$, with weights on both vertices and edges, and 3 non-negative constants K_1, K_2 and K_3. Is there a subset of vertices $V' \subseteq V$, such that $|V'| \leq K_1$, $\sum_{x \in V'} w(x) \geq K_2$ and $\sum_{x,y \in V'} w(x, y) \leq K_3$?

Once a guessing module finds the set V', the other conditions are easily checked in polynomial time, so SDP $\in$ NP. To show that SDP is also NP-hard, the reduction is from Independent Set (IS) [10], defined by: given a graph $G_1 = (V_1, E_1)$ and an integer L, does G_1 contain an independent set of size at least L, that is, does there exist a subset $V_1' \subseteq V_1$, such that $|V_1'| \geq L$, and such that if x and y are both in V_1', then $(x, y) \notin E_1$?

Given a general instance of IS, define the following instance of SDP: let $G = G_1$, define $w(x) = 1$ for all vertices $x \in V$, $w(x, y) = 1$ for all edges $(x, y) \in E$, $K_1 = |V|$, $K_2 = L$ and $K_3 = 0$. Suppose that there is an independent set V_1' of size at least L in G_1. We claim that the same set also satisfies the constraints for SDP. Indeed, $|V_1'| = L \leq K_1$, $\sum_{x \in V_1'} w(x) = |V_1'| = L \geq K_2$, and since in an independent set, there are no edges between the vertices, $\sum_{x,y \in V_1'} w(x, y) = 0 \leq K_3$.

Conversely, suppose there is a set $V' \subseteq V$ which fulfills the conditions of SDP in the graph G. Then because the weight of each vertex is 1, it follows from the second condition that there are at least $K_2 = L$ vertices in V'. The choice of K_3 as 0 in the third condition, together with the fact that the weight of each edge is 1, implies that no two vertices of V' may be connected by an edge, that is, V' is an independent set. ∎

Two problems have therefore to be dealt with: first, we need a fast procedure for the construction of the graph, and second we seek a reasonable heuristic, running fast enough to yield a practical algorithm, and still making better choices than discarding any strings that overlap with one previously chosen.

3 Implementation Details

3.1 Graph Construction

Since we are looking for a special set of substrings of the given input text T, a *position-tree* or *suffix-tree* may be the data structure of choice for our application [2,14]. The strings occurring at least twice correspond to the internal nodes of the tree. As to condition 3. above, if a string x occurs always as a prefix of y, the node corresponding to x has only one son in the position tree. Seeking the longest re-occurring strings, these are found, for each branch of the tree, at the lowest level of the internal nodes, that is, for each path from the root to a leaf, the node just preceding the leaf corresponds to one of the strings we consider as a vertex in our graph. The number of vertices in the graph, m, is thus clearly bounded by the number of leaves in the position tree, which is $n + 1$, one leave for each position in the string $T\$$, where \$ is a special end-marker not belonging to our basic alphabet Σ. Figure 2 shows a part of the (compacted) position tree for our example, where the black vertices correspond to the four strings left after applying conditions 1.–3. above.

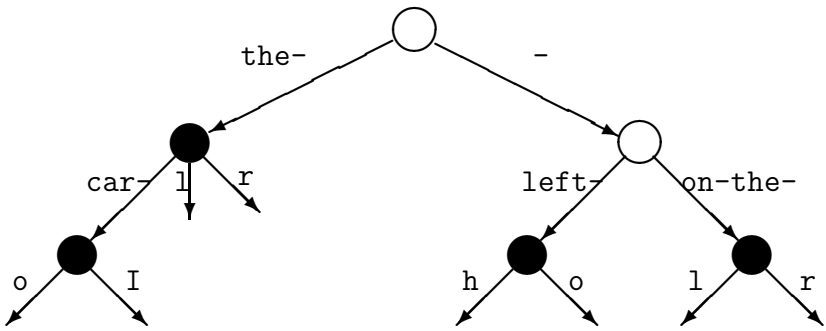


FIGURE 2: *Part of the position tree*

Compacted position trees can be constructed in time linear in the length of the input text T. Also in linear time one could add to each vertex in the tree, the number of times the corresponding string occurs in T [1]. Thus, the set of vertices V and their labels can be constructed in time $O(n)$. We shall refer below to the nodes of the position tree corresponding to the vertices of V as *black* vertices.

As to the edges, it suffices to check for each string x corresponding to a black vertex v_x whether there should be a directed edge from it to a black vertex corresponding to a string y. This will be the case if and only if the super-string $\overline{xy}$ exists. We have thus to search the position tree for each of the $|x|-1$ proper suffixes of x. If the path in the position tree corresponding to one of these proper suffixes leads to an internal node t, there will be an edge from v_x to each of the black vertices in the subtree rooted by t.

If there is an upper limit K, independent of the size n of the text, on the length of a string corresponding to a black vertex, the number of edges in the graph G will be linear. Indeed, a proper suffix of length i of x can be extended to at most $|\Sigma|^{K-i}$ strings y such that $\overline{xy}$ exists, therefore the outdegree of x is bounded by $|\Sigma|^K$, which is a constant relative to n. Assuming the existence of such an upper bound K is not unrealistic for real-life applications, where most re-occurring strings will tend to be words or short phrases. Even if some long strings re-occur, e.g., runs of zeros or blanks, it will often be the case that a bound K exists for the lengths of all except a few strings, which will still yield a linear number of edges. The following Theorem shows that such a linear bound on the number of edges can not always be found.

Theorem 2. *There are input texts T for which the corresponding graph $G = (V, E)$ satisfies $|E| = \Theta(|V|^2)$.*

Proof: Recall that a DeBruijn sequence of order k is a binary string B_k of length 2^k, such that each binary string of length k occurs exactly once in B_k, when B_k is considered as a circular string (see [7, Section 1.4]). For example, B_3 could be 00111010.

Consider the text $T = CB_k$, where C is the suffix of length $k-1$ of B_k. Each of the 2^k possible binary strings of length k appears exactly once in CB_k, so no string of length longer than k appears more than once. Every binary sequence of length $k-1$ appears exactly twice in a DeBruijn sequence, and as sub-strings of different strings of length k, thus there is a vertex in G for each of the 2^{k-1} strings of length $k-1$. More generally, for $2 \leq i < k$ (recall that strings of length 1 do not generate vertices), there are 2^i strings of length i that occur more than once in T, and each of these strings of length $< k$ is a substring of more than one string of length k. The number of vertices in the graph is therefore $\sum_{i=2}^{k-1} 2^i = 2^k - 4$. We shall refer below to the corresponding strings as *vertex-strings*.

Consider one of the vertex-strings x, and denote its rightmost bit by b. Then there must be edges from the corresponding vertex v_x to at least all the vertices corresponding to strings starting with b. There are 2^{i-1} strings of length i starting with b, $2 \leq i < k$. Thus the number of edges emanating from v_x is at least $\sum_{i=2}^{k-1} 2^{i-1} = 2^{k-1} - 2$. Therefore the total number of edges in the graph is at least $(2^k - 4)(2^{k-1} - 2)$, which is quadratic in the number of vertices. ∎

3.2 Heuristics for Finding a Good Sub-graph

The optimization problem has been shown above to be NP-complete, so there is probably no algorithm to find an optimal solution in reasonable time. A family of simple greedy heuristics that have previously been used (see, e.g., [6]) includes the following steps:

1. Decide on a set S of potential strings and
   calculate their weights $w(x)$ for $x \in S$
2. Sort the set S by decreasing values of $w(x)$
3. Build the set $\mathcal{A}$ of the strings forming the meta-alphabet by
   3.1 start with $\mathcal{A}$ empty
   3.2 repeat until $|\mathcal{A}|$ is large enough or S is exhausted
      3.2.1 $x \longleftarrow$ next string of sorted sequence S
      3.2.2 if x does not overlap with any string in $\mathcal{A}$, add it to $\mathcal{A}$

The set S could be chosen as the set of all sub-strings of the text of length up to some fixed constant k, or it might be generated iteratively, e.g., starting with character pairs, purging the overlaps, extending the remaining pairs to triplets, purging overlaps again, etc. Such a strategy of not allowing any overlaps corresponds in our approach above to choosing an independent set of vertices.

To extend this greedy heuristic to include also overlapping strings, we have to update the weights constantly. It is therefore not possible to sort the strings by weight beforehand, and the data structure to be used is a *heap*. This will give us at any stage access in time $O(1)$ to the element x with largest weight $W(x)$, which should represent the expected additional savings we incur by adding x to the set $\mathcal{A}$, the currently defined meta-alphabet. $W(x)$ is therefore $w(x)$ if none of the neighbors of x belongs to $\mathcal{A}$, and it is more generally $w(x) - \sum_{y \in \mathcal{A}} w(x, y)$. The proposed heuristic is thus as follows:

1. Define the set S as substrings of the text complying with constraints 1.–3.
2. for each $x \in S$ define $W(x) \longleftarrow w(x)$
3. build heap of elements $W(x)$, with root pointing to maximal element
4. Build the set $\mathcal{A}$ of the strings forming the meta-alphabet by
   4.1 start with $\mathcal{A}$ empty
   4.2 repeat until $|\mathcal{A}|$ is large enough or heap is empty
      4.2.1 $x \longleftarrow$ string with weight at root of heap
      4.2.2 add x to $\mathcal{A}$ and remove $W(x)$ from heap
      4.2.3 repeat for all neighbors y of x in graph
         4.2.3.1 $W(y) \longleftarrow W(y) - w(x, y)$
         4.2.3.2 if $W(y) \leq 0$ remove $W(y)$ from heap
         4.2.3.3 else relocate $W(y)$ in heap

Note that $W(x)$ may indeed become negative, as can be seen in Figure 1. This would mean that the potential gain obtained from including x in $\mathcal{A}$ might be canceled because of the overlaps.

The complexity of the heuristic can be evaluated as follows: step 1. can be done in time linear in the size n of the text using a position tree as explained above. Let $m = |S|$ be the number of vertices. Steps 2. and 3. are $O(m)$. Step 4.2.1 is $O(1)$ and step 4.2.2 is $O(\log m)$. Any edge (x, y) of the graph is inspected at most once in the loop of 4.2.3, and for each such edge, a value $W(y)$ is either removed or relocated in the heap in time $O(\log m)$. Thus the total time of step 4. is $O(|E| \log m)$. As we saw in Theorem 2, $|E|$ could be quadratic in m. But when $W(y)$ is updated, it may only decrease, so that its relocation in the heap can only be to a lower level. The total number of updates for any $W(y)$ is therefore bounded by the depth of the heap, which implies that the complexity of step 4. is only $O(m \log m)$.

Though such greedy heuristics seem to give good results for the type of problem we consider here [11], we shall try in another line of investigation to adapt other approximation schemes to our case. The Independent Set optimization problem has been extensively studied, and some good heuristics exist (see [12]). In particular, we can use the fact that our graph has a low average vertex degree, and is of bounded vertex degree if we put some constraints on the lengths of the strings we consider. Another problem similar to ours, with edge weights but without weights on the vertices, has been approximated in [15].

3.3 Variable Length Encodings

In our above description, we have made various simplifying assumptions. In a more precise analysis, we shall try in a second stage to adapt the general strategy to more complex — and more realistic — settings.

When defining the graph, we assumed that the elements of the meta-alphabet are encoded by a fixed length code. Such a code will, however, be optimal only if the occurrence distribution of the elements is close to uniform. Otherwise, variable-length codes such as Huffman or arithmetic codes should be used. The problem is then one of the correct definition of the graph weights, but the generalization of the above method is not straightforward. We look for a set of strings which are selected on the basis of the lengths of their encodings; the lengths depend on their probabilities; these in turn are a function of the full set of strings, which is the set we wish to define.

A possible solution to this chicken and egg problem is as follows. We first estimate the average length, $\hat{\ell}$, of the strings $s_1, s_2, \ldots$ that will ultimately be selected. This can be done by some rough rule of thumb or by applying the heuristic iteratively. Clearly, if T is the length of the text and N is the number of the selected strings, we have

$$T = \sum_{i=1}^{N} |s_i| freq(s_i).$$

Replacing now all $|s_i|$ by their estimated value $\hat{\ell}$, we get

$$T = \hat{\ell} \sum_{i=1}^{N} freq(s_i),$$

from which an estimate for the total frequency, $W = \sum_{i=1}^{N} freq(s_i)$, of the selected elements can be derived as

$$W = \frac{T}{\hat{\ell}}.$$

Hence, when defining the probabilities in the weights of the vertices and edges, we shall use as approximation the frequency of a string divided by W, even though this is not a real probability distribution, as the selected values will not necessarily add up to 1. But the estimation bias is alleviated by the fact that the probabilities are only needed to determine the lengths of the codewords. We shall use $-\log_2 p$ as approximation for the length of a codeword that appears with probability p. This is exact for arithmetic coding, and generally close for Huffman coding.

The new weights are not measured in number of characters, but in number of bits. For a string $x = x_1 \cdots x_r$, the weight of the corresponding vertex will be

$$w(x) = freq(x) \left(-\sum_{i=1}^{r} \log_2 \frac{freq(x_i)}{W} + \log_2 \frac{freq(x)}{W} \right),$$

where $freq(x_i)$ are the frequencies of the individual characters making up the string, and similarly for the weights $w(x, y)$ of the edges. Once the set of meta-characters $\mathcal{A}$ is determined by the heuristic, we can update our estimate for the average length $\hat{\ell}$, and repeat the process iteratively. Even without a convergence guarantee, this will be useful for the elimination of bad strings.

4 Experimental Results

Three texts of varying lengths and different languages were chosen for the tests: the King James version of the *Bible* in English, a French text by Voltaire called *Dictionaire philosophique* and a lisp program *progl* from the Calgary corpus [3]. Table 1 lists the full and compressed sizes of these files in bytes.

We first considered the weights in the graph corresponding to fixed length encodings and generated the set of meta-characters according to the second heuristic above. The weights were then changed to reflect the expected savings with variable length encodings, and the string generation process was repeated. For comparison, we also produced, for each file, a list of the most frequent words, as well as a list of the most frequent character bigrams. Each of these in turn were used to define a new set of meta-characters. The sizes of the compressed files using a fixed length encoding (9 bits per meta-character) are left-justified in the corresponding boxes of Table 1. The right-justified numbers in these boxes correspond to the sizes of the compressed files if Huffman coding is applied according to the occurrence frequencies of the meta-characters. For the variable length encodings, these results were obtained after 2–3 iterations. Throughout, the parsing was done greedily, seeking repeatedly the longest possible match.

As can be seen, among the alternatives tested, the meta-characters produced by the heuristic on the graph with the weights corresponding to fixed length

	Bible	*Voltaire*	*progl*
Full size	3483553	554719	71646
Best substrings for fixed	2066427	307967	32136
Huffman	1552112	238065	22437
Best substrings for variable	2307194	335568	35548
Huffman	1546552	236531	22273
Most frequent words	2503537	414877	50204
Huffman	1569385	256701	28943
Most frequent pairs	2118704	322829	43939
Huffman	1812016	270994	37420

TABLE 1: *Compression results*

encoding indeed achieve the best compression by fixed length encodings, and the meta-characters produced with the weights of the variable length encoding are best when Huffman coding is applied. It is noteworthy that when passing from the fixed length to the variable length weights, the number of meta-characters in the parsing increases, and nevertheless, the size of the corresponding Huffman encoded file is smaller. This is due to the fact that the distribution of frequencies in the latter case is much skewer than in the former, resulting in the overall gain.

5 Concluding Remarks

Alphabet extension is not new to data compression. However, the decision about the inclusion into the alphabet of various strings, such as frequent words, phrases or word fragments, has often been guided by some *ad-hoc* heuristic. The present work aims at making this decision in a systematic, theoretically justifiable way. There are still many more details to be taken care of, in particular, devising more precise rules wherever approximations have been used. Future work will also consider other models, e.g., basing the encoding process on a first-order Markov model, as in [4,5], which can considerably improve compression performance. The figures in Table 1 are still far from the compression that can be achieved with adaptive dictionary methods. For instance, the Bible file can be reduced by `compress` to 1.203 MB and by `gzip` to 1.022 MB. But recall that we are looking for a static method, which will allow random and not only sequential access to the compressed file, so that the dynamic methods are ruled out. On the other hand, using Huffman coding in combination with Markov modeling, when applied to the extended alphabet obtained by the fixed length weights,

compresses the Bible file to 1.033 MB (including the overhead of storing the large model), which is almost as good as for `gzip`. We thus anticipate that by adapting the graph weights to the Markov model, even better compression can be achieved.

References

1. Apostolico A., The myriad virtues of subword trees, *Combinatorial Algorithms on Words,* NATO ASI Series Vol **F12**, Springer Verlag, Berlin (1985) 85–96.
2. Aho A.V., Hopcroft J.E., Ullman J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA (1974).
3. Bell T.C., Cleary J.G., Witten I.A., *Text Compression*, Prentice Hall, Englewood Cliffs, NJ (1990).
4. Bookstein A., Klein S.T., Compression, Information Theory and Grammars: A Unified Approach, *ACM Trans. on Information Systems* **8** (1990) 27–49.
5. Bookstein A., Klein S.T., Raita T., An overhead reduction technique for megastate compression schemes, *Information Processing & Management* **33** (1997) 745–760.
6. Bookstein A., Klein S.T., Ziff D.A., A systematic approach to compressing a full text retrieval system, *Information Processing & Management* **28** (1992) 795–806.
7. Even S., *Graph Algorithms*, Computer Science Press (1979).
8. Fraenkel A.S., All about the Responsa Retrieval Project you always wanted to know but were afraid to ask, Expanded Summary, *Jurimetrics J.* **16** (1976) 149–156.
9. Fraenkel A.S., Mor M., Perl Y., Is text compression by prefixes and suffixes practical? *Acta Informatica* **20** (1983) 371–389.
10. Garey M.R., Johnson D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco (1979).
11. Halldorsson M.M., Radhakrishnan J., Greed is good: approximating independent sets in sparse and bounded degree graphs, *Proc. 26th ACM-STOC* (1994) 439–448.
12. Hochbaum D.S., *Approximation Algorithms for NP-Hard Problems*, PWS Publishing Company, Boston (1997).
13. Klein S.T., Space and time-efficient decoding with canonical Huffman trees, *Proc. 8th Symp. on Combinatorial Pattern Matching,* Aarhus, Denmark, *Lecture Notes in Computer Science* **1264**, Springer Verlag, Berlin (1997) 65–75.
14. McCreight E.M., A space economical suffix tree construction algorithm, *Journal of the ACM* **23** (1976) 262–272.
15. Kortsarz G., Peleg D., On choosing dense subgraphs, *Proc. 34th FOCS*, Palo-Alto, CA (1993) 692–701.
16. Storer J.A., Szymanski, T.G., Data compression via textual substitution, *J. ACM* **29** (1982) 928–951.
17. Witten I.H., Moffat A., Bell T.C., *Managing Gigabytes: Compressing and Indexing Documents and Images,* Van Nostrand Reinhold, New York (1994).

Genome Rearrangement by Reversals and Insertions/Deletions of Contiguous Segments

Nadia El-Mabrouk

Département d'informatique et de recherche opérationnelle, Université de Montréal
CP 6128 succursale Centre-Ville, Montréal, Québec, H3C 3J7
mabrouk@iro.umontreal.ca

Abstract. Analysis of genome rearrangements allows to compare molecular data from species that diverged a very long time ago. Results and complexities are tightly related to the type of data and genome-level mutations considered. For sorted and signed data, Hannenhalli and Pevzner (HP) developed the first polynomial algorithm in the field. This algorithm solves the problem of sorting by reversals. In this paper, we show how to extend the HP approach to include insertions and deletions of gene segments, allowing to compare genomes containing different genes. We provide an exact algorithm for the asymmetric case, as applies in organellar genome evolution, and a heuristic for the symmetric case, with bounds and a diagnostic for determining whether the output is optimal.

1 Introduction

The polynomial algorithm of Hannenhalli and Pevzner (hereafter "HP") for sorting by reversals [HP95a] was a breakthough for the formal analysis of evolutionary genome rearrangement. Moreover, they were able to extend their approach to include the analysis of interchromosomal translocations [HP95b,Han95]. The applicability of this theory, however, has been limited by the difficulty in extending it to other rearrangement mechanisms, such as gene duplication, or the transposition, duplication or insertion of genomic fragments. In this paper, I show how to extend the HP approach to include insertions and deletions of gene segments, providing an exact algorithm for the asymmetric case, as applies in organellar genome evolution, and a heuristic for the symmetric case, with bounds and a diagnostic for determining whether the output is optimal.

This work will be situated in the context of circular genomes, as is most frequently the case for single chromosomal genomes such as organelles and prokaryotes. We assume that the direction of transcription is known. Formally, given a set $\mathcal{A}$ of genes, a genome is a circular string of signed (+ or −) elements of $\mathcal{A}$, no gene appearing more than once. A **gene segment** is a subsequence of consecutive genes in a genome. For a string $X = x_1 x_2 \cdots x_r$, denote by $-X$ the **reverse** string $-x_r - x_{r-1} \cdots - x_1$. A **reversal** (or **inversion**) transforms some proper substring of a genome into its reverse.

Let G and H be two genomes with some genes in common, and others specific to each genome. Write $\mathcal{A}$ for the set of genes in both G and H, and $\mathcal{A}_G$ and $\mathcal{A}_H$ for those in G and H only respectively. Our problem is to find the minimum number

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 222–234, 2000.

of reversals, insertions and deletions of gene segments necessary to transform G to H, with the restriction that deletions and insertions are not applied on genes present in both genomes G and H. Although such manipulations may be meaningful in some circonstances, in this paper we permit insertions/deletions of genes only in $\mathcal{A}_G$ or $\mathcal{A}_H$.

2 The Hannenhalli–Pevzner Theory

Given a set $\mathcal{A}$ of genes and two circular genomes G and H such that each gene of $\mathcal{A}$ appears exactly once in each genome, how can we find the minimum number of reversals necessary to transform G into H? The HP solution [HP95a] is an exact polynomial algorithm, based on a bicoloured cycle graph constructed from G and H. This graph will be the focus of our extension to the problem to include segment insertion and deletion. We first sketch the HP method.

The HP bicoloured **cycle graph** $\mathcal{G}$ of G with respect to H is as follows. If gene x_i in genome $G = x_1 \cdots x_n$ has positive sign, replace it by the pair $x_i^t x_i^h$, and if it is negative, by $x_i^h x_i^t$. Then the vertices of $\mathcal{G}$ are just the x^t and the x^h for all x in $\mathcal{A}$. Without confusion, we can also apply the terminology "vertices of G" to mean elements like x^t or x^h. Any two vertices which are adjacent in G, other than x_i^t and x_i^h from the same x, are connected by a black edge, and any two adjacent in H, by a gray edge. Each vertex is incident to exactly one black and one gray edge, so that there is a unique decomposition of $\mathcal{G}$ into c disjoint cycles of alternating edge colours. By the **size of a cycle** we mean the number of black edges it contains. Note that c is maximized when $G = H$, in which case each cycle is of size 1 and $c = |\mathcal{A}|$.

A reversal is determined by the two points where it "cuts" the current genome, which correspond each to a black edge. We say that a **reversal ρ is determined by the two black edges** $e = (a, b)$ and $f = (c, d)$ if ρ reverses the segment of G bounded by vertices b and c. The **reversal determined by a gray edge** g is the reversal determined by the two black edges adjacent to g. A reversal may change the number of cycles, so that minimizing the number of operations can be seen in terms of increasing the number of cycles as fast as possible.

Let e and f be two black edges of a cycle C of length greater than 1. Think of both e and f as being oriented clockwise in the graph. Each one of them induces an orientation of C. If the orientation induced by e and f coincide, we say that they are **convergent**; otherwise we say that they are **divergent**. Let ρ be a reversal determined by the two black edges e and f, and $\Delta(c)$ the increase in the number of cycles of the graph after the reversal ρ.

1. If e and f belong to different cycles, then $\Delta(c) = -1$.
2. If e and f belong to the same cycle and converge, then $\Delta(c) = 0$.
3. If e and f belong to the same cycle and diverge, then $\Delta(c) = 1$.

The goal is to perform as many reversals of the third type as possible. A key concept in the method is the **unoriented components**. A gray edge g in

a cycle of size > 1 is **oriented** if the two black edges adjacent to g diverge. Otherwise, the edge is **unoriented**. A cycle C is **oriented** if it contains at least one oriented gray edge. Otherwise it is **unoriented**. We say that two gray edges g_1 and g_2 **cross** if g_1 links vertices a and c, g_2 links vertices b and d, but these vertices are ordered a, b, c, d in G. Two cycles containing gray edges that cross are **connected**. A (connected) **component** of $\mathcal{G}$ is a subset of the cycles, built recursively from one cycle, at each step adding all the remaining cycles connected to any of those already in the construction. A **good component** has at least one oriented cycle and thus at least one oriented gray edge. Otherwise it is a **bad component**.

HP showed that an oriented component can be **solved**, i.e transformed to a set of cycles of size 1, by a series of **good reversals**, that is reversals of the third type. However bad components often require **bad reversals** (reversals of the first and second type). The set of bad components is subdivided into subsets, depending on the difficulty to solve them (i.e. transform them into good components). This subdivision is explained below.

We say that component U **separates** two components U' and U'' if any potential edge we tried to draw from a vertex of U' to one of U'' would cut a gray edge of U. A **hurdle** is a bad component which does not separate any pair of bad components. We further distinguish between two types of hurdles. A hurdle $\mathcal{H}$ is a **superhurdle** if the elimination of $\mathcal{H}$ from $\mathcal{G}$ transforms a non-hurdle $\mathcal{H}'$ into a hurdle, and is a **simple hurdle** if not. A cycle graph $\mathcal{G}$ is a **fortress** if it contains an odd number of hurdles, and they are all superhurdles.

The underlying idea is that a bad component U that separates two bad components U' and U'' is automatically solved by solving U' and U'', and thus, can be considered as a good one. On the other hand, a hurdle requires bad reversals to be solved, and a superhurdle is a "bad hurdle", giving rise to additional problems.

To underestand how hurdles are solved, we require the following remark, proved by HP:

Remark 1:

- A reversal determined by two divergent black edges in a single unoriented cycle C transforms C into an oriented cycle C'. The total number of cycles of the graph remains unchanged.
- A reversal determined by two black edges from two different unoriented cycles C_1 and C_2 transforms C_1 and C_2 into a single oriented cycle C. The total number of cycles in the graph decreases by 1.

In order to solve hurdles, HP perform **merging** of appropriate pairs of hurdles. More precisely, components U' and U'' are solved by performing a particular reversal acting on one edge of an unoriented cycle of U' and one edge of an unoriented cycle of U'', which transforms U' and U'' into one good component. In the case of an odd number of hurdles, some hurdles are **cut**, by performing

a reversal of the first type on two edges of the same cycle of a hurdle, which transforms that hurdle into a good component. These operations are more precisely described in the HP algorithm (Figure 1). We require one more notion. Two hurdles $\mathcal{H}_1$ and $\mathcal{H}_2$ are **consecutive** in $\mathcal{G}$ if there is no hurdle $\mathcal{H}$ in $\mathcal{G}$ that has all its vertices between any vertex of $\mathcal{H}_1$ and any vertex of $\mathcal{H}_2$ (clockwise or counterclockwise).

HP showed that the minimum number of reversals necessary to transform G into H is:

$$R(G,H) = b(G,H) - c(G,H) + h(G,H) + fr(G,H) \quad (\mathbf{1})$$

where $b(G,H)$ is the number of black edges, $c(G,H)$ is the number of cycles (of size ≥ 1), $h(G,H)$ is the number of hurdles of $\mathcal{G}$, and $fr(G,H)$ is 1 if $\mathcal{G}$ is a fortress and 0 otherwise.

The HP algorithm for transforming G into H with a minimum of reversals is described below. Note that $h(G,H)$ changes value during the execution of the algorithm. In [HP95a], it is shown that at each step, a reversal satisfying the conditions of the algorithm can be found. For a more detailed presentation of the HP formula and algorithm, see also [SM-97].

We extend the HP method to take into account deletions in Section 3, insertions in Section 4, and deletions plus insertions in Section 5. For reasons of space, we omit proofs in this abstract.

3 Deletions

Returning to the problem at hand, we require the minimum number of **rearrangement operations**, i.e., reversals, insertions and deletions, necessary to transform genome G into genome H. We first consider the case where there are no insertions. All genes in H are also in G, i.e. $\mathcal{A}_H = \emptyset$. The problem is then to transform G into H with a minimum of reversals and segment deletions. Given that the genes of $\mathcal{A}_G$ are destined to be deleted, their identities and signs are irrelevant (a segment does not need to have all its genes with the same orientation to be deleted), and could be replaced with any symbols different from those used for the genes in $\mathcal{A}$. For any subsequence of vertices of G of form $S = u_1 u_2 \cdots u_{p-1} u_p$, where u_1, u_p correspond to two genes of $\mathcal{A}$, and for all i, $2 \leq i \leq p-1$, u_i corresponds to a gene of $\mathcal{A}_G$, we replace S by the subsequence $S' = u_1 \delta(u_1) u_p$.

Example 1. *Consider sets* $\mathcal{A} = \{a,b,c,d,e,f\}$, $\mathcal{A}_G = \{g,h,i,j,k,l\}$, *and the circular genome* G *(i.e. the first term follows the last one):* $+g \ +a \ -h \ +b \ +c \ +i \ +d \ +l \ +e \ -j \ +k \ +f$. *Then* G *can be rewritten:* $\delta(f^h) \ +a \ \delta(a^h) \ +b \ +c \ \delta(c^h) \ +d \ \delta(d^h) \ +e \ \delta(e^h) \ +f$.

We represent genomes G and H by the bicoloured cycle graph $\mathcal{G}(G,H)$. Denote by $\mathbf{V}$ the set of vertices, B the set of black edges and D the set of gray edges of $\mathcal{G}(G,H)$. These three sets are defined as follows :

Algorithm HP:
While $\mathcal{G}$ contains a cycle of size > 1
do
  If $\mathcal{G}$ contains a good component $\mathcal{CC}$
    Choose an oriented gray edge g of $\mathcal{CC}$ such that the reversal determined by g does not create a new bad component;
    Execute the reversal determined by g;
  Otherwise
    If $h(G,H)$ is even {hurdle merging}
      Consider two appropriate (not consecutive) hurdles $\mathcal{H}_1$ and $\mathcal{H}_2$;
      Arbitrarily choose a black edge e in $\mathcal{H}_1$ and a black edge f in $\mathcal{H}_2$, and execute the reversal determined by e and f;
    Otherwise
      If $h(G,H)$ is odd and there is a simple hurdle $\mathcal{H}$ {hurdle cutting}
        Execute the reversal determined by two arbitrarily chosen edges e and f belonging to a single cycle in $\mathcal{H}$;
      Otherwise {fortress, hurdle merging}
        If $\mathcal{G}$ is a fortress with more than three superhurdles
          Consider two not consecutive hurdles $\mathcal{H}_1$ and $\mathcal{H}_2$;
          Arbitrarily choose a black edge e in $\mathcal{H}_1$ and a black edge, f in $\mathcal{H}_2$ and execute the corresponding reversal;
        Otherwise {3-fortress, hurdle merging}
          Consider any two hurdles $\mathcal{H}_1$ and $\mathcal{H}_2$;
          Arbitrarily choose a back edge e in $\mathcal{H}_1$ and a black edge f in $\mathcal{H}_2$, and execute the corresponding reversal;

Fig. 1. The HP algorithm for sorting genomes by reversals.

- $\mathbf{V} = \{x^s\}_{x\in\mathcal{A}}^{s\in\{h,t\}}$.
- The black edges pertain to genome G. There are two sorts of black edges: **direct** black edges link two adjacent vertices in G, and **indirect** black edges link two vertices separated by a δ. For an indirect black edge $e = (a, b)$, $\delta(a)$ is the **label of e**.
- Gray edges link adjacent vertices in H.

An **indirect cycle** is one containing at least one indirect edge. A hurdle containing at least one indirect cycle is an **indirect hurdle**, otherwise it is a **direct hurdle**.

Example 2. *Consider the genome G of Example 1 and the genome $H = +a\ +d\ +c\ +b\ +f\ +e$. The graph $\mathcal{G}(G,H)$ corresponding to these two genomes is depicted in Figure 2. This graph is made up of two cycles, C_1 and C_2, both indirect. These two cycles are unoriented and form a hurdle.*

The **reduced genome** $\overline{G}$ induced by G is obtained by deleting from G all the genes in $\mathcal{A}_G$. The cycle graph $\mathcal{G}(\overline{G},H)$ associated with genomes $\overline{G}$ and H is obtained from $\mathcal{G}(G,H)$ by considering all the black edges of B to be direct.

For a reduced genome, the reversal acting on the two black edges $e = (a, b)$ and $f = (c, d)$ reverses the segment of the genome $\overline{G}$ bounded by vertices b and c. For a genome G containing genes to be deleted, the indirect edges represent, not an adjacency in the genome G, but rather an interval containing only genes

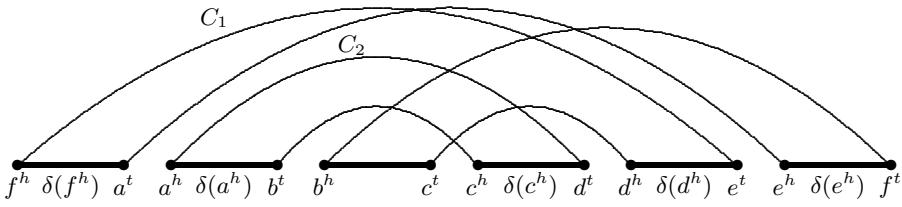


Fig. 2. The black edges with a δ are indirect, i.e., there are genes of $\mathcal{A}_G$ between the two vertices that determine each of these edges.

to be deleted. We thus have to define what we mean by "the reversal acting on two black edges", of which at least one is indirect.

Let $e = (a, b)$ and $f = (c, d)$ be two indirect edges. The interval $[b, c]$ designates the interval of G bounded by the two vertices b and c; the interval $[b, \delta(c)]$ designates the interval of G bounded on the left by b and on the right by the element of $\mathcal{A}_G$ adjacent to d.

Definition 1. *Consider two black edges $e = (a, b)$ and $f = (c, d)$. The* **reversal ρ determined by e and f** *is defined as follows. If e and f are not both indirect, ρ reverses segment $[b, c]$. Else (e and f are both indirect), ρ reverses segment $[b, \delta(c)]$ (see Figure 3.a).*

Consider an oriented gray edge $g = (b, d)$ adjacent to two black edges $e = (a, b)$ and $f = (c, d)$. The **reversal ρ determined by** *g reverses segment $[b, c]$ if f is direct, and reverses segment $[b, \delta(c)]$ if f is indirect (see Figure 3.b).*

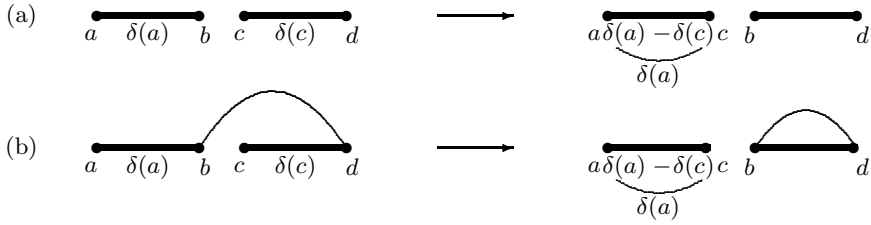


Fig. 3. (a) A reversal determined by two indirect black edges. (b) A reversal determined by a gray edge adjacent to 2 indirect black edges. This creates a direct cycle of size 1.

The HP algorithm can be generalized to graph containing direct and indirect edges by making use of the above definition of a reversal determined by two black edges, or by a gray edge. After each reversal, for each vertex a, $\delta(a)$ represents the sequence of genes of $\mathcal{A}_G$ following a in the resulting genome. These gene sequences change throughout the execution of the algorithm.

Lemma 1. *A reversal determined by an oriented gray edge of a cycle C transforms C into two cycles C_1 and C_2, at least one of which is of size 1. Suppose this is true of C_1. Then the black edge of C_1 is direct (Figure 3.b)*

Corollary 1. *An oriented cycle C of size n is transformed by the HP algorithm, in $n-1$ reversals, into n cycles of size 1. If C is direct, the n cycles are all direct. If not, only one of these cycles is indirect.*

Corollary 1 states that for each oriented cycle C, the HP algorithm gathers all the genes to delete from C into a single segment. At the end of the algorithm, a single deletion is required for each oriented cycle. Now, consider the case of unoriented cycles forming hurdles. HP solve hurdles by merging and cutting them in a specific way, described in the HP algorithm (Figure 1). The merging of two hurdles $\mathcal{H}_1$ and $\mathcal{H}_2$ is achieved by combining two unoriented cycles C_1 and C_2, one from each of the two hurdles. This gets rid of the two hurdles, and creates an oriented cycle. This cycle is then solved by HP as described above. So that there are as few deletions as possible, the strategy is to carry out as many merging as possible of pairs of indirect cycles. This requires a careful hurdle choice, and cycle choice for each hurdle, within the HP algorithm. Notice that, according to HP algorithm, merging occurs only when the total number of hurdles is even. In order to keep as many indirect hurdles as possible (to be able to perform as many merging as possible of pairs of indirect hurdles), if the initial number of hurdles is odd and one hurdle must be cut, we will prefer to do this to a direct hurdle. In summary:

hurdle merging
Choose two non-consecutive hurdles $\mathcal{H}_1$ and $\mathcal{H}_2$, indirect if possible, in a certain way;
Choose e in $\mathcal{H}_1$ and f in $\mathcal{H}_2$, indirect if possible;
hurdle cutting
Choose a simple hurdle $\mathcal{H}$, direct if possible;
Perform the reversal determined by any 2 black edges from the same cycle of $\mathcal{H}$.

It remains to specify precisely how to choose hurdles to merge. Let $\mathcal{SH}$ be the set of hurdles of $\mathcal{G}(G,H)$, and let $h(G,H) = |\mathcal{SH}|$. We subdivide $\mathcal{SH}$ into subsets $I_1, D_1, \cdots I_p, D_p$, where each I_i is a maximal set made up of consecutive indirect hurdles, bounded at either end by maximal sets D_{i-1} and D_{i+1} of consecutive direct hurdles. The order within the sets I_i and D_i is arbitrary. We set $I_{p+1} = I_1$, and $I_0 = I_p$ (and similarly for D_{p+1} and D_0). The sets I_i, for $1 \leq i \leq p$, are called indirect. The order in which hurdle merging occurs is given by the procedure **Indirect-hurdle-merging** described in Figure 4.

Let $\mu(G,H)$ be the number of merging during procedure Indirect-hurdle-merging.

Lemma 2. *Procedure Indirect-hurdle-merging is correct, i.e. it respects the merging constraints of the HP algorithm. Moreover, if* $p = 1$, $h(G,H)$ *is even and there is at least one direct hurdle, then* $\mu(G,H) = \frac{h(G,H)}{2} - 1$. *Otherwise,* $\mu(\mathcal{G}) = \left\lfloor \frac{h(G,H)}{2} \right\rfloor$.

Lemma 3. *Let* μ_P *be the number of merging applied to pairs of indirect hurdles of* $\mathcal{G}(G,H)$, *performed by the HP algorithm , according to any order* P. *Then* $\mu(G,H) \geq \mu_P$.

From now on, when we refer to the HP algorithm, this choice procedure will be assumed. No specific constraints are required to merge hurdles (direct plus at most one indirect) that remain unmerged after the procedure.

Procedure Indirect-hurdle-merging
For all i, $1 \leq i \leq p$, do
  While there are at least three hurdles in I_i do
    Perform the merging of any two non-consecutive hurdles in I_i;
{*At the end of this step, it remains p indirect sets, each of size 1 or 2*}
If $p > 1$
  Let $I'_1, \cdots, I'_r$ be the obtained indirect sets of size 1
  For all $j \in \{1, 3, \cdots r - 1 - (r \ mod\ 2)\}$ do
    Merge the hurdle of I'_j with the hurdle of I'_{j+1};
  Let $I''_1, \cdots, I''_s$ be the obtained indirect sets of size 2
  For all j, $1 \leq j \leq s$, do
    Merge one of the two hurdles of I''_j with one of the two hurdles of I''_{j+1};
Else
  If $p = 1$, I_1 is of size 2, and there are no direct hurdles
    Merge the two hurdles of I_1.

Fig. 4. A suitable order for hurdle merging

Let $\mathcal{G}_1(G, H)$ be the graph containing only size 1 cycles obtained after applying the HP algorithm to the graph $\mathcal{G}(G, H)$. Consider the procedure **Deletion**, described as follow, applied to $\mathcal{G}_1(G, H)$:

Procedure Deletion:
For each indirect edge $e = (a, b)$ of $\mathcal{G}_1(G, H)$
  Delete the gene segment between a and b, symbolized by $\delta(a)$.

We will call the HP algorithm augmented by Procedure Deletion the **Reversal-deletion algorithm**. This algorithm transforms the genome G into the genome H. Denote by $ic(G, H)$ the number of indirect cycles of $\mathcal{G}(G, H)$, and by $R(\overline{G}, H)$ the minimum number of reversals necessary to transform $\overline{G}$ into H. This value is obtained by formula (1) of HP (section 2).

Theorem 1. *The Reversal-deletion algorithm performs* $R(\overline{G}, H)$ *reversals and* $D(G, H) = ic(G, H) - \mu(G, H)$ *deletions. Therefore:* $RD(G, H) = R(\overline{G}, H) + D(G, H)$. *Moreover,* $RD(G, H)$ *is the minimum number of rearrangement operations necessary to transform* G *into* H.

4 Insertions

The case $\mathcal{A}_G = \emptyset$, where the set of genes of G is a subset of the set of genes of H, i.e. the problem of transforming G into H with a minimum of reversals and insertions, can be solved by the reversals-deletions analysis, where G takes on the role of H, and vice versa. Each deletion in the H-to-G solution becomes an insertion in the G-to-H. Each reversal in one direction corresponds to a reversal of the same segment in the opposite direction. Nevertheless, we will examine the case of insertions in order to introduce some concepts for the subsequent analysis.

Consider then the problem of transforming H to G by a minimum number of reversals and deletions. Let $\mathcal{G}(H, G)$ be the corresponding graph. We will

not, however, replace the genes of H to be deleted by the δ parameters. Let $RD(H,G) = R(\overline{H},G) + D(H,G)$ be the minimum number of reversals and deletions necessary to transform H into G, where $R(\overline{H},G)$ is the number of reversals and $D(H,G)$ the corresponding number of deletions. Let $\mathcal{G}_1$ be the graph containing only cycles of size 1 obtained by applying the HP algorithm to the graph $\mathcal{G}(H,G)$. The number of indirect cycles of this graph is $D(H,G)$ (with the definition given in theorem 1). Each indirect black edge of $\mathcal{G}_1$ is labeled by a sequence of genes of $\mathcal{A}_H$. Consider the genome G^c obtained from G as follows: for each pair of adjacent vertices a and b in G, if (a,b) is an indirect black edge of $\mathcal{G}_1$ labeled by the sequence of genes $y_1 \cdots y_p$ de $\mathcal{A}_H$, then insert this sequence of genes between a and b in G. We call G^c the **filled genome of** G relative to H.

Lemma 4. *The minimum number of reversals and insertions necessary to transform G into H is $RI(G,H) = R(\overline{H},G) + D(H,G)$. Moreover, $R(\overline{H},G)$ is the number of reversals, and $D(H,G)$ is the number of insertions.*

Example 3. *Consider the sets $\mathcal{A} = \{a,b,c,d,e,f,g\}$, $\mathcal{A}_H = \{y_1,y_2,y_3,y_4\}$, and the two genomes $G = +a\ +b\ +c\ +d\ +e\ +f\ +g$ and $H = +y_1\ +d\ +y_2\ +f\ +b\ +y_3\ +c\ +e\ +y_4\ +a\ +g$. Figure 5 depicts the graph $\mathcal{G}(H,G)$.*

Given that $\mathcal{G}(H,G)$ contains 7 black edges, 3 cycles and 1 hurdle, $R(\overline{H},G) = 7-3+1 = 5$. The HP algorithm does 5 reversals, and produces the following filled genome: $G^c = +y_1\ -y_2\ -y_4\ +a\ +b\ +y_3\ +c\ +d\ +e\ +f\ +g$. Two insertions are necessary to transform G into G^c.

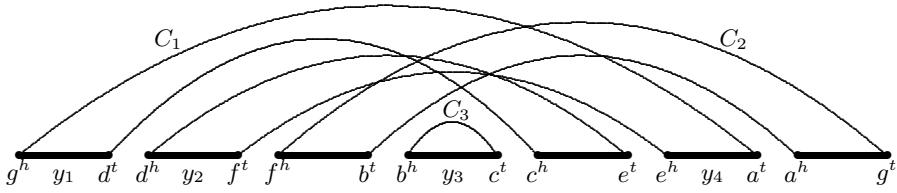


Fig. 5. $\mathcal{G}(H,G)$ contains 7 black edges, 3 cycles, of which 2 are indirect, and 1 hurdle.

5 Insertions and Deletions

Consider now the general case where $\mathcal{A}_G \neq \emptyset$ and $\mathcal{A}_H \neq \emptyset$. Let $\overline{G}$ be the reduced genome of G, and $\overline{H}$ the reduced genome of H.

Lemma 5. *Let $RDI(G,H)$ be the minimum number of rearrangement operations necessary to transform G into H. Then $RDI(G,H) \geq R(\overline{G},\overline{H}) + D(H,\overline{G}) + D(G,\overline{H})$.*

Let $\overline{G}^c$ be the filled genome of $\overline{G}$ relative to H. Given that $\overline{G}^c$ and H contain the same set of vertices, the edges of $\mathcal{G}(\overline{G}^c,H)$ must all be direct.

We must now take into account the vertices of $\mathcal{A}_G$. Let a and b be two adjacent vertices in $\overline{G}$, separated in G by a segment $X(a)$ of vertices of $\mathcal{A}_G$. If a and b are separated in $\overline{G}^c$ by a segment $Y(a)$ of vertices of $\mathcal{A}_H$, then we have the choice to place $X(a)$ before or after $Y(a)$. This choice is made so that we require as few deletions as possible to transform $\overline{G}^c$ into H. By G^c we denote the genome obtained by appropriately adding the vertices of $\mathcal{A}_G$ to $\overline{G}^c$. The graph $\mathcal{G}(G^c, H)$ is obtained from $\mathcal{G}(\overline{G}^c, H)$ by transforming certain black edges of $\mathcal{G}(\overline{G}^c, H)$ into indirect edges. The procedure **Construct-G^c** constructs such a graph $\mathcal{G}(G^c, H)$ corresponding to a genome G^c, from the graph $\mathcal{G}(\overline{G}^c, H)$. We write $X(a)$ for the gene segment of $\mathcal{A}_G$ that follows vertex a in G. At each step of the construction, we say that a cycle (or a hurdle) of the obtained graph is indirect, if it contains indirect edges (constructed during previous steps). In addition, note that when an edge of the graph is labeled, it becomes indirect.

Procedure Construct-G^c:
For each black edge (a, b) of $\mathcal{G}(\overline{G}^c, H)$ such that a is succeeded in G by a segment $X(a)$ do
  If (a, b) is a (direct) black edge of $\mathcal{G}(\overline{G}^c, H)$
    Transform (a, b) into indirect edge by labeling it by $X(a)$;
  Otherwise, let $y_1 \cdots y_p$ be the sequence of vertices of $\mathcal{A}_H$ situated between a and b in $\overline{G}^c$
    If one of the two black edges (a, y_1) or (y_p, b) belongs to an indirect cycle,
      Consider one such edge and label it by $X(a)$;
    Otherwise
      If one of the black edges (a, y_1) or (y_p, b) belongs to an indirect hurdle
        Consider one such edge and label it by $X(a)$;
      Otherwise
        Label by $X(a)$ either one of the two edges;

After having obtained the genome G^c from G by $D(H, \overline{G})$ suitable insertions, the transformation of G^c into H is obtained by the Reversal-deletion algorithm applied to the graph $\mathcal{G}(G^c, H)$. This algorithm does $R(\overline{G}, \overline{H})$ reversals and $D(G^c, H)$ deletions.

We summarize the transformation from G to H as follows:

1. Apply the HP algorithm to the graph $\mathcal{G}(H, \overline{G})$. The final graph obtained contains $D(H, \overline{G})$ indirect edges.
2. Do the $D(H, \overline{G})$ insertions deducible from the indirect edges, and transform $\overline{G}$ to $\overline{G}^c$.
3. Fill the genome $\overline{G}^c$ to obtain genome G^c, using Procedure Construct-G^c.
4. Apply the Reversal-deletion algorithm to the graph $\mathcal{G}(G^c, H)$. The algorithm does $R(\overline{G}, \overline{H})$ reversals and $D(G^c, H)$ deletions.

Proposition 1. *The number of rearrangement operations occurring during the application of the method is:* $RDI_1(G, H) = R(\overline{G}, \overline{H}) + D(H, \overline{G}) + D(G^c, H)$, *where* $R(\overline{G}, \overline{H})$ *is the number of reversals,* $D(H, \overline{G})$ *is the number of insertions and* $D(G^c, H)$ *is the number of deletions.*

Moreover, the number of reversals and insertions is minimal, and if $D(G^c, H) = D(G, \overline{H})$, *then the total number of rearrangement operations is minimal.*

Example 4. *Consider the sets* $\mathcal{A} = \{a, b, c, d, e, f, g\}$, $\mathcal{A}_G = \{x_1, x_2, x_3, x_4, x_5\}$, $\mathcal{A}_H = \{y_1, y_2, y_3, y_4\}$, *and the two genomes* $G = +x_1\ +a\ +x_2\ +b\ +c +x_3\ +d\ +x_4\ +e\ +x_5\ +\underline{f}\ +g$, *and* $H = +y_1\ +d\ +y_2\ +f\ +b\ +y_3\ +c\ +e\ +y_4\ +a\ +g$.

Given that $\overline{G}$ *and* H *are the two genomes in Example 3, Figure 5 represents the graph* $\mathcal{G}(H, \overline{G})$. *Two insertions are necessary to transform* $\overline{G}$ *into* $\overline{G}^c = +y_1\ -y_2\ -y_4 +a\ +b\ +y_3\ +c\ +d\ +e\ +f\ +g$.

Figure 6 depicts $\mathcal{G}(G^c, H)$, *which contains two indirect cycles* C_1 *and* C_2, *and a single hurdle. Therefore* $D(G^c, H) = 2$. *Moreover, since* $R(\overline{G}, \overline{H}) = 5$ *(Example 3), 5 reversals followed by 2 deletions are necessary to convert* G^c *to* H. *The total number of rearrangement operations used to transform* G *to* H *is thus* $2 + 5 + 2 = 9$.

On the other hand, the construction of graph $\mathcal{G}(G, \overline{H})$ *reveals that* $D(G, \overline{H}) = 2$. *Nine is thus the minimal number of rearrangement operations necessary to transform* G *to* H.

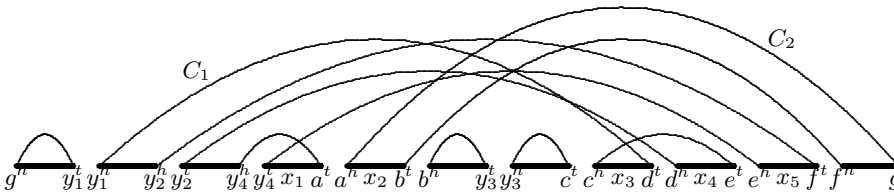


Fig. 6. The graph $\mathcal{G}(G^c, H)$.

Theorem 2. *Let* $ie(G, \overline{H})$ *be the number of indirect edges of* $\mathcal{G}(G, \overline{H})$, *and let* $RDI_1(G, H)$ *be the number of rearrangement operations obtained by the method described above. Then:*

$$R(\overline{G}, \overline{H}) + D(H, \overline{G}) + D(G, \overline{H}) \leq RDI_1(G, H) \leq R(\overline{G}, \overline{H}) + D(H, \overline{G}) + ie(G, \overline{H})$$

Remark 2: A better solution will, sometimes, be found by calculating the minimum number of operations necessary to transform H to G. Thus, it is a good idea to always consider $\min(RDI_1(G, H), RDI_1(H, G))$.

The time complexity of our method depends on the complexity of the HP algorithm. If we require only the minimum number of reversals necessary to transform a genome G into a genome H, both of size n, it suffices to construct the cycle graph of G and H, and to count the number of cycles, super-hurdles and simple hurdles within it. As presented in [HP95a], the cycle graph is constructed in $O(n^2)$ time. The complexity becomes $O(n^4)$ if we want to produce an explicit sequence of reversals which transforms G to H. The HP algorithm was sped up by Berman and Hannenhalli [BH96]. The complexities they report are expressed in terms of $\alpha(n)$, the inverse of Ackerman's function. The construction of the cycle graph is $O(n\alpha(n))$ and the actual production of the reversals is $O(n^2\alpha(n))$.

Kaplan, Shamir and Tarjan [KST97] further improved on this complexity. First, they considered a simplified representation of the graph and for efficiency

considerations, they avoided generating it explicitly. Secondly, they considered the notion of "happy clique" to simplify the reversal choice procedure to solve good components. Their method does not change hurdles cutting and hurdles merging procedures. With these improvements, the time complexity for the production of a minimal sequence of reversals becomes $O(n^2)$.

As our modifications to take into account insertions and deletions are not affected by the way reversals are chosen to solve good components, they are compatible with Kaplan, Shamir and Tarjan improvements. Therefore, we can assume their complexity for the HP algorithm in analyzing our extension to deletion and insertion.

Theorem 3. *Consider* $n = \max(|\mathcal{A}| + |\mathcal{A}_G|, |\mathcal{A}| + |\mathcal{A}_H|)$. *The transformation of* G *to* H *by the above method is of time complexity* $O(n^2)$.

Remark 3: To obtain the parameter $D(G^c, H)$, we should construct genome G^c. To do so, the HP algorithm must be applied to the graph $\mathcal{G}(H, \overline{G})$. The complexity of our method thus remains $O(n^2)$, even if we are only looking for the minimum number of rearrangement operations, and not for the actual operations.

6 An Application to the Comparison of Mitochondrial Genomes

To test our method on some realistic data of moderate size, we selected three mitochondrial genomes from GOBASE archives [KLRB+98]. *Reclimonas americana* is thought to most resemble the ancestral mitochondrion, and other mitochondria to have evolved through deletion of segments or individual genes from the set of genes found in *Reclinomonas*. *Allomyces macrogynus* is a fungus and *Tetrahymena pyriformis* an unrelated protist. To simplify the problem somewhat, we discarded all tRNA genes from the gene orders, since the homological correspondences among tRNAs is not always clear in mitochondria. This left 66 genes for *Reclinomonas*, 17 for *Allomyces* and 23 for *Tetrahymena*.

Reclinomonas contains 49 genes in addition to those in *Allomyces*, and these genes are scattered among 10 segments in the genome. The transformation of the *Reclinomonas* genome to the *Allomyces* one by our algorithm requires 14 reversals but only three segment deletions.

Reclinomonas contains 43 genes in addition to those in *Tetrahymena*, and these genes are scattered among 11 segments in the genome. The transformation of the *Reclinomonas* genome to the *Tetrahymena* one by our algorithm requires 19 reversals but only two segment deletions.

Allomyces and *Tetrahymena* have 12 genes in common. The five supplementary genes in *Allomyces* are each in a separate segment, and the 11 supplementary genes in *Tetrahymena* are found in 6 segments. Converting the *Tetrahymena* genome into the *Allomyces* one by our algorithm requires 9 reversals, 2 insertions et 3 deletions. Moreover, the construction of the graph $\mathcal{G}(G, \overline{H})$ reveals that this number of operations is minimal.

7 Conclusion

The approach taken in this paper could readily be extended to a distance defined on multichromosomal genomes, where the rearrangement operations include reciprocal translocation, chromosome fusion and chromosome fission, as well as reversal, insertion and deletion. The Hannenhalli and Pevzner approach to translocation and reversal distance [HP95b] makes use of the same kind of cycle graph that we exploit in the present research.

Acknowledgments

Research supported in part by a grant from the Natural Science and Engineering Research Council of Canada. The author is a Scholar in the Evolutionary Biology Program of the Canadian Institute for Advanced Research.

References

BH96. P. Berman and S. Hannenhalli. Fast sorting by reversals. In D. Hirschberg and G. Myers, editors, *Combinatorial Pattern Matching, Proceedings of the 7th Annual Symposium*, number 1075 in Lecture Notes in Computer Science, pages 168–185. Springer, 1996.

Cap97. A. Caprara. Sorting by reversals is difficult. In *Proceedings of the First Annual International Conference on Computational Molecular Biology (RECOMB 97)*, pages 75–83, New York, 1997. ACM.

Cap99. A. Caprara. Formulations and hardness of multiple sorting by reversals. In S. Istrail, P.A. Pevzner, and M.S. Waterman, editors, *Proceedings of the Third Annual International Conference on Computational Molecular Biology (RECOMB 99)*, pages 84–93, New York, 1999. ACM.

Han95. S. Hannenhalli. Polynomial-time algorithm for computing translocation distance between genomes. In Z. Galil and E. Ukkonen, editors, *Sixth Annual Symposium on Combinatorial Pattern Matching*, volume 937 of *Lecture Notes in Computer Science*, pages 162–176, Berlin, 1995. Springer.

HP95a. S. Hannenhalli and P.A. Pevzner. Transforming cabbage into turnip. (polynomial algorithm for sorting signed permutations by reversals). In *Proceedings of the 27th Annual ACM-SIAM Symposium on the Theory of Computing*, pages 178–189, 1995.

HP95b. S. Hannenhalli and P.A. Pevzner. Transforming men into mice (polynomial algorithm for genomic distance problem). In *Proceedings of the IEEE 36th Annual Symposium on Foundations of Computer Science*, pages 581–592, 1995.

KLRB$^+$98. M. Korab-Laskowska, P. Rioux, N. Brossard, TG. Littlejohn, MW. Gray, BF. Lang, and G. Burger. The organelle genome database project (gobase). *Nucleic Acids Res*, 26:139–146, 1998.

KST97. H. Kaplan, R. Shamir, and R.E. Tarjan. Faster and simpler algorithm for sorting signed permutations by reversals. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 344–351, New York, 1997.

SM-97. *Introduction to Computational Molecular Biology.* PWS Publishing Company, 1997.

A Lower Bound for the Breakpoint Phylogeny Problem

David Bryant*

CRM Université de Montréal. CP 6128, succ. centre-ville
Montréal H3C 3J7, Québec
bryant@crm.umontreal.ca

Abstract. Breakpoint phylogenies methods have been shown to be an effective way to extract phylogenetic information from gene order data. Currently, the only practical breakpoint phylogeny algorithms for the analysis of large genomes with varied gene content are heuristics with no optimality guarantee. Here we address this shortcoming by describing new bounds for the breakpoint median problem, and for the more complicated breakpoint phylogeny problem. In both cases we employ Lagrangian multipliers and subgradient optimization to tighten the bounds. The experimental results are promising: we achieve lower bounds close to the upper bounds established using breakpoint phylogeny heuristics.

1 Phylogenetic Inference Using Gene Order Data

Molecular systematics has long focused on the comparison of DNA or protein sequences which have evolved through the local operations of replacement of one term by another, or the insertion or deletion of one or more terms. With the growing availability of genomic data in recent years, however, attention has been drawn to global rearrangements over the whole genome, rather than just local nucleotide or amino acid patterns. As genomes evolve, genes are inserted or deleted, segments of the genome are reversed, or removed and re-inserted at a new position. This leads to different genomes having homologous genes arranged in different orders on their gene maps.

For some time, gene order maps have been investigated as a possible source of phylogenetic information [13]. The approach is particularly attractive for groups of organisms, such as the early branching eukaryotes studied here, that have been difficult to analyse using conventional sequence based methods [16].

Much early work in the field focused on the estimation of rearrangement distances. A phylogeny is then constructed using standard distance based tree estimation methods [13,5]. Such an approach is limited by the inability of distance based methods to estimate ancestral gene orders.

An alternative to distance based approaches is to construct phylogenies and ancestral gene orders using a parsimony criterion, where the length of an edge in the tree is determined according to a rearrangement distance. Unfortunately,

* Supported in part by a Bioinformatics Postdoc. Fellowship from the CIAR, Evolutionary Biology Program and by NSERC and CGAT grants to D. Sankoff.

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 235–247, 2000.

the extension of rearrangement distances to more than two genomes proved to be computationally difficult [3]. Existing heuristic methods work well for only very small problems [4,14].

1.1 Breakpoint Analysis of Gene Order Data

The breakpoint phylogeny method for the analysis of gene order data was introduced by Sankoff and Blanchette [15]. The approach avoids many of the computational difficulties inherent in rearrangement distance based methods. Recently, the method has been applied successfully to the study of early eukaryote evolution, extracting clear phylogenetic signals from genomes that vary greatly in both gene order and content [16].

At the current time, the only practical methods for the analysis of gene order data using breakpoints are heuristics. The breakpoint median problem (see section 1.3) and the breakpoint phylogeny problem (see section 1.4) are both NP-hard. Approximation algorithms, based on the traveling salesman problem (TSP), are possible for the breakpoint median problem when the genomes have equal gene sets, however these all break down when gene content differs or the problem is extended to a phylogenetic tree. In any case, the combination of fast heuristics and tight lower bounds have been of far more practical use in the TSP than approximation algorithms. They give much closer optimality guarantees, and can be used as the basis for branch and bound methods.

1.2 Gene Order Data and the Normalised Breakpoint Distance

A genome A is described in terms of a (circular) ordering $A = \langle a_1, a_2, \ldots, a_n, a_1 \rangle$. The genes are signed either positive (a_i) or negative ($-a_i$) to indicate the transcription direction of the gene. Reversing the direction and signs of a genome therefore gives the same genome. We let $\mathcal{G}(A)$ denote the set of genes in A, including both positive and negative copies of each gene. For each $a_i \in \mathcal{G}(A)$ we let $succ(a_i, A)$ denote the **successor** of g in A. In this example, $succ(a_i, A) = a_{i+1}$ and $succ(a_n, A) = a_1$. Note that $succ(g, A) = h$ if and only if $succ(-h, A) = -g$.

Let A be a genome and let W be a set of genes. We let $A|_W$ denote the genome A with all genes *not* in W removed, and all remaining genes left in the same order. The **(normalised) breakpoint distance** between two genomes A and B is then defined

$$d(A, B) = \frac{1}{2|\mathcal{G}(A) \cap \mathcal{G}(B)|} |\{g \in \mathcal{G}(A) \cap \mathcal{G}(B) : succ(g, A|_{\mathcal{G}(B)}) \neq succ(g, B|_{\mathcal{G}(A)})\}| \tag{1}$$

Thus $d(A, B)$ is unaffected by genes that only appear in one of A or B. We note that this breakpoint distance differs from that used in [2,1] by the introduction of the scaling factor $\frac{1}{2|G(A) \cap G(B)|}$. The normalisation is introduced to overcome a problem of the former breakpoint distances: when there is a great deal of variation in gene content, unnormalised breakpoint distances will tend to produce trees where large genomes are forced apart [16].

1.3 The Breakpoint Median Problem

Let $\mathcal{A} = A_1, \ldots, A_N$ be a collection of genomes and define

$$\mathcal{G}(\mathcal{A}) = \mathcal{G}(A_1) \cup \mathcal{G}(A_2) \cdots \cup \mathcal{G}(A_N) \ . \tag{2}$$

We will assume that each gene $g \in \mathcal{G}(\mathcal{A})$ appears in at least two genomes, as genes appearing in only one genome do not contribute to gene order information.

The median score $\Psi(X, \mathcal{A})$ of a genome X with gene set $\mathcal{G}(X) = \mathcal{G}(\mathcal{A})$ is defined

$$\Psi(X, \mathcal{A}) = \sum_{i=1}^{N} d(X, A_i) \tag{3}$$

and the breakpoint median problem is to find X with $\mathcal{G}(X) = \mathcal{G}(\mathcal{A})$ that minimizes $\Psi(X, \mathcal{A})$. The breakpoint median problem is NP-hard, even for three genomes with equal gene content [10]. However, when the genomes do have equal gene content the problem can be converted into a travelling salesman problem of the same size, and solved for small to moderate instances. The reduction does not apply to the case when genomes have unequal gene content. In this case, one can apply the gene by gene insertion heuristic of [16]. The lower bounds in this paper can be used to check how close a heuristic solution is to optimal.

1.4 The Breakpoint Phylogeny Problem

Let $T = (V, E(T))$ be a binary rooted tree with N leaves. We direct all edges of T towards the root. To each leaf we assign a different genome A_i from the input collection $\mathcal{A} = A_1, A_2, \ldots, A_N$. A **valid assignment** of genomes to *internal* nodes of T is a function $\mathcal{X}$ from V to the set of genomes satisfying:

1. For each leaf $v \in V$, $\mathcal{X}(v)$ is the genome assigned to A_i.
2. For each internal node $v \in V$ with children u_1, u_2 we have $\mathcal{G}(\mathcal{X}(v)) = \mathcal{G}(\mathcal{X}(u_1)) \cup \mathcal{G}(\mathcal{X}(u_2))$.

The second condition models the situation in early branching eukaryotes where all genes are believed to be present in the ancestral genome and gene content differs only through deletions.

The **length** of a tree T with respect to the input genomes $\mathcal{A}$ and a valid assignment $\mathcal{X}$ is defined

$$l(\mathcal{X}, \mathcal{A}, T) = \sum_{(u,v) \in E(T)} d(\mathcal{X}(u), \mathcal{X}(v)) \tag{4}$$

and the **breakpoint phylogeny problem** is to find a valid assignment $\mathcal{X}$ minimizing $l(\mathcal{X}, \mathcal{A}, T)$.

Current heuristics for the breakpoint phylogeny problem use multiple applications of a breakpoint median method [15,16]. Initially, a valid assignment for the tree is determined randomly. The heuristic makes multiple passes through the tree. At each internal node v it constructs a solution to the median breakpoint problem with input equal to the genomes assigned to the neighbours of v. If the solution gives an assignment of shorter length, the new genome is assigned to v. The procedure repeats until a local optimum is reached.

2 A Lower Bound for the Breakpoint Median Problem

In many ways, the lower bound we develop for the breakpoint median problem parallels the 2-neighbour bound for the TSP. In the TSP the length of a tour is the sum of the lengths of the edges of a tour, which is equal to twice the sum of distances from each city to the two adjacent cities in the tour. If we calculate, for each city x, the two closest cities to x, and then sum these distances over all cities, we obtain a lower bound for the length of a tour.

The same idea holds for the breakpoint median problem: we calculate an optimum collection of successors for each gene and then sum these to obtain a lower bound. The situation is complicated by the fact that the breakpoint distance is calculated from induced genomes. We can no longer calculate a single "closest successor" but instead calculate a vector of closest successors. Like the 2-neighbour bound, we are calculating successors in both directions: successors for g and successors for $-g$.

2.1 Local Optimum Bound

Let A and B be two signed genomes. For every gene $g \in \mathcal{G}(A) \cap \mathcal{G}(B)$ we define the **local breakpoint distance**

$$d_g(A,B) = \begin{cases} \frac{1}{|\mathcal{G}(A)\cap\mathcal{G}(B)|}, & \text{if } succ(g, A|_{\mathcal{G}(B)}) \neq succ(g, B|_{\mathcal{G}(A)}); \\ 0, & \text{otherwise.} \end{cases} \tag{5}$$

If $g \notin \mathcal{G}(A) \cap \mathcal{G}(B)$ then $d_g(A,B) = 0$. The breakpoint distance between two genomes can then be composed into the sum of local breakpoint distances, using

$$d(A,B) = \tfrac{1}{2} \sum_{g \in \mathcal{G}(A)\cup\mathcal{G}(B)} d_g(A,B) \ . \tag{6}$$

We can decompose the median score $\Psi(X)$ in the same way. Let $\mathcal{A} = A_1, A_2, \ldots, A_N$ be a collection of signed genomes. Let X be a genome with gene set $\mathcal{G}(\mathcal{A})$. For each $g \in \mathcal{G}(\mathcal{A})$ we define

$$\Psi_g(X,\mathcal{A}) = \sum_{i=1}^{N} d_g(X, A_i) \tag{7}$$

so that

$$\Psi(X,\mathcal{A}) = \tfrac{1}{2} \sum_{g \in \mathcal{G}(\mathcal{A})} \Psi_g(X,\mathcal{A}) \tag{8}$$

For each gene $g \in \mathcal{G}$ define

$$\Psi^*_g(\mathcal{A}) = \min_X \{\Psi_g(X,\mathcal{A}) : \mathcal{G}(X) = \mathcal{G}(\mathcal{A})\} \ . \tag{9}$$

The **local optimum bound** for the breakpoint median problem with input genomes $\mathcal{A}$ is then defined

$$LOB(\mathcal{A}) = \tfrac{1}{2} \sum_{g \in \mathcal{G}(\mathcal{A})} \Psi^*_g(\mathcal{A}) \ . \tag{10}$$

Lemma 1. *Let $\mathcal{A}$ be a collection of genomes. For all genomes X with gene set $\mathcal{G}(\mathcal{A})$ we have*

$$\Psi(X, \mathcal{A}) \geq LOB(\mathcal{A}) \ . \tag{11}$$

2.2 Computing the Local Optimum Bound

The utility of the local optimum bound depends on our ability to compute $\Psi_g^*(\mathcal{A})$ quickly. Here we show that if the number N of input genomes is bounded (as is usually the case) then $\Psi_g^*(\mathcal{A})$ can be computed in $O(n)$ time, but if the number of genomes is unbounded then computing $LOB(\mathcal{A})$ (and therefore computing $\Psi_g^*(\mathcal{A})$) is NP-hard.

The first and most important observation is that the only genes of X that influence $\Psi_g(X, \mathcal{A})$ are those in $\{succ(g, X|_{\mathcal{G}(A_i)}) : i = 1, \ldots, N\}$. We can therefore assume that X is of the form

$$\langle g, h_1, h_2, \ldots, h_k, a_1, \ldots, a_m, g \rangle \tag{12}$$

where each h_j equals $succ(g, X|_{\mathcal{G}(A_i)})$ for some $i \in \{1, \ldots, N\}$ and none of the genes a_j do. The problem of optimizing $\Psi_g(X, \mathcal{A})$ then boils down to finding optimal successors $h_1, \ldots, h_k$. This can be done using a simple search tree, where each node of the tree except the root is labelled by a gene. The children of the root are labelled by different choices for h_1, their children are labelled by choices for h_2, and so on. For a given node, if every genome has a gene from that node or an ancestor of that node we stop and evaluate the corresponding sequence of successors, otherwise we branch on possible choices for the next successor.

The depth of this search tree is bounded by N, the number of genomes. The degree is bounded by the number of genes we need to search through.

Define

$$S(g, \mathcal{A}) = \{succ(g, A_i) : g \in \mathcal{G}(A_i), i = 1, \ldots, N\} \ . \tag{13}$$

Lemma 2. *There is X minimizing $\Psi_g(X, \mathcal{A})$ such that*

$$\{succ(g, X|_{\mathcal{G}(A_i)}) : i = 1, \ldots, N\} \subseteq S(g, \mathcal{A}) \ . \tag{14}$$

Hence we can minimize $\Psi_g(X, \mathcal{A})$ in $O(NN!)$ time.

It follows that we can calculate $LOB(\mathcal{A})$ in $O(nNN!)$ time. This bound is useful when the number of genomes is bounded, but impractical when the number of genomes is large. In this case, it appears that there will be no efficient algorithm for computing LOB.

Lemma 3. *It is NP-hard to calculate $LOB(A_1, \ldots, A_N)$ for a set of genomes $\mathcal{A}$ with unequal gene sets. Hence it is also NP-hard to calculate $\Psi_g^*(\mathcal{A})$ for a given gene g.*

Proof. We provide a reduction from FEEDBACK ARC SET [8], which is NP-complete even when there is no pair of vertices u, v with an arc from u to v and another arc from v to u. Let the directed graph $G = (V, E)$ and number

$K \leq |E|$ make up an arbitrary instance of FEEDBACK ARC SET satisfying this condition. Put $\mathcal{G} = V \cup \{x\}$ where x is a new gene. Construct a collection of genomes

$$\mathcal{A} = \{\langle x, a, a', x\rangle : (a, a') \in E\} \tag{15}$$

which all have genes in $\mathcal{G}$. We claim that $G = (V, E)$ has a feedback arc set $E' \subseteq E$ of size K if and only if $LOB(\mathcal{A}) \leq K$.

For each $a \in V$, put $B = \{b : (a, b) \in E\}$, let $b_1, \ldots, b_k$ be an arbitrary ordering of B, and consider the subsequence $a, b_1, \ldots, b_k, x$. This gives $\Psi^*_a(\mathcal{A}) = 0$ for all $a \in V$. The same trick (in reverse) gives $\Psi^*_{-a}(\mathcal{A}) = 0$ for all $a \in V$. Hence $LOB(\mathcal{A}) = \frac{1}{2}(\Psi^*_x(\mathcal{A}) + \Psi^*_{-x}(\mathcal{A}))$.

Suppose that E' is a FEEDBACK arc set of size K. Let $a_1, \ldots, a_n$ be an ordering of V such that for each arc $(a_i, a_j) \in E - E'$ we have $i < j$. Consider $X = \langle x, a_1, a_2, \ldots, a_n, x\rangle$. Then $\Psi_x(X, \mathcal{A})$ is the number of arcs $(a_j, a_i) \in E$ such that $j > i$, and $\Psi_{-x}(X, \mathcal{A}) = \Psi_x(X, \mathcal{A})$. Hence $LOB(\mathcal{A}) \leq |E'| \leq K$.

Conversely, suppose that $LOB(\mathcal{A}) \leq K$. Without loss of generality, $\Psi_x(X, \mathcal{A}) \leq K$. Let $X = \langle x, a_1, a_2, \ldots, a_n, x\rangle$ be a genome with $\Psi_x(X, \mathcal{A}) = \Psi^*_x(\mathcal{A})$. There are at most K genomes $\langle x, a_j, a_i, x\rangle \in \mathcal{A}$ such that $j > i$, and the corresponding edges form a feedback arc set for G.

We stress, however, that in most current applications the number of genomes N in the breakpoint median problem is bounded (typically $N \leq 3$). In these cases, the above NP-hardness result is irrelevant.

2.3 Tightening the Bound Using Lagrange Multipliers

Here we show how the introduction of Lagrange multipliers can be used to tighten the bound.

For every genome A_i and every gene $g \in \mathcal{G}(A_i)$ we introduce a Lagrangian multiplier $\lambda[A_i, g]$. We use the Lagrange multipliers to weight the decomposed Ψ score. For each gene g, define

$$\Psi_g(X, \mathcal{A}, \lambda) = \Psi_g(X, \mathcal{A}) + \sum_{i=1}^{N} \lambda[A_i, succ(g, X|_{\mathcal{G}(A_i)})] \ . \tag{16}$$

Thus for any genome X,

$$\Psi(X, \mathcal{A}) = \tfrac{1}{2}\sum_{i=1}^{N} \Psi_g(X, \mathcal{A}, \lambda) - \tfrac{1}{2}\sum_{i=1}^{N} \sum_{g \in \mathcal{G}(A_i)} \lambda[A_i, g] \ . \tag{17}$$

We define the (weighted) local optima by

$$\Psi^*_g(\mathcal{A}, \lambda) = \min_X \{\Psi_g(X, \mathcal{A}, \lambda) : \mathcal{G}(X) = \mathcal{G}(\mathcal{A})\} \ . \tag{18}$$

The weighted local optimum bound is given by

$$LOB(\mathcal{A}, \lambda) = \tfrac{1}{2}\sum_{g \in \mathcal{G}} \Psi^*_g(\mathcal{A}, \lambda) - \tfrac{1}{2}\sum_{i=1}^{N} \sum_{g \in \mathcal{G}(A_i)} \lambda[A_i, g] \tag{19}$$

so that for all choices of λ, and all genomes X with $\mathcal{G}(X) = \mathcal{G}(\mathcal{A})$, we have

$$LOB(\mathcal{A}, \lambda) \leq \Psi(X, \mathcal{A}) \ . \tag{20}$$

We would like to find λ that gives as tight a bound as possible, that is, λ that maximises $LOB(\mathcal{A}, \lambda)$. We discuss this problem in section 2.5. First, however, we consider the problem of efficiently computing $LOB(\mathcal{A}, \lambda)$ for a given λ.

2.4 Computing $LOB(\mathcal{A}, \lambda)$

We now need to generalise the results obtained in section 2.2 to the case when the genes are weighted. Clearly, when N is unbounded the problem remains NP-hard—the unweighted case is the same as when all λ values are zero. We will therefore assume that N is bounded, with a particular interest in the case $N = 3$.

Lemma 2 does not hold in this case. There could be a gene that is not a successor of g in any of the genomes, but has a large negative weight so would be included in the optimal genome. We use another trick to restrict the number of genes to search.

Suppose that for each genome A_i and each $g \in \mathcal{G}(A_i)$ we have a weight $w(A_i, g)$. For each $I, J \subseteq \{1, 2, \ldots, N\}$ let $\Gamma[I, J]$ be the gene g minimizing $\sum_{i \in I} w(A_i, g)$ over all g such that

$$I \subseteq \{i : g \in \mathcal{G}(A_i)\} \subseteq J \ . \tag{21}$$

The table Γ is called a **successor profile** for $\mathcal{A}$ with respect to weighting function w.

Lemma 4. *Let Γ and γ be successor profiles for $\mathcal{A}$ with respect to the weighting function $w[A_i, h] = \lambda[A_i, h]$. For each $g \in \mathcal{G}(\mathcal{A})$ there is X minimizing $\Psi_g(X, \mathcal{A}, \lambda)$ such that*

$$\{succ(g, X|_{\mathcal{G}(A_i)}) : i = 1, \ldots, N\} \subseteq S(g, \mathcal{A}) \cup \Gamma \ . \tag{22}$$

Since $|\Gamma|$ is $O(3^N)$ we can minimizing $\Psi_g(X, \mathcal{A}, \lambda)$ in $O(N3^{NN})$ time.

Note that the time bound in Lemma 4 is very rough. Generally, not all the elements of Γ need to be determined, and the search tree is much smaller than indicated. For example, when $N = 3$ and each gene appears in at least two genomes, we need only determine 16 entries of Γ and the size of the search tree is about 50 nodes.

2.5 Subgradient Optimization

Since every λ gives a new bound, we wish to find λ that maximizes $LOB(\mathcal{A}, \lambda)$. As in the Held-Karp bound (c.f. [6,7,11]), the lower bound function is convex and piecewise linear. We will use the Lagrangian multipliers to force the choice of successors in the local optimum bound to be more like the successors from

a genome, thereby obtaining a tighter lower bound for the breakpoint median problem.

Fix a particular matrix of values for λ. For each $g \in \mathcal{G}(\mathcal{A})$ let X_g be the genome chosen such that $\Psi_g(X_g, \mathcal{A}, \lambda) = \Psi_g^*(\mathcal{A}, \lambda)$. For each genome A_i and each $h \in \mathcal{G}(A_i)$ we compute the frequency

$$f[A_i, h] = |\{g : succ(g, X_g|_{\mathcal{G}(A_i)}) = h, A_i \in \mathcal{A}\}| \ . \tag{23}$$

We can use f to compute a sub-gradient for LOB_λ at λ. For each $i = 1, \ldots, N$ and each $h \in G_i$ define

$$\Delta[A_i, h] = f[A_i, h] - 1 \ . \tag{24}$$

Intuitively, if a gene h in an genome is chosen as a successor of a lot of genes, then that gene is given a larger weight to encourage less selections of the gene the next time through. Alternatively, if a gene is not selected as a successor then it is given a small weight.

We use Δ as the next search direction in a simple line search (algorithm 1), based on the subgradient optimization algorithm of [11], pg 176.

Algorithm 1: Subgradient optimization of $LOB(\mathcal{A}, \lambda)$.

input : Genomes $\mathcal{A}$. Initial step length τ. Decrement factor $\alpha \in [0, 1]$. Number of iterations m.
output: A lower bound for $Psi(X, \mathcal{A})$.
foreach $i = 1, \ldots, N$ *and* $h \in G_i$ **do** $\lambda[i, h] \leftarrow 0$;
for $k = 1$ *to* m **do**
  Compute the bound $LOB(\mathcal{A}, \lambda)$ and the frequency table f.;
  Compute Δ using equation 24;
  foreach $i = 1, \ldots, N$ *and* $h \in G_i$ **do**
    $\lambda[A_i, h] \leftarrow \lambda[A_i, h] + \tau\Delta$;
  $\tau \leftarrow \alpha\tau$;
Output the largest bound calculated.;
end ;

Our experimental studies indicated an average improvement of 20% from using Lagrangian optimization.

3 A Lower Bound for Breakpoint Phylogeny

While the lower bound we have derived for the breakpoint median problem can be used to assess the performance of the breakpoint median heuristic at each individual internal node, it does not give a lower bound for the entire breakpoint phylogeny problem. Here we describe how the local optimum bound for the breakpoint median problem can be extended to a lower bound with the breakpoint phylogeny problem, including the Lagrangian optimization.

3.1 Local Optimum Bound on a Tree

The intuitive idea behind the lower bound for the breakpoint phylogeny problem is the same as for the local optimum bound for the breakpoint median problem. For each gene we determine an optimal sequence of successors so that the sum of the breakpoint distances, this time over the whole tree, is minimum.

For each gene $g \in \mathcal{G}(\mathcal{A})$ and each valid assignment $\mathcal{X}$ of genomes to nodes of T, we define a "local" score

$$l_g(\mathcal{X}, \mathcal{A}, T) = \sum_{(u,v)\in E(T)} d_g(\mathcal{X}(u), \mathcal{X}(v)) \ . \tag{25}$$

The length of a tree (eqn. 4) can then be written as

$$l(\mathcal{X}, \mathcal{A}, T) = \tfrac{1}{2} \sum_{g\in\mathcal{G}(\mathcal{A})} l_g(\mathcal{X}, \mathcal{A}, T) \ . \tag{26}$$

Let $l_g^*(\mathcal{A}, T)$ be the minimum of $l_g(\mathcal{X}, \mathcal{A}, T)$ over all valid assignments of internal genomes $\mathcal{X}$. The **local optimum bound** for the breakpoint phylogeny problem is then defined as

$$L(\mathcal{A}, T) = \tfrac{1}{2} \sum_{g\in\mathcal{G}(\mathcal{A})} l_g^*(\mathcal{A}, T) \ . \tag{27}$$

so that all valid assignments of internal genomes $\mathcal{X}$,

$$l(\mathcal{X}, \mathcal{A}, T) \geq L(\mathcal{A}, T) \ . \tag{28}$$

3.2 Computing the Local Optimum Bound on a Tree

The local optimum $l_g^*(\mathcal{A}, T)$ can be computed using a combination of the techniques derived for the breakpoint median problem, and a dynamical programming method based on the general Steiner points algorithm described of [12]. Due to space constraints we omit details of the algorithm - it is simply a special case of the algorithm outlined in section 3.4.

3.3 Tightening the Bound Using Lagrangian Multipliers

Once again, we apply Lagrangian multipliers to tighten the bound.

For each edge (u, v) of T and every gene $h \in \mathcal{G}(u)$, we introduce the Lagrangian multiplier $\lambda[u, v, h]$. The multipliers are used to weight choices of successor genes. For each gene $g \in \mathcal{G}(\mathcal{A})$ we define

$$l_g(\mathcal{X}, \mathcal{A}, T, \lambda) = \sum_{(u,v)\in E(T)} \left(d_g(\mathcal{X}(u), \mathcal{X}(v)) + \lambda[u, v, succ(g, \mathcal{X}|_{\mathcal{G}(u)})]\right) \ . \tag{29}$$

The length of a tree can then be rewritten as

$$l(\mathcal{X},\mathcal{A},T) = l(\mathcal{X},\mathcal{A},T,\lambda) = \tfrac{1}{2}\sum_{g\in\mathcal{G}(\mathcal{A})} l_g(\mathcal{X},\mathcal{A},T,\lambda) - \tfrac{1}{2}\sum_{(u,v)\in E(T)}\sum_{h\in\mathcal{G}(u)} \lambda[u,v,h]\,. \tag{30}$$

For each gene $g \in \mathcal{G}(\mathcal{A})$ let $l_g^*(\mathcal{A},T,\lambda)$ denote the minimum of $l_g(\mathcal{X},\mathcal{A},T,\lambda)$ over all valid assignments $\mathcal{X}$. The (weighted) local optimum bound for the breakpoint phylogeny problem is then defined as

$$L(\mathcal{A},T,\lambda) = \tfrac{1}{2}\sum_{g\in\mathcal{G}(\mathcal{A})} l_g^*(\mathcal{A},T,\lambda) \tag{31}$$

so that, for all choices of λ, and all valid assignments $\mathcal{X}$, we have

$$L(\mathcal{A},T,\lambda) \leq l(\mathcal{X},\mathcal{A},T)\ . \tag{32}$$

Once again, the aim is to find λ that gives the largest, and therefore tightest, lower bound.

3.4 Computing $L(\mathcal{A},T,\lambda)$

Here we incorporate Lagrange multipliers into the dynamical programming algorithm briefly outlined in section 3.2. Let v be an internal node of T and let $\mathcal{X}$ be a valid assignment of genomes. We put

$$l_g(\mathcal{X},\mathcal{A},T_v,\lambda) = \sum_{(u,v)\in E(T_v)} \big(d_g(\mathcal{X}(u),\mathcal{X}(v)) + \lambda[u,v,succ(g,\mathcal{X}|_{\mathcal{G}(u)})]\big) \tag{33}$$

and denote the minimum of $l_g(\mathcal{X},\mathcal{A},T_v,\lambda)$ of all valid assignments $\mathcal{X}$ by $l_g^*(\mathcal{A},T_v,\lambda)$. If v is the root of T then $l_g(\mathcal{X},\mathcal{A},T_v,\lambda) = l_g(\mathcal{X},\mathcal{A},T,\lambda)$.

For each node v and $h \in \mathcal{G}(v)$, let $m_g[h,v]$ denote the minimum of $l_g(\mathcal{X},\mathcal{A},T_v,\lambda)$ over all valid assignments $\mathcal{X}$ such that $succ(g,\mathcal{X}(v)) = h$. The values $m_g[h,v]$ can be calculated dynamically as follows:

- If v is a leaf, put $m_g[h,v] = 0$ if $succ(g,\mathcal{X}(v)) = h$ and put $m_g[h,v] = \infty$ if $succ(g,\mathcal{X}(v)) \neq h$.
- Suppose that v is an internal vertex with children u_1 and u_2. If $\{g,h\} \subseteq \mathcal{G}(u_1) \cap \mathcal{G}(u_2)$ then consider all possible successors h_1, h_2 of g in $\mathcal{X}(u_1)$ and $\mathcal{X}(u_2)$ respectively. For each possible choice of successors we calculate the value $l_g(\mathcal{X},\mathcal{A},T_v,\lambda)$ given that $l_g(\mathcal{X},\mathcal{A},T_{u_1},\lambda) = m_g[h_1,u_1]$ and $l_g(\mathcal{X},\mathcal{A},T_{u_2},\lambda) = m_g[h_1,u_2]$. The minimum value is then assigned to $m_g[h,v]$. The same principle applies for the cases when $g \notin \mathcal{G}(u_1)\cap\mathcal{G}(u_2)$ or $h \notin \mathcal{G}(u_1)\cap\mathcal{G}(u_2)$.

In order to determine frequencies later on we construct a table M_g to store the optimal choices of successors for every node v and gene $h \in \mathcal{G}(v)$. If h_1 and h_2 are the choices of successor used to calculate $m_g[v,g]$ we put $M_g[u_1,v,h] = h_1$ and $M_g[u_2,v,h] = h_2$.

We evaluate $m_g[h, v]$ for all nodes $v \in V(T)$ and all $h \in \mathcal{G}(v)$. If v is the root, then we have

$$l_g^*(\mathcal{A}, T_v, \lambda) = \min\{m_g[h, v] : h \in \mathcal{G}(v)\} \ . \tag{34}$$

The calculation at each node can be greatly improved by employing successor profiles as in section 2.4. The successor table is constructed only for the genomes (actually the gene sets) of the children of v. We use weight function $w[\mathcal{X}(u_i), h_i] = m_g[u_i, h_i]$ if h_i is a gene in both genomes, and $w[\mathcal{X}(u_i), h_i] = m_g[u_i, h_i] + \lambda[u_i, v, h_i]$ if h_i is a gene in only one genome. The remaining cases are dealt with similarly. Using these techniques, we can calculate $m_g[v, h]$ for all $h \in \mathcal{G}(v)$ in $O(n)$ time (observing that the degree of T is bounded).

3.5 Subgradient Optimization with the Breakpoint Phylogeny Problem

Once again we use subgradient optimization to try and maximize the lower bound. Suppose that, for each gene g, $\mathcal{X}_g$ is the assignment chosen to optimize $l_g(\mathcal{X}_g, \mathcal{A}, T, \lambda)$. For each edge $(u, v) \in E(T)$ and each gene $h \in \mathcal{G}(u)$ we compute the frequency

$$f[u, v, h] = |\{g : succ(g, \mathcal{X}_g(v)|_{\mathcal{G}(u)})\}| \ . \tag{35}$$

The frequency values can be extracted recursively from the table M_g constructed during the calculation of m_g. A subgradient vector for λ is then given by $\Delta[u, v, h] = f[u, v, h] - 1$. The line search algorithm is then almost exactly identical to Algorithm 1 so will be omitted.

4 Experimental Results

4.1 Gene Order Data

The gene order data used in our experimental studies comes from a collection analysed in [16], and available in the GOBASE organelle genome database [9]. We selected 10 protists that represent the major eukaryote groupings.

4.2 The Breakpoint Median Bound

The first question to address when applying the local optimum bound is the choice of parameters τ, α, and the number of iterations. To this end we selected three genomes, *Reclinomonas Americana, Acanthamoeba Castellanii*, and *Cafeteria Roenbergensis* from among the larger, early branching protists. Each genome has about 70 genes that are shared by one of the other genomes. We applied the lower bound using a large number of different values for τ and α, each time determining the number of iterations by the time taken for the search to converge.

Our first observation is that a reasonable choice of τ is quite critical to the success of the bound. If α is large, an improved bound is achieved but convergence is slower. For these genomes, we found that the bound performed

well with $\tau = 0.002$ and $\alpha = 0.99$, in which case the bound converged after about 1000 iterations (taking about one second computing time on my Sparc Ultra 5). These were the parameters we used for later investigations.

We applied the median lower bound, with the above parameters, to all 120 triples of genomes from out collection of 10. Upper bounds were calculated using a fast new heuristic based on those described in [16].

The results are presented in table 1.

Table 1. Experimental results for the breakpoint median problem. $U(\mathcal{A})$ is the upper bound, $LOB(\mathcal{A})$ is the lower bound, $LOB(\mathcal{A}, \lambda)$ is the lower bound after Lagrangian optimization

	Min.	Max.	Avg.
$U(\mathcal{A}) - LOB(\mathcal{A})$	0.1702	0.6255	0.3929
$LOB(\mathcal{A})/U(\mathcal{A})\%$	60.17	89.71	73.03
$U(\mathcal{A}) - LOB(\mathcal{A}, \lambda)$	0.0292	0.4411	0.1871
$LOB(\mathcal{A}, \lambda)/U(\mathcal{A})\%$	71.22%	98.23%	87.04%
$LOB(\mathcal{A}, \lambda)/LOB(\mathcal{A})\%$	108.82%	130.11%	119.39%

From the table we observe that the improvement from Lagrange multipliers, even with such a crude line search, is still significant: an improvement of 20% on average. With the improved bound, we are on average achieving lower bounds that are better than 15% of optimal.

4.3 Applying the Breakpoint Phylogeny Bound

The breakpoint phylogeny bound is currently in the process of implementation, so experimental data is at present unavailable. Due to the similarities of the two bounds we would expect the performance to be similar, with more potential for improvement from Lagrangian optimisation in the phylogeny case.

5 Conclusions

We have described and implemented a bound for the median breakpoint problem that achieves bounds that are quite close to the calculated upper bounds. Nevertheless a gap of 15% between upper and lower bounds is still considerably worse than what can be typically obtained for the TSP. This may simply be a consequence of a more complicated combinatorial optimization. Alternatively, the gap could well be explained by the shortcomings of the simple line search, or by the lack of local optimization in the heuristic algorithms of [16]. Improving both strategies will be a focus of future research.

We demonstrate how the bound can be extended to calculate a lower bound for the breakpoint phylogeny and how Lagrange multipliers can be applied in this case. Though experimental evidence is not presently available, the similarity

between the two methods would suggest that the bound would has comparable performance.

References

1. Blanchette, M., Kunisawa, T. and Sankoff, D. 1999. Gene order breakpoint evidence in animal mitochondrial phylogeny. *Journal of Molecular Evolution* 49, 193-203.
2. Bryant, D., Deneault, M. and Sankoff, D. 1999. The breakpoint median problem. Centre de recherches mathématiques, Université de Montréal. ms.
3. Caprara, A. 1999. Formulations and hardness of multiple sorting by reversals. In: Istrail, S., Pevzner, P.A. and Waterman, M. (eds) *Proceedings of the 3rd Annual International Conference on Computational Molecular Biology (RECOMB 99)*. ACM, New York, pp 84-93.
4. Hannenhalli, S., Chappey, C., Koonin, E.V. and Pevzner, P.A. 1995. Genome sequence comparison and scenarios for gene rearrangements: a test case. *Genomics* 30, 299-311.
5. Hannenhalli, S. and Pevzner, P.A. 1995. Transforming cabbage into turnip. (polynomial algorithm for sorting signed permutations by reversals). In: *Proceedings of the 27th Annual ACM-SIAM Symposium on the Theory of Computing*, pp 178-189.
6. Held, M. and Karp, R. 1970. The traveling salesman problem and minimum spanning trees: part II. *Math. Programming* 1, 16–25.
7. Held, M. and Karp, R. 1970. The traveling salesman problem and minimum spanning trees. *Operations Res.* 18, 1138–1162.
8. Karp, R.M. Reducibility among combinatorial problems. In: Miller, R.E., and Thatcher, J. (eds) *Complexity of Computer Computations*, Plenum Press, New York, 85–103.
9. Korab-Laskowska, M., Rioux, P., Brossard, N., Littlejohn, T.G., Gray, M.W., Lang, B.F. and Burger, G. 1998. The Organelle Genome Database Project (GOBASE). *Nucleic Acids Research* 26, 139-146. `http://megasun.bch.umontreal.ca/gobase/gobase.html`
10. Pe'er, I. and Shamir, R. 1998. The median problems for breakpoints are NP-complete. Electronic Colloquium on Computational Complexity Technical Report 98-071, `http://www.eccc.uni-trier.de/eccc`
11. Reinelt, G. 1991. *The traveling salesman - computational solutions for TSP applications*. Springer Verlag.
12. Sankoff, D., Rousseau, P. 1975. Locating the vertices of a Steiner tree in an arbitrary metric space. *Math. Programming* 9(2), 240–246.
13. Sankoff, D. 1992. Edit distance for genome comparison based on non-local operations. In: Apostolico, A., Crochemore, M., Galil, Z. and Manber, U. (eds) *Combinatorial Pattern Matching. 3rd Annual Symposium. Lecture Notes in Computer Science* 644. Springer Verlag, New York, pp 121-135.
14. Sankoff, D., Sundaram, G. and Kececioglu, J. 1996. Steiner points in the space of genome rearrangements. *International Journal of the Foundations of Computer Science* 7, 1-9.
15. Sankoff, D., Blanchette, M. 1998. Multiple genome rearrangement and breakpoint phylogeny. *Journal of Computational Biology* 5, 555-570.
16. Sankoff, D., Bryant, D., Denault, M., Lang, B.F., Burger, G. 2000. Early eukaryote evolution based on mitochondrial gene order breakpoints. In: Istrail, S., Pevzner, P.A. and Waterman, M. (eds) *Proceedings of the 5th Annual International Conference on Computational Molecular Biology (RECOMB 00)*. ACM, New York (to appear).

Structural Properties and Tractability Results for Linear Synteny

David Liben-Nowell* and Jon Kleinberg**

Department of Computer Science
Cornell University, Ithaca, NY 14853, USA
{dln,kleinber}@cs.cornell.edu

Abstract. The syntenic distance between two species is the minimum number of fusions, fissions, and translocations required to transform one genome into the other. The linear syntenic distance, a restricted form of this model, has been shown to be close to the syntenic distance. Both models are computationally difficult to compute and have resisted efficient approximation algorithms with non-trivial performance guarantees. In this paper, we prove that many useful properties of syntenic distance carry over to linear syntenic distance. We also give a reduction from the general linear synteny problem to the question of whether a given instance can be solved using the maximum possible number of translocations. Our main contribution is an algorithm exactly computing linear syntenic distance in nested instances of the problem. This is the first polynomial time algorithm exactly solving linear synteny for a non-trivial class of instances. It is based on a novel connection between the syntenic distance and a scheduling problem that has been studied in the operations research literature.

1 Introduction

Numerous models for measuring the evolutionary distance between two species have been proposed in the past. These models are often based upon high-level (non-point) mutations which rearrange the order of genes within a chromosome. The distance between two genomes (or chromosomes) is defined as the minimum number of moves of a certain type required to transform the first into the second. A move for the *reversal distance* [2] is the replacement of a segment of a chromosome by the same segment in reversed order. For the *transposition distance* [3], a legal move consists of removing a segment of a chromosome and reinserting it at some other location in the chromosome.

* The author is on leave from graduate studies at MIT and is currently studying at Cambridge University. The majority of this work was performed at Cornell University. Supported in part by a Churchill Scholarship from the Winston Churchill Foundation and the ONR Young Investigator Award of the second author.

** Supported in part by a David and Lucile Packard Foundation Fellowship, an Alfred P. Sloan Research Fellowship, an ONR Young Investigator Award, and NSF Faculty Early Career Development Award CCR-9701399.

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 248–263, 2002.

In [9], Ferretti, Nadeau, and Sankoff propose a somewhat different sort of measure of genetic distance, known as *syntenic distance*. This model abstracts away from the order of the genes within chromosomes, and handles each chromosome as an unordered set of genes. The legal moves are *fusions*, in which two chromosomes join into one, *fissions*, in which one chromosome splits into two, and reciprocal *translocations*, in which two chromosomes exchange sets of genes. In practice, the order of genes within chromosomes is often unknown, and this model allows the computation of the distance between species regardless. Additional justification follows from the observation that interchromosomal evolutionary events occur with relative rarity with respect to intrachromosomal events. (For some discussion of this and related models, see [8,13].)

Work on syntenic distance was initiated by Ferretti et al. [9], who give a heuristic for approximating syntenic distance, as well as empirical evidence of its success. DasGupta, Jiang, Kannan, Li, and Sweedyk [7] show a number of results on the complexity and approximability of syntenic distance, specifically proving the problem NP-Complete and providing a simple 2-approximation. Liben-Nowell [11] proves several structural properties of the model and formally analyzes some heuristics, including that of [9].

The *linear synteny* problem is a restricted form of the general synteny problem defined by DasGupta et al. [7]. In attempting to determine the distance from genome $\mathcal{G}_1$ to genome $\mathcal{G}_2$, we consider only move sequences of the following form:

- The chromosomes of $\mathcal{G}_1$ are ordered and merged together in succession. The ith move is a fusion unless all of the genes contained in some chromosome C of $\mathcal{G}_2$ have already been merged; in this case, move i is a translocation which produces C and a chromosome containing all the other remaining merged genes.
- After all the chromosomes of $\mathcal{G}_1$ have been merged, whatever chromosomes from $\mathcal{G}_2$ remain are fissioned off one at a time.

While linear syntenic distance is largely unmotivated biologically, its relation to the syntenic distance makes it worthy of study. DasGupta et al. prove that the linear distance between two species is not much larger than the unconstrained distance: if d is the syntenic distance for any instance, then the linear distance is at most $d + \log_{4/3}(d)$.

Structural Properties of Linear Synteny. Although the additional constraints on the linear version of the problem seem to make it simpler to reason about, little work has made use of this model — possibly because many of the useful properties known for the unconstrained model were not known to carry over to the linear case.

In this paper, we prove a number of structural results for linear syntenic distance. Most are properties of the general model that were proven in [7] and [11] which we now extend to the linear case. We prove a *monotonicity property* for instances of the linear synteny problem, showing a natural ordering on problem instances. We give a method of *canonicalization* for linear move sequences: given

an arbitrary move sequence σ solving an instance, we produce another sequence σ' such that (1) σ' is no longer than σ, (2) σ' solves the instance, and (3) in σ', all fusions precede all translocations. We also prove that *duality* holds in the linear model, i.e., that the measure is indeed symmetric. These properties, coupled with the additional structure imposed by the problem definition itself, make the linear problem much easier to consider.

Solving Nested Linear Synteny. One of the most prominent features that the various measures of genomic distance share is that no efficient algorithms are known for any of them, and most have been shown to be NP-complete; see the hardness results for sorting by reversals in [4,5] and for the syntenic distance in [7]. Much of the previous work on these distances has focused on approximation algorithms with good performance guarantees: this approach has yielded performance guarantees of 3/2 for reversal distance [6] and transposition distance [3], and 2 for syntenic distance [7,9,11].

In this paper, we present the first polynomial-time algorithm to solve a nontrivial class of instances of the syntenic distance problem. For two chromosomes C_i and C_j in $\mathcal{G}_1$, let S_i and S_j be the set of chromosomes in $\mathcal{G}_2$ from which C_i and C_j draw their genes. Call an instance of synteny *nested* if for any chromosomes such that S_i and S_j are not disjoint, either $S_i \subseteq S_j$ or $S_j \subseteq S_i$.

We give a polynomial-time algorithm that solves nested instances of the linear synteny problem, by developing a connection between the syntenic distance and a scheduling problem that has been studied in the operations research literature. Specifically, the scheduling problem to which we relate syntenic distance is the following. (Precise definitions will be given in Section 6.) Imagine a company that must undertake a sequence of tasks, each with an associated profit or loss. Moreover, there is a partial order specifying dependencies among the tasks. The company's goal is to perform the tasks in the order that minimizes its maximum cumulative "debt" at any point in time. When these dependencies have a *series-parallel* structure, polynomial-time solutions were given independently by Abdel-Wahab and Kameda [1] and Monma and Sidney [12].

It is intuitively natural that genome rearrangement problems should have a connection to scheduling; in seeking an optimal rearrangement sequence, one rapidly encounters the combinatorial problem of "sequencing" certain rearrangement events as parsimoniously as possible. Our polynomial-time result provides one of the first true formalizations of this intuition, and we hope that it suggests other applications in this area for the voluminous body of work on scheduling.

2 Notational Preliminaries

The syntenic distance model is as follows: a *chromosome* is a subset of a set of n *genes*, and a *genome* is a collection of k chromosomes. A genome can be transformed by any of the following moves (for S, T, U, and V non-empty sets of genes): (1) a *fusion* $(S,T) \longrightarrow U$, where $U = S \cup T$; (2) a *fission* $S \longrightarrow (T,U)$, where $T \cup U = S$; or (3) a *translocation* $(S,T) \longrightarrow (U,V)$, where $U \cup V = S \cup T$.

The *syntenic distance* between genomes $\mathcal{G}_1$ and $\mathcal{G}_2$ is then given by the minimum number of moves required to transform $\mathcal{G}_1$ into $\mathcal{G}_2$.

The *compact representation* of an instance of synteny is described in [9] and formalized in [7]. This representation makes the goal of each instance uniform and thus eases reasoning about move sequences. For an instance in which we are attempting to transform genome $\mathcal{G}_1$ into genome $\mathcal{G}_2$, we relabel each gene a contained in a chromosome of $\mathcal{G}_1$ by the indices of the chromosomes of $\mathcal{G}_2$ in which a appears. Formally, we replace each of the k sets S in $\mathcal{G}_1$ with $\bigcup_{\ell \in S}\{i \mid \ell \in G_i\}$ (where $\mathcal{G}_2 = G_1, G_2, \ldots, G_n$) and attempt to transform these sets into the collection $\{1\}, \{2\}, \ldots, \{n\}$. As an example of the compact representation (given in [9]), consider the instance

$$\begin{array}{lll} \mathcal{G}_1 = \{x, y\}, & \text{(Chromosome 1)} & \qquad \mathcal{G}_2 = \{p, q, x\}, \quad \text{(Chromosome 1)} \\ \quad\ \{p, q, r\}, & \text{(Chromosome 2)} & \qquad\quad\ \{a, b, r, y, z\} \text{ (Chromosome 2).} \\ \quad\ \{a, b, c\} & \text{(Chromosome 3)} & \end{array}$$

The compact representation of $\mathcal{G}_1$ with respect to $\mathcal{G}_2$ is $\{1, 2\}, \{1, 2\}, \{2\}$ and the compact representation of $\mathcal{G}_2$ with respect to $\mathcal{G}_1$ is $\{1, 2\}, \{1, 2, 3\}$. For an instance of synteny in this compact notation, we will write $\mathcal{S}(n, k)$ to refer to the instance where there are n elements and k sets. Let $D(\mathcal{S}(n, k))$ be the minimum number of moves required to solve a synteny instance $\mathcal{S}(n, k)$.

We will say that two sets S_i and S_j are *connected* if $S_i \cap S_j \neq \emptyset$, and that both are in the same *component*.

The *dual* of a synteny instance $\mathcal{S}(n, k) = S_1, S_2, \ldots, S_k$ is the synteny instance $\mathcal{S}'(k, n) = S'_1, S'_2, \ldots, S'_n$, where $j \in S'_i$ iff $i \in S_j$. In [7], DasGupta et al. prove that $D(\mathcal{S}(n, k)) = D(\mathcal{S}'(k, n))$.

Formally, the *linear synteny* problem is the restricted form of the synteny problem in which we consider only move sequences of the following form:

- The first $k - 1$ moves must be fusions or severely restricted translocations, as follows. One of the input sets is initially designated as the *merging set*. Each of the first $k-1$ moves takes the current merging set Δ as input, along with one unused input set S, and produces a new merging set Δ'. If some element a appears nowhere in the genome except in Δ and S, then the move is the translocation $(\Delta, S) \longrightarrow (\Delta', \{a\})$, where $\Delta' = (\Delta \cup S) - \{a\}$. If there is no such element a, then the move simply fuses the two sets: $(\Delta, S) \longrightarrow \Delta'$, where $\Delta' = \Delta \cup S$.
- If Δ is the merging set after the $k - 1$ fusions and translocations, then each of the next $|\Delta| - 1$ moves simply fissions off a singleton $\{a\}$ and produces the new merging set $\Delta' = \Delta - \{a\}$.

Let $\widetilde{D}(\mathcal{S}(n, k))$ be the length of the optimal linear move sequence. Computing the linear syntenic distance between two genomes is also NP-hard [7]. The main theorem about linear syntenic distance is that it is not much larger than the general syntenic distance:

Theorem 1 ([7]). $\widetilde{D}(\mathcal{S}(n, k)) \leq D(\mathcal{S}(n, k)) + \log_{4/3}(D(\mathcal{S}(n, k)))$. □

Note that if a linear move sequence performs α fusions in the first $k-1$ moves, then the move sequence contains $k-\alpha-1$ translocations. After the $k-1$ fusions and translocations are complete, there are $n-k+\alpha+1$ elements left in the merging set, since exactly one element is eliminated by each translocation. Therefore, $n-k+\alpha$ fissions must be performed to eliminate the remaining elements. Thus the length of the linear move sequence is $n+\alpha-1$ moves. (Every move either is a fusion or removes one element, and all but the last element must be removed.)

A linear move sequence can be completely determined by a permutation $\pi = (\pi_1, \pi_2, \ldots, \pi_k)$ of the input sets: the sets are merged in the order given by π, and the smallest possible gene is removed whenever more than one element can be emitted. We will let $\sigma^{\pi,\mathcal{S}}$ denote the move sequence that results from using permutation π to order the input sets, and to produce elements as described above. We will use ι to denote the *identity permutation* $(1, 2, \ldots, k)$.

We will say that an instance $\mathcal{S}(n,k)$ of synteny is *linear exact* iff $\widetilde{D}(\mathcal{S}(n,k)) = \max(n,k) - 1$. An instance is linear exact iff it can be solved using translocations whenever possible, i.e., fusions and fissions are only used to make up for differences in the number of chromosomes in each genome.

Let $\mathcal{A} = A_1, A_2, \ldots, A_k$ and $\mathcal{B} = B_1, B_2, \ldots, B_k$ be two collections of sets. If, for all i, $A_i \supseteq B_i$, then we say that $\mathcal{A}$ *dominates* $\mathcal{B}$.

3 Properties of Linear Synteny

In this section, we prove a number of structural properties for linear syntenic distance. The majority of these are properties of the general model that were proven in [7] and [11] which we now extend to the linear case.

Theorem 2 (Linear Monotonicity). *Let $S_1, S_2, \ldots, S_k$ be a collection of sets that dominates the collection $T_1, T_2, \ldots, T_k$. Let $n = |\bigcup_i S_i|$ and $n' = |\bigcup_i T_i|$. If $\mathcal{S}(n,k) = S_1, S_2, \ldots, S_k$ and $\mathcal{T}(n',k) = T_1, T_2, \ldots, T_k$ are two instances of synteny, then $\widetilde{D}(\mathcal{S}(n,k)) \geq \widetilde{D}(\mathcal{T}(n',k))$.*

Proof. Let π be a permutation of $(1, 2, \ldots, k)$ such that $\sigma^{\pi,\mathcal{S}}$ is optimal. We claim that $|\sigma^{\pi,\mathcal{T}}| \leq |\sigma^{\pi,\mathcal{S}}|$. (This proves the theorem, since $\sigma^{\pi,\mathcal{T}}$ is then a linear move sequence that solves $\mathcal{T}(n',k)$ in at most $\widetilde{D}(\mathcal{S}(n,k))$ moves.) We assume that the elements are ordered such that, whenever possible, $\sigma_i^{\pi,\mathcal{S}}$ and $\sigma_i^{\pi,\mathcal{T}}$ translocate the same element.

It is sufficient to show that for any move $\sigma_i^{\pi,\mathcal{S}}$ that translocates an element a, either a is translocated in some move in $\sigma_{1\ldots i}^{\pi,\mathcal{T}}$ or a does not appear in $\mathcal{T}(n',k)$ at all. (If $\sigma^{\pi,\mathcal{S}}$ and $\sigma^{\pi,\mathcal{T}}$ do α_1 and α_2 fusions, respectively, this gives us that $\alpha_2 - \alpha_1 \leq n - n'$ since each extra fusion in $\sigma^{\pi,\mathcal{T}}$ can be charged to an element that does not appear in $\mathcal{T}(n',k)$. Then $|\sigma^{\pi,\mathcal{S}}| = n+\alpha_1-1 \geq n'+\alpha_2-1 = |\sigma^{\pi,\mathcal{T}}|$.)

Suppose not. Let ℓ be the minimum index such that: (1) $\sigma_\ell^{\pi,\mathcal{S}}$ is a translocation emitting the element a; (2) a is not translocated in $\sigma_{1\ldots\ell}^{\pi,\mathcal{T}}$; and (3) a does appear in the genome $\mathcal{T}(n',k)$.

Let χ_ℓ and χ'_ℓ be the sets of elements translocated out in $\sigma^{\pi,\mathcal{S}}_{1\ldots\ell-1}$ and $\sigma^{\pi,\mathcal{T}}_{1\ldots\ell-1}$, respectively. The current merging sets just before move ℓ are thus

$$\Delta = [S_{\pi_1} \cup S_{\pi_2} \cup \cdots \cup S_{\pi_\ell}] - \chi_\ell \qquad \Gamma = [T_{\pi_1} \cup T_{\pi_2} \cup \cdots \cup T_{\pi_\ell}] - \chi'_\ell.$$

The genes in the remaining unused input sets are

$$\overline{\Delta} = S_{\pi_{\ell+2}} \cup S_{\pi_{\ell+3}} \cup \cdots \cup S_{\pi_k} \qquad \overline{\Gamma} = T_{\pi_{\ell+2}} \cup T_{\pi_{\ell+3}} \cup \cdots \cup T_{\pi_k}.$$

Note that $\overline{\Delta} \supseteq \overline{\Gamma}$.

Since $\sigma^{\pi,\mathcal{S}}_\ell$ is a translocation emitting the element a, we know $a \in \Delta \cup S_{\pi_{\ell+1}}$ and $a \notin \overline{\Delta}$. From this and $\overline{\Delta} \supseteq \overline{\Gamma}$, we have that $a \notin \overline{\Gamma}$. So if $a \in \Gamma \cup T_{\pi_{\ell+1}}$, the move $\sigma^{\pi,\mathcal{T}}_\ell$ could emit a, but, by assumption, it does not. Then $a \notin \Gamma$, $a \notin T_{\pi_{\ell+1}}$, and $a \notin \overline{\Gamma}$. Either a was emitted earlier in $\sigma^{\pi,\mathcal{T}}$, or a does not appear anywhere in the genome $\mathcal{T}(n', k)$, contradicting our assumption. □

Lemma 3 (Merging Set Expansion). *Let $\mathcal{S}(n, k+1) = \Delta, S_1, S_2, \ldots, S_k$ be an instance of synteny in which Δ is the current merging set. Let the instance $\mathcal{T}(n, k+1) = \Delta \cup T, S_1, S_2, \ldots, S_k$ in which $\Delta \cup T$ is the merging set, for any set $T \subseteq S_1 \cup S_2 \cup \cdots \cup S_k$. Then $\widetilde{D}(\mathcal{S}(n, k+1)) = \widetilde{D}(\mathcal{T}(n, k+1))$.*

Proof. $\widetilde{D}(\mathcal{S}(n, k+1)) \leq \widetilde{D}(\mathcal{T}(n, k+1))$ by linear monotonicity. For the other direction, let π be any permutation of $(1, 2, \ldots, k+1)$ where $\pi_1 = 1$, i.e., in which Δ (or $\Delta \cup T$) is the initial merging set. We assume that the elements are ordered such that, whenever possible, $\sigma^{\pi,\mathcal{S}}_i$ and $\sigma^{\pi,\mathcal{T}}_i$ translocate the same element. We claim that $|\sigma^{\pi,\mathcal{S}}| \geq |\sigma^{\pi,\mathcal{T}}|$. This proves the lemma since $\pi_{2\ldots k+1}$ was arbitrary.

Suppose not, and let ℓ be the index of the first move in which $\sigma^{\pi',\mathcal{S}}_\ell$ produces the element a by translocation, and $\sigma^{\pi',\mathcal{S}}_\ell$ cannot produce a. Then a appears in $\Delta \cup S_{\pi_1} \cup \cdots \cup S_{\pi_\ell}$ but not in $S_{\pi_{\ell+1}} \cup \cdots \cup S_{\pi_k}$. But clearly a also appears in $\Delta \cup T \cup S_{\pi_1} \cup \cdots \cup S_{\pi_\ell}$, and still does not appear in $S_{\pi_{\ell+1}} \cup \cdots \cup S_{\pi_k}$. a cannot have been translocated earlier, because of our assumption that, whenever possible, the two move sequences produce the same element. We have chosen ℓ to be the first time this cannot be done. Thus $\sigma^{\pi',\mathcal{T}}_\ell$ can produce a, violating the assumption. □

Theorem 4 (Linear Canonicalization). *For any instance $\mathcal{S}(n, k)$ of synteny, there exists a permutation π of $(1, 2, \ldots, k)$ such that $\sigma^{\pi,\mathcal{S}}$ is optimal and has all fusions preceding all translocations.*

Proof. Let π be a permutation of $(1, 2, \ldots, k)$ such that $\sigma^{\pi,\mathcal{S}}$ is optimal and has as many initial fusions as possible. Suppose that move $\sigma^{\pi,\mathcal{S}}_i$ is the last initial fusion and $\sigma^{\pi,\mathcal{S}}_j$ is the first non-initial fusion ($j \geq i+2$). (If there is no non-initial fusion, we are done.)

Let $\pi' = (\pi_1, \ldots, \pi_{i+1}, \pi_{j+1}, \pi_{i+2}, \ldots, \pi_j, \pi_{j+2}, \ldots, \pi_k)$ be π modified so that π_{j+1} is immediately after π_{i+1}. We claim that $\sigma^{\pi',\mathcal{S}}$ is also optimal, and has

one more initial fusion than $\sigma^{\pi,\mathcal{S}}$. This violates our choice of π and proves the theorem.

First we claim that $\sigma^{\pi',\mathcal{S}}$ has $i+1$ initial fusions. Clearly, the first i moves of the two sequences are identical, since they merge exactly the same sets (and exactly the same sets remain unmerged). Thus we need only prove that $\sigma_{i+1}^{\pi',\mathcal{S}}$ is a fusion. Suppose that it were a translocation, i.e., there is some element ℓ that appears only in the sets $S_{\pi_1}, S_{\pi_2}, \ldots, S_{\pi_{i+1}}, S_{\pi_{j+1}}$. If $\ell \in S_{\pi_{j+1}}$, then the move $\sigma_j^{\pi,\mathcal{S}}$ would be a translocation, since the last occurrence of the element ℓ is in the set $S_{\pi_{j+1}}$. If $\ell \notin S_{\pi_{j+1}}$, and instead $\ell \in [S_{\pi_1} \cup S_{\pi_2} \cup \cdots \cup S_{\pi_{i+1}}]$, there would be a translocation somewhere in $\sigma_{1\ldots i}^{\pi,\mathcal{S}}$, since ℓ does not appear outside the first $i+1$ sets. Neither of these occur, so there is no such ℓ, and $\sigma_{i+1}^{\pi',\mathcal{S}}$ is a fusion.

For optimality, we claim that every translocation in $\sigma^{\pi,\mathcal{S}}$ corresponds with a translocation in $\sigma^{\pi',\mathcal{S}}$. Note that all moves after π and π' converge (after π_{j+2}) can emit exactly the same elements. For earlier moves, suppose that $\sigma_r^{\pi,\mathcal{S}}$ produces an element a by translocation. That is, a appears in $S_{\pi_1}, S_{\pi_2}, \ldots, S_{\pi_r}$ and not in $S_{\pi_{r+1}}, S_{\pi_{r+2}}, \ldots, S_{\pi_k}$. Obviously having already merged $S_{\pi_{j+1}}$ changes neither the presence of a in the current merging set nor the absence of a in the unused input sets. Thus $\sigma_{r+1}^{\pi',\mathcal{S}}$ is a translocation. □

Proposition 5. *Let $\mathcal{S}(n,k) = S_1, S_2, \ldots, S_k$ and $\mathcal{T}(n,k) = T_1, T_2, \ldots, T_k$ be two instances of synteny. If $\mathcal{S}(n,k)$ dominates $\mathcal{T}(n,k)$, then $\mathcal{S}'(k,n)$ dominates $\mathcal{T}'(k,n)$.*

Proof. Suppose not. Then there is some set i such that $S'_i \not\supseteq T'_i$. That is, there is some element $\ell \in T'_i$ but $\ell \notin S'_i$. By the definition of the dual, this means that the element $i \in T_\ell$ but $i \notin S_\ell$. This violates the assumption that $\mathcal{S}(n,k)$ dominates $\mathcal{T}(n,k)$. □

Definition 6. *For $\alpha \leq \min(n,k) - 1$, the instance $\mathcal{K}_\alpha(n,k)$ consists of the following sets:*

- *$k - \alpha$ copies of $\{1, 2, \ldots, n\}$*
- $\{1, 2, \ldots, n-1\}$
- $\{1, 2, \ldots, n-2\}$
- $\vdots$
- $\{1, 2, \ldots, n-\alpha\}$.

Note that

- $\widetilde{D}(\mathcal{K}_\alpha(n,k)) = n + k - \alpha - 2$. Merging the sets in the above order requires $k - \alpha - 1$ fusions, α translocations, and $n - \alpha - 1$ fissions, or $n + k - \alpha - 2$ moves total. There are only $\alpha + 1$ elements that appear in at most $k - 1$ sets, so this is optimal by the count bound [11].
- $\mathcal{K}'_\alpha(k,n) = \mathcal{K}_\alpha(k,n)$. (In other words, the dual of the instance $\mathcal{K}_\alpha(n,k)$ is simply the instance $\mathcal{K}_\alpha(k,n)$.) Verifying this is straightforward: the first $n-\alpha$ elements appear in all sets, $n - \alpha + 1$ appears in all but one set, etc.

Theorem 7 (Linear Duality). *For all* $\mathcal{S}(n,k)$, $\widetilde{D}(\mathcal{S}(n,k)) = \widetilde{D}(\mathcal{S}'(k,n))$.

Proof. Suppose not, i.e., suppose that $\widetilde{D}(\mathcal{S}(n,k)) < \widetilde{D}(\mathcal{S}'(k,n))$.

Relabel the sets and elements of $\mathcal{S}(n,k)$ such that the move sequence $\sigma^{\iota,\mathcal{S}}$ is optimal, canonical, and produces elements in the order $n, n-1, \ldots, 1$. Note that this relabeling does not change $\widetilde{D}(\mathcal{S}(n,k))$. Let $\widetilde{D}(\mathcal{S}(n,k)) = n+k-\alpha-2$.

Notice that the element $n-i$ does not appear outside the first $k-\alpha+i$ sets, since otherwise the $(k-\alpha+i-1)$th move could not produce element $n-i$. Therefore, we have that $\mathcal{K}_\alpha(n,k)$ dominates $\mathcal{S}(n,k)$. Thus $\mathcal{K}'_\alpha(k,n)$ dominates $\mathcal{S}'(k,n)$ by Proposition 5. Linear monotonicity, along with the fact that $\mathcal{K}'_\alpha(k,n) = \mathcal{K}_\alpha(k,n)$, then gives us

$$\widetilde{D}(\mathcal{S}(n,k)) = \widetilde{D}(\mathcal{K}_\alpha(n,k)) = \widetilde{D}(\mathcal{K}_\alpha(k,n)) = \widetilde{D}(\mathcal{K}'_\alpha(k,n)) \geq \widetilde{D}(\mathcal{S}'(k,n)).$$

This contradicts the assumption and proves the theorem. □

4 From General Linear to Exact Linear Synteny

In this section, we give a reduction from the general linear to the exact linear synteny problem, a conceptually simpler problem. We first define an augmentation to instances of synteny:

Definition 8. *For an instance* $\mathcal{S}(n,k) = S_1, S_2, \ldots, S_k$, *let*

$$\mathcal{S}^{i\oplus\delta}(n+\delta,k) = S_1, S_2, \ldots, \hat{S}_i, \ldots, S_k$$

be a new instance of synteny, where (1) $1 \leq i \leq k$, *(2)* $\hat{S}_i = S_i \cup \{a_1, a_2, \ldots, a_\delta\}$, *and (3)* $a_\ell \notin S_j$.

The intuition behind this instance is that we have augmented S_i with extra elements that will be expelled during would-be fusions. This new instance is basically the original with δ fusion "coupons" that can be used to turn fusions into translocations. For some choices of i and δ, this increases the number of elements and translocations without a corresponding increase in distance.

Theorem 9. *Let* $\mathcal{S}(n,k)$ *be an instance of synteny. Suppose* $\sigma^{\pi,\mathcal{S}}$ *is a move sequence solving* $\mathcal{S}(n,k)$ *such that* $|\sigma^{\pi,\mathcal{S}}| = n+\alpha-1$. *Then* $|\sigma^{\pi,\mathcal{S}^{\pi_1\oplus\delta}}| = n + \max(\alpha,\delta) - 1$.

Proof. There are α fusions in $\sigma^{\pi,\mathcal{S}}$. When the jth of these fusions occurs, $\sigma^{\pi,\mathcal{S}^{\pi_1\oplus\delta}}$ could emit the element a_j (since a_j does not appear in any of the unused input sets, and is in the merging set as of the first move). Every translocation in the original move sequence remains a translocation in the new sequence since we have the same remaining unused input sets at every point.

Thus we can eliminate fusions from the move sequence using a-elements, until we run out of fusions or a-elements with which to eliminate them. Thus we have $\alpha-\delta$ fusions left if there are too many fusions, and therefore $|\sigma^{\pi,\mathcal{S}^{\pi_1\oplus\delta}}| = n+\delta+\max(\alpha-\delta,0)-1 = n+\max(\alpha,\delta)-1$. □

Proposition 10. *Let $\mathcal{S}(n,k)$ be an instance of synteny. Suppose that there exists an optimal move sequence $\sigma^{\pi,\mathcal{S}}$ solving $\mathcal{S}(n,k)$ such that $\pi_1 \in \Gamma$, for some set $\Gamma \subseteq \{1,2,\ldots,k\}$. Then*

$$\left[\exists i \in \Gamma\ \widetilde{D}(\mathcal{S}^{i\oplus\delta}(n+\delta,k)) = n+\delta-1\right] \iff \widetilde{D}(\mathcal{S}(n,k)) \leq n+\delta-1.$$

Proof. Immediate from monotonicity and Theorem 9. □

5 Nested Synteny

In this section, we consider the special class of instances of synteny in which all non-disjoint sets are totally ordered by the subset relation.

Definition 11 (Nested Synteny). *An instance $\mathcal{S}(n,k) = S_1, S_2, \ldots, S_k$ of synteny is* nested *if, for all $i \neq j$, either (1) $S_i \cap S_j = \emptyset$, (2) $S_i \subseteq S_j$, or (3) $S_j \subseteq S_i$.*

In each component of a nested instance of synteny, the set containing all elements in the component is called the *root* of the component. (If there are multiple copies of this set in some component, we will identify the root as the copy with the smallest index.)

Lemma 12. *If $\mathcal{S}(n,k) = S_1, S_2, \ldots, S_k$ and $S_i \subseteq S_j$, then there exists a optimal linear move sequence in which S_j is merged before S_i.*

Proof. Suppose π is a permutation of $(1,2,\ldots,k)$ such that $\sigma^{\pi,\mathcal{S}}$ is optimal and i appears before j in π. (If there is no such π, then we are done.) Let $\pi_x = i$.

Let $\mathcal{T}(n',k'+3) = T_1, T_2, \ldots, T_{k'}, S_i, S_j, \Delta$ be the instance resulting after $\sigma^{\pi,\mathcal{S}}_{1\ldots(x-2)}$, where Δ is the current merging set. The next move $\sigma^{\pi,\mathcal{S}}_{x-1}$ would merge S_i. We have two cases for this move:

- $\sigma^{\pi,\mathcal{S}}_{x-1}$ is a fusion, $(\Delta, S_i) \longrightarrow \Delta \cup S_i$.
  This produces the sets $T_1, T_2, \ldots, T_{k'}, S_j, \Delta \cup S_i$. This collection of sets is no easier to solve than $T_1, T_2, \ldots, T_{k'}, S_j, \Delta \cup S_j$, by Lemma 3, since $\Delta \cup S_j \supseteq \Delta \cup S_i$. By monotonicity, this instance is no easier to solve than $T_1, T_2, \ldots, T_{k'}, S_i, \Delta \cup S_j$, since $S_i \subseteq S_j$.
  If we instead complete the move $(\Delta, S_j) \longrightarrow \Delta \cup S_j$, then the sets remaining after this operation are $T_1, T_2, \ldots, T_{k'}, S_i, \Delta \cup S_j$. By the above, this is no harder to solve than the result of doing $\sigma^{\pi,\mathcal{S}}_{x-1}$, so we make this move instead.
- $\sigma^{\pi,\mathcal{S}}_{x-1}$ is a translocation, $(\Delta, S_i) \longrightarrow (\Delta \cup S_i - \{a\}, \{a\})$ for some $a \in \Delta \cup S_i$ and $a \notin T_1 \cup T_2 \cup \ldots \cup T_{k'} \cup S_j$.
  This produces the sets $T_1, T_2, \ldots, T_{k'}, S_j, \Delta \cup S_i - \{a\}$. By successively applying Lemma 3 and Theorem 2 (as in the fusion case), we have that this is no easier to solve than the instance consisting of $T_1, T_2, \ldots, T_{k'}, S_i, \Delta \cup S_j - \{a\}$. If instead of $\sigma^{\pi,\mathcal{S}}_{x-1}$, we merge S_j instead, we can still complete a translocation: $a \in \Delta \cup S_j \supseteq \Delta \cup S_i$, and $a \notin S_i \subseteq S_j$, so this move is $(\Delta, S_j) \longrightarrow (\Delta \cup S_j - \{a\}, \{a\})$. By the above, this is no harder to solve than the result of doing $\sigma^{\pi,\mathcal{S}}_{x-1}$, so we might as well make this move instead. □

Corollary 13. *For any instance $\mathcal{S}(n,k)$ of synteny, there exists an optimal move sequence $\sigma^{\pi,\mathcal{S}}$ solving the instance such that for all $i<j$, $S_{\pi_i} \not\subset S_{\pi_j}$.* □

Corollary 14. *If $\mathcal{S}(n,k)$ is a nested synteny instance with roots $R_1, R_2, \ldots, R_p$, then there exists an optimal move sequence $\sigma^{\pi,\mathcal{S}}$ solving $\mathcal{S}(n,k)$ such that $\pi_1 \in \{R_1, R_2, \ldots, R_p\}$.* □

Note that if $\mathcal{S}(n,k)$ is nested, then so is $\mathcal{S}^{R_i \oplus \delta}(n+\delta,k)$ since we are only adding extra elements to the root of some component.

6 The Minimum Loan Problem

Definition 15. *Let $T = T_1, T_2, \ldots, T_n$ be a set of* tasks. *Let $p : T \longrightarrow \mathbb{Z}$ be a function giving the* profit *of each task. Then, for π a permutation of $(1,2,\ldots,n)$, let*

$$P_\pi(i) \stackrel{def}{=} \sum_{j=1}^{i} p(T_{\pi_j})$$

be the cumulative profit *of the first i tasks under π.*

Note that if tasks have negative profits (i.e., costs), then the cumulative profit can also be negative. We will say that a permutation π *respects* a partial order $\prec$ on T if, for all $i<j$, $T_{\pi_j} \not\prec T_{\pi_i}$. This gives rise to the following scheduling problem:

Definition 16 (Min Loan). *Let $T = T_1, T_2, \ldots, T_n$ be a set of tasks. Let $\prec$ be a partial order on T defining a precedence relation among the tasks. Let $p : T \longrightarrow \mathbb{Z}$ be a function giving the profit of each task. Then $(T, \prec, p)$ is an instance of the minimum loan problem: find*

$$\max_{\pi} \min_{0 \le i \le n} P_\pi(i)$$

for π respecting $\prec$.

(In [1], Abdel-Wahab and Kameda define this problem in terms of the cumulative *cost* of the tasks rather than the cumulative profit.) Notice that for any permutation π, we have $P_\pi(0) = 0$, so the optimum value for any instance of the Min Loan problem is always at most 0.

The intuition for this problem is the following: suppose that there is a company with a set of jobs that it has chosen to undertake. Each job will result in either a profit or a loss. The jobs are also limited to follow some precedence constraints, e.g., the engines must be built before the cars can be assembled. The *minimum loan* is the minimum amount of initial funding necessary to be able complete all of the jobs without ever running out of money. (Alternatively, this is the maximum amount of debt for the company at the worst financial moment.)

The Min Loan problem is NP-Complete in general [10, p. 238], but Abdel-Wahab and Kameda [1] give an $O(n^2)$ algorithm when $\prec$ is *series-parallel*. Monma and Sidney [12] independently give a polynomial-time algorithm for this case as well. A partial order is series-parallel when its associated DAG is a series-parallel graph, according to the following rules: (1) a graph with two nodes with an edge from one to the other is series-parallel; (2) if G is series-parallel, then so is the graph that results from adding a node to the middle of any edge in G; and (3) if G is series-parallel, then so is the graph that results from duplicating any edge in G.

We will not go into the details of the algorithms of Abdel-Wahab and Kameda or Monma and Sidney here; rather, we will use them in a "black-box" fashion in our approach to the nested linear synteny problem in the following section.

7 Minimum Loans and Exact Linear Synteny

Let $\mathcal{S}(n,k) = S_1, S_2, \ldots, S_k$ be a nested instance of synteny with $n \geq k$ and with p components. Let the roots of the components be $S_{R_1}, S_{R_2}, \ldots, S_{R_p}$. We will make use of the Min Loan algorithms of Abdel-Wahab and Kameda [1] and Monma and Sidney [12] to determine $\widetilde{D}(\mathcal{S}(n,k))$.

Let $T = T_1, T_2, \ldots, T_k$ be a set of tasks where T_j denotes the merging of the set S_j with the current merging set. Let $\prec$ be the smallest relation such that (1) $T_j \prec T_\ell$ if $S_j \supset S_\ell$ and (2) $T_j \prec T_\ell$ if $S_j = S_\ell$ and $j \leq \ell$.

Definition 17. *For a component q, the* q-profit *of a task T_i is the following:*

$$p^q(T_i) \stackrel{def}{=} \begin{cases} \left| S_i - \bigcup_{T_j \succ T_i} S_j \right| - 1 & \text{if } i \neq R_q \\ \left| S_i - \bigcup_{T_j \succ T_i} S_j \right| & \text{if } i = R_q. \end{cases}$$

Let $P^q_\pi(i) = \sum_{j=1}^{i} p^q(T_{\pi_j})$ be the *cumulative q-profit* of the first i steps under permutation π.

Definition 18. *For a permutation π of $(1, 2, \ldots, k)$ let:*

1. *$x_\pi(i)$ be 1 iff $\sigma_i^{\pi,\mathcal{S}}$ is a translocation.*
2. *$\chi_\pi(i)$ be the set of elements emitted by translocation during $\sigma_{1\ldots i}^{\pi,\mathcal{S}}$.*

Proposition 19. *For all π and i, $|\chi_\pi(i)| \leq i$.*

Proof. Immediate from the fact that we can translocate at most one element per move. □

Proposition 20. *Let $\mathcal{S}(n,k)$ be a nested instance of synteny, and let π be a permutation of $(1, 2, \ldots, k)$ that respects $\prec$. Then*

$$\left| S_{\pi_i} - \bigcup_{T_{\pi_j} \succ T_{\pi_i}} S_{\pi_j} \right| = \left| S_{\pi_i} - \bigcup_{j>i} S_{\pi_j} \right|.$$

Proof. If $T_{\pi_j} \succ T_{\pi_i}$, then $j > i$ since π respects $\prec$. If $j > i$ but $T_{\pi_j} \not\succ T_{\pi_i}$, then $S_{\pi_i} \cap S_{\pi_j} = \emptyset$ since $\mathcal{S}(n,k)$ is nested. □

Proposition 21. *For some component q, there is an optimal linear move sequence that uses S_{R_q} is its initial merging set and respects $\prec$.*

Proof. Let $\sigma^{\pi,\mathcal{S}}$ be an optimal move sequence respecting the subset precedence (by Theorem 13), and let q be the component of S_{π_1}. $S_{\pi_1} = S_{R_q}$ since all other sets in that component are subsets of that one. We can make this sequence respect clause (2) of the definition of $\prec$ trivially by using identical sets in increasing order of index. □

Lemma 22. *Let $\mathcal{S}(n,k)$ be a linear exact nested instance of synteny with $n \geq k$. Let π be a permutation of $(1,2,\ldots,k)$ such that $\sigma^{\pi,\mathcal{S}}$ is optimal, $\pi_1 = R_q$, and π respects $\prec$. Then, for all $0 \leq i \leq k$,*

$$P_\pi^q(i) = \left| \bigcup_{j \leq i} S_{\pi_j} - \bigcup_{j > i} S_{\pi_j} \right| - |\chi_\pi(i-1)|\,.$$

Proof (by induction on i).

- $[i = 0]$. Then $P_\pi^q(i) = \sum_{j=1}^{0} p^q(T_{\pi_j}) = 0$ and $\bigcup_{j \leq 0} S_{\pi_j} = \emptyset = \chi_\pi(-1)$.
- $[i = 1]$. Then

$$P_\pi^q(1) = \sum_{j=1}^{1} p^q(T_{\pi_j}) = p^q(T_{\pi_1}) = \left| S_{R_q} - \bigcup_{T_j \succ T_{\pi_1}} S_j \right| = \left| \bigcup_{j \leq 1} S_{\pi_j} - \bigcup_{j > 1} S_{\pi_j} \right|$$

  by Proposition 20. Since $\chi_\pi(0) = \emptyset$,

$$P_\pi^q(i) \quad = \quad \left| \bigcup_{j \leq i} S_{\pi_j} - \bigcup_{j > i} S_{\pi_j} \right| - |\chi_\pi(i-1)|.$$

- $[i \geq 2]$. Then

$$P_\pi^q(i) = \sum_{j=1}^{i} p^q(T_{\pi_j}) = p^q(T_{\pi_i}) + \sum_{j=1}^{i-1} p^q(T_{\pi_j}) = p^q(T_{\pi_i}) + P_\pi^q(i-1)$$

  by the definition of P^q. Applying the induction hypothesis and the definition of p^q (since $\pi_i \neq R_q$), we have

$$P_\pi^q(i) = \left| S_{\pi_i} - \bigcup_{T_{\pi_j} \succ T_{\pi_i}} S_{\pi_j} \right| - 1 + \left| \bigcup_{j \leq i-1} S_{\pi_j} - \bigcup_{j > i-1} S_{\pi_j} \right| - |\chi_\pi(i-2)|\,.$$

Since $\sigma^{\pi,\mathcal{S}}$ is optimal, and $\mathcal{S}(n,k)$ is linear exact, and $n \geq k$, we know that the move $\sigma^{\pi,\mathcal{S}}_{i-1}$ is a translocation. Furthermore, the element translocated in this move cannot have been previously translocated, so $|\chi_\pi(i-1)| = 1+|\chi_\pi(i-2)|$. Therefore,

$$P^q_\pi(i) = \left| S_{\pi_i} - \bigcup_{T_{\pi_j} \succ T_{\pi_i}} S_{\pi_j} \right| + \left| \bigcup_{j \leq i-1} S_{\pi_j} - \bigcup_{j > i-1} S_{\pi_j} \right| - |\chi_\pi(i-1)|\,.$$

By Proposition 20,

$$P^q_\pi(i) = \left| S_{\pi_i} - \bigcup_{j > i} S_{\pi_j} \right| + \left| \bigcup_{j \leq i-1} S_{\pi_j} - \bigcup_{j > i-1} S_{\pi_j} \right| - |\chi_\pi(i-1)|\,.$$

The first two terms in this sum are simply those elements that appear in S_{π_i} and never after, and those elements that appear in $S_{\pi_1}, S_{\pi_2}, \ldots, S_{\pi_{i-1}}$ and never after. We can simply combine these terms since these two sets are disjoint, and we end up with

$$P^q_\pi(i) = \left| \bigcup_{j \leq i} S_{\pi_j} - \bigcup_{j > i} S_{\pi_j} \right| - |\chi_\pi(i-1)|\,.$$

□

Corollary 23. *If $\mathcal{S}(n,k)$ is nested, linear exact, and has $n \geq k$, then there exists a q such that* $\mathsf{minLoan}(T, \prec, p^q) = 0$.

Proof. Let $\sigma^{\pi,\mathcal{S}}$ be an optimal move sequence solving $\mathcal{S}(n,k) = S_1, S_2, \ldots, S_k$ such that $\pi_1 = R_q$ and π respects $\prec$. (One exists by Proposition 21.) For all i, $\chi_\pi(i-1)$ is a subset of $\bigcup_{j \leq i} S_{\pi_j} - \bigcup_{j > i} S_{\pi_j}$, since only elements that do not appear in the remainder of the genome can be translocated. Thus $P^q_\pi(i) \geq 0$ for all i, and $P^q_\pi(0) = 0$. □

Proposition 24. *If π is a permutation of $(1, 2, \ldots, k)$ respecting $\prec$ such that $\min_i P^q_\pi(i) = 0$ then there exists a permutation π' with $\pi'_1 = R_q$ such that $\min_i P^q_{\pi'}(i) = 0$ and π' respects $\prec$.*

Proof. Let π' be π with R_q moved to the front. From the fact that $p^q(T_{R_q}) \geq 0$, this move does not make $\min_i P^q_{\pi'}(i)$ worse. Since $T_j \not\prec T_{R_q}$ for all j, π' respects $\prec$ if π did. □

Proposition 25. *For all π and $1 \leq i \leq k-1$,*

$$x_\pi(i) = 1 \quad \Longleftrightarrow \quad \left| \bigcup_{j \leq i+1} S_{\pi_j} - \bigcup_{j > i+1} S_{\pi_j} \right| - |\chi_\pi(i-1)| \quad \geq \quad 1.$$

Proof. Notice that move $\sigma_i^{\pi,\mathcal{S}}$ takes the set $S_{\pi_{i+1}}$ and the previous merging set $\bigcup_{j<i+1} S_{\pi_j} - \chi_\pi(i-1)$, all previously merged elements less those that have already been emitted by translocation.

$$\begin{aligned}
x_\pi(i) = 1 &\iff \exists a \left[a \in S_{\pi_{i+1}} \cup \left(\bigcup_{j \le i} S_{\pi_j} - \chi_\pi(i-1) \right) \text{ and } a \notin \bigcup_{j>i+1} S_{\pi_j} \right] \\
&\iff \exists a \left[a \in \left(\bigcup_{j \le i+1} S_{\pi_j} - \chi_\pi(i-1) \right) - \bigcup_{j>i+1} S_{\pi_j} \right] \\
&\iff \left| \bigcup_{j \le i+1} S_{\pi_j} - \chi_\pi(i-1) - \bigcup_{j>i+1} S_{\pi_j} \right| \quad \ge \quad 1 \\
&\iff \left| \bigcup_{j \le i+1} S_{\pi_j} - \bigcup_{j>i+1} S_{\pi_j} \right| - |\chi_\pi(i-1)| \quad \ge \quad 1
\end{aligned}$$

since $\chi_\pi(i-1)$ and $\bigcup_{j>i+1} S_{\pi_j}$ are disjoint and $\chi_\pi(i-1) \subseteq \bigcup_{j \le i+1} S_{\pi_j}$. □

Lemma 26. *Let $\mathcal{S}(n,k)$ be a nested instance of synteny such that $n \ge k$. If $\mathsf{minLoan}(T, \prec, p^q) = 0$, then $\mathcal{S}(n,k)$ is linear exact.*

Proof. Let π be a permutation of $(1, 2, \ldots, k)$ respecting $\prec$ and with $\pi_1 = R_q$ such that $\min_i P_\pi^q(i) = 0$. (One exists by Proposition 24.) We claim that $\sigma^{\pi,\mathcal{S}}$ starts with $k-1$ translocations, which proves the theorem. To do this, it is sufficient to show that $x_\pi(i) = 1$ for all $1 \le i \le k-1$.

Consider an arbitrary i. We know that $0 \le P_\pi^q(i+1) = \sum_{\ell=1}^{i+1} p^q(T_{\pi_\ell})$. By the definition of p^q and the fact that $\pi_1 = R_q$, we know that

$$P_\pi^q(i+1) \quad = \quad \left| S_{R_q} - \bigcup_{T_{\pi_j} \succ T_{R_q}} S_{\pi_j} \right| + \sum_{\ell=2}^{i+1} \left(\left| S_{\pi_\ell} - \bigcup_{T_{\pi_j} \succ T_{\pi_\ell}} S_{\pi_j} \right| - 1 \right) \quad \ge \quad 0.$$

Rearranging, we have

$$-i + \sum_{\ell=1}^{i+1} \left| S_{\pi_\ell} - \bigcup_{T_{\pi_j} \succ T_{\pi_\ell}} S_{\pi_j} \right| \quad \ge \quad 0.$$

By Proposition 19, $|\chi_\pi(i-1)| \le i-1$. Applying Proposition 20 and this fact, we have

$$-1 - |\chi_\pi(i-1)| + \sum_{\ell=1}^{i+1} \left| S_{\pi_\ell} - \bigcup_{j>\ell} S_{\pi_j} \right| \quad \ge \quad 0.$$

The sets in the sum are simply the sets of all elements that appear for the last time in S_{π_ℓ}. These sets are disjoint, and can be rewritten as simply

$$\left| \bigcup_{j \le i+1} S_{\pi_j} - \bigcup_{j > i+1} S_{\pi_j} \right| - |\chi_\pi(i-1)| - 1 \quad \ge \quad 0.$$

which by Lemma 25 gives us that $x_\pi(i) = 1$. □

Theorem 27. *Let $\mathcal{S}(n,k)$ be a nested instance of synteny with $n \ge k$. Then $\mathcal{S}(n,k)$ is linear exact iff, for some q,* $\mathsf{minLoan}(T, \prec, p^q) = 0$.

Proof. Immediate from Corollary 23 and Lemma 26. □

Putting It Together

If we are given a nested instance of synteny $\mathcal{S}(n,k) = S_1, S_2, \ldots, S_k$ with p components with roots $R_1, R_2, \ldots, R_p$, we can determine $\widetilde{D}(\mathcal{S}(n,k))$ as follows:

1. Let $\delta = \max(k-n, 0)$.
2. Let T_i be the event of merging the set S_i into the current merging set.
3. Let $\prec$ be the relation so that $S_j \prec S_\ell$ iff $S_j \supset S_\ell$, or $S_j = S_\ell$ and $j \le \ell$.
4. Let $p(T_j) = \left| \bigcup_{T_\ell \prec T_j} S_\ell - \bigcup_{T_\ell \succ T_j} S_\ell \right|$.
5. For each $1 \le q \le p$: binary search for the minimum x_q such that the instance $\mathcal{S}^{R_q \oplus \delta + x_q}(n + \delta + x_q, k)$ is linear exact, by the following procedure. (Note that this instance too is nested.)
   (a) Let $p^q(T_j) = p(T_j) - 1$ for $j \ne R_q$, and let $p^q(T_{R_q}) = p(T_{R_q}) + \delta + x_q$.
   (b) Return true if $\mathsf{minLoan}(T, \prec, p^q) = 0$.
6. Return $\widetilde{D}(\mathcal{S}(n,k)) = n + \delta + \min_q(x_q) - 1$.

Correctness follows from Proposition 10, Corollary 14, and Theorem 27. The running time of this algorithm is $O(pk^2 \log k + nk^2)$: the binary search requires at most $\log k$ iterations because we know that $x_q \le k$; the $\mathsf{minLoan}$ call requires $O(k^2)$ time since there are k events in question, and we must run this search for each of the p components of the instance. Computing $\prec$ and $p(T_j)$ requires $O(nk^2)$ time.

By linear duality, we can also use the above algorithm to compute the linear syntenic distance of an *anti-nested* instance, i.e., one whose dual is nested.

References

1. H. M. Abdel-Wahab and T. Kameda. Scheduling to minimize maximum cumulative costs subject to series-parallel precedence constraints. *Operations Research*, 26(1):141–158, January/February 1978.
2. Vineet Bafna and Pavel A. Pevzner. Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, 25(2):272–289, April 1996. A previous version appeared in FOCS'93.

3. Vineet Bafna and Pavel A. Pevzner. Sorting by transpositions. *SIAM Journal on Discrete Mathematics*, 11(2):224–240, 1998. A previous version appeared in SODA'95.
4. Piotr Berman and Marek Karpinski. On some tighter inapproximability results. *Electronic Colloquium on Computational Complexity*, Report No. 29, 1998.
5. Alberto Caprara. Sorting by reversals is difficult. In *1st Annual International Conference on Computational Molecular Biology*, pages 75–83, 1997.
6. D. A. Christie. A 3/2-approximation algorithm for sorting by reversals. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 244–252, 1998.
7. Bhaskar DasGupta, Tao Jiang, Sampath Kannan, Ming Li, and Elizabeth Sweedyk. On the complexity and approximation of syntenic distance. *Discrete Applied Mathematics (special issue on computational biology)*, 88(1–3):59–82, November 1998. A previous version appeared in RECOMB'97.
8. Jason Ehrlich, David Sankoff, and Joseph H. Nadeau. Synteny conservation and chromosome rearrangements during mammalian evolution. *Genetics*, 147(1):289–296, September 1997.
9. Vincent Ferretti, Joseph H. Nadeau, and David Sankoff. Original synteny. In *7th Annual Symposium on Combinatorial Pattern Matching*, pages 159–167, 1996.
10. Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.
11. David Liben-Nowell. On the structure of syntenic distance. In *10th Annual Symposium on Combinatorial Pattern Matching*, pages 43–56, 1999.
12. C. L. Monma and J. B. Sidney. A general algorithm for optimal job sequencing with series-parallel precedence constraints. Technical Report 347, School of Operations Research, Cornell University, 1977.
13. David Sankoff and Joseph H. Nadeau. Conserved synteny as a measure of genomic distance. *Discrete Applied Mathematics (special issue on computational biology)*, 71(1–3):247–257, December 1996.

Shift Error Detection in Standardized Exams
(Extended Abstract)

Steven Skiena* and Pavel Sumazin**

State University of New York at Stony Brook, Stony Brook NY 11794-4400
{skiena,psumazin}@cs.sunysb.edu

1 Introduction

Computer-graded multiple choice examinations are a familiar and dreaded part of most student's lives. Many test takers are particularly fearful of form-filling *shift* errors, where absent-mindedly marking the answer to (say) question 32 in position 31 causes a long run of answers to be successively displaced. Test-taking strategies where students answer questions out of sequence (such as answering easy questions first) seem particularly likely to cause unrecognized shift errors. Such errors can result in many correct answers being marked wrong, and significantly lower the exam taker's score.

In this paper, we consider the question of whether these shift errors can be accurately recognized and corrected for. Our results suggest that students are right to fear such errors, and that a non-trivial fraction of multiple-choice exams appear to contain significant shift errors. In particular, we found evidence that 1%-2% of multiple-choice exam papers at SUNY Stony Brook contain likely shift errors, each causing the loss of about 10% of the student's grade. Given the importance of standardized examinations, clerical mistakes should not be allowed to have such an impact on the student's score. If our observed shift error rate holds across the millions of examinations given annually, this is a serious but yet unstudied problem. Indeed, based on the preliminary results in the paper, we have begun working with The College Board to study the prevalence of uncorrected shift-errors in SAT examinations.

The problem of detecting shift errors proves to be surprisingly subtle, with such factors as (1) the quality of the student, (2) the difficulty of the test, (3) incorrect shifted answers, (4) random guessing by examinee, and (5) the pattern of answers in the key impacting our assessment of whether the scoring improvement of a putative shift is significant. However, in this paper we introduce algorithms to detect and correct for a substantial fraction of such shift errors with very low false detection probability. Our experiments show that we reliably detect and correct almost all shifts that cause a loss of at least three answers, and most of the shifts that cause a loss of at least two answers. We believe that our methods should be employed within the standardized testing industry.

* Corresponding author. 1-631-632-9026 (phone) and 1-631-632-8334 (fax). This work is partially supported by NSF Grant CCR-9625669 and ONR Award N00149710589.
** Partially supported by the 1998 GAAN graduate fellowship P200A97030199.

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 264–276, 2002.

Our paper is organized as follows. Previous work and notation are discussed in Section 1. In Section 2, we introduce three different algorithmic approaches based on dynamic programming, discrepancy theory, and proper patch scoring. The fundamental problems of recognizing portions of an exam with suspiciously poor student performance, and properly interpreting the significance of the score change in shifting a region are discussed in Sections 3 and 4 respectively. Our experimental results are reported in Section 5. Finally, in Section 6, we consider an interesting combinatorial problem on strings, which gives insight into possible adversary strategies for test takers. Suppose multiple-choice examinations were graded to compensate as generously as possible for potential shift errors, by awarding as the score the length of the longest-common subsequence between the student's paper and the correct answers. How should a clever but illiterate student answer so as to maximize her expected score? Through computer searches, we provide new bounds in the quest for what Dancik [8] terms the most *adaptable* sequence.

1.1 The Standardized Testing Industry and Related Work

Standardized testing is a large and growing industry. In the 1997-1998 academic year, the Educational Testing Service (ETS) administered roughly $3,000,000$ paper-based exams. Although some of these exams (such as the GREs) are no longer paper-based, the vast majority are expected to remain paper-based for many years to come. Companies such as Scantron and NCS cater to the increasing industrial use of paper based standardized testing for hiring, training and for supervision purposes. Scantron scans over sixty million forms [11] per year, while NCS (with revenues over $500 million dollars in 1999) boasts to capture data from over 200 million documents annually [13], large portion of which are marked forms. Roughly 64,000 forms are scanned annually in class exams at SUNY Stony Brook alone, which extrapolates to perhaps six million per year in universities across the United States.

Despite an extensive literature search, and discussions with experts in the standardized testing industry, we have uncovered no previous work on algorithmically detecting/correcting student shift-errors. This is likely a consequence of the difficulty of recognizing shifts. However, there *has* been considerable empirical study into factors affecting student's answer changing behavior, which can be observed by studying erasure marks on grading sheets. We are aware of at least 20 studies on answer-changing in multiple choice exams – the earliest dating back to 1929 [3, 4, 6, 7, 9, 12, 14, 15, 16, 17, 18, 19, 20, 21, 22, 24, 25, 27, 28, 30]. The aggregate of these studies suggests that roughly 3% of all answers are erased and changed, an unknown fraction of which results from student-detected shift-errors.

The best evidence for the frequency of shift errors on standardized tests arises from Matter's [20] studies of student answer-changing behavior. Matter sought to identify mismarkings due to clerical errors in the set of erasures. By analyzing 21,829 erasures on 1903 student exams, Matter confirmed that roughly 3% of all answers were erased. Of these, Matter observed 633 runs of 3 or more consecutive

answer changes. Since an assumption of independent errors at this observed rate would be expected to create no more than 20 such runs, Matter concluded that almost all of these runs were due to clerical errors. By extrapolating from this data, we can conclude that a substantial fraction of all tests exhibit *corrected* shift errors. A significant problem with *uncorrected* shift errors remains even if 90% of all shift errors are corrected by students.

The factors affecting the fairness of multiple-choice exams, have also been extensively studied. Aamodt et.al. [1] examined the effect that the exam taker's moods have on performance, and concluded that sad exam takers do better than happy ones. Sinclair et.al. [26] found that even the color of the paper can affect performance, and that students with blue answer sheets outperform those with red answer sheets. Harpp et.al. [10] studied cheating on 75 different sets of undergraduate university multiple choice exams over 4 years, discovering that at least 10% of examinees copy from one another. Fortunately, such copying can be almost completely eliminated by creating multiple versions of each exam. However, Balch [2] and Carlson et.al. [5] found that question ordering affects performance, and in particular that grouping questions by subject leads to higher scores. Thus creating multiple versions of exams (to combat cheating) may lead to bias even if all the questions are identical. Different approaches have been taken to correct for the effect of guessing. ETS penalizes the examinee for every wrong answer in an attempt to discourage guessing. Prieto et.al. [23] determined that although the penalizing approach reduces the effects of guessing, it also alters the distribution of scores in the exam. We note that any attempt to discourage guessing should increase the frequency of student shift errors, because each skipped question provides an opportunity to start a shift.

1.2 Notation and Shift Model

Throughout this paper, we assume that we are given a set of exams and the correct answer key. Each exam is described by strings of length N on an alphabet Σ, where $\alpha = |\Sigma|$. Unanswered questions are assigned a symbol not in Σ, thus condemning them to be marked as incorrect in any re-alignment. We let $A_1 A_2 \ldots A_N$ denote the answer key and $E_1 E_2 \ldots E_N$ denote a given student's exam. A *patch* is a substring $E_i E_{i+1} \ldots E_j$ of a given exam E.

The structure of a right *shift* is shown in Figure 1. The questions intended to have been marked in positions $x+1$ to $x+l$ were marked in positions $x+\Delta+1$ to $x+\Delta+l$, and random answers were inserted in positions $x+1$ to $x+\Delta$. The shift size $s = \Delta + l$ is the number of questions that will potentially be marked incorrectly. The problem of shift detection can then be defined as discovering the most likely intended answer string, given the marked string.

2 Three Approaches to Shift Detection and Correction

Our goal is to discriminate between exams that contain shifted patches, and exams that do not. There is an inherent tradeoff between selectivity and specificity; minimizing the number of false detections while detecting as many real

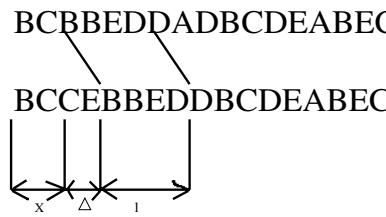


Fig. 1. A shift of length l by offset Δ at position x.

shifts as possible. We have developed and evaluated three basic approaches to shift-detection, each of which scans an individual exam, and identifies/scores patches in it that represent potential shifts. The models differ according to what information they consider in scoring the patches, and are ordered by increasing sophistication. Here we present a brief description of each model:

- The *dynamic programming model* performs a standard string alignment between the answer key and the exam, where the cost of a shift is set high enough to discourage its use, since shifts are presumed to be rare. Each shift represents a temporary deviation from the main diagonal of the dynamic programming matrix. This method performed badly, however, tending to excessively reward poor-performing students.
- The *single scan model* analyzes each patch of a given exam using a probabilistic model (discussed in Sections 3 and 4) to determine the likelihood that a student with this score will have a patch with the given number of errors and the given improvement. This model accounts for the fact that higher-scoring students are less likely to have poor-scoring patches, and thus any such patch may be more significant than a patch made by a lower-scoring student.
- The *double scan model* extends the single scan model by using the probability distribution of each answer for each question by the entire class. Obtaining this information requires an initial pass through the exams prior to assigning grades to any of them. We can augment our probability model to take advantage of this information, since errors in a patch of relatively easy questions are more suggestive of a shift.

3 Discovering Suspicious Patches

Consider the characterization of each exam E with a binary string of length N, where 0 denotes a correct answer and 1 an incorrect answer. We term a patch *suspicious* if it contains an unusually large number of errors (i.e. 1s). Let $P(N, n, k, m)$ denote the probability that such a binary string of length N with m 0s contains a patch of length n with at least k 1s. For the single scan model, this probability is sufficient to evaluate whether a patch is suspicious. For the

double scan model, we must add question bias to the calculation of $P(N, n, k, m)$. Figure 2 illustrates the effect that the student's score T and the length of the exam N have on our ability to detect shifts.

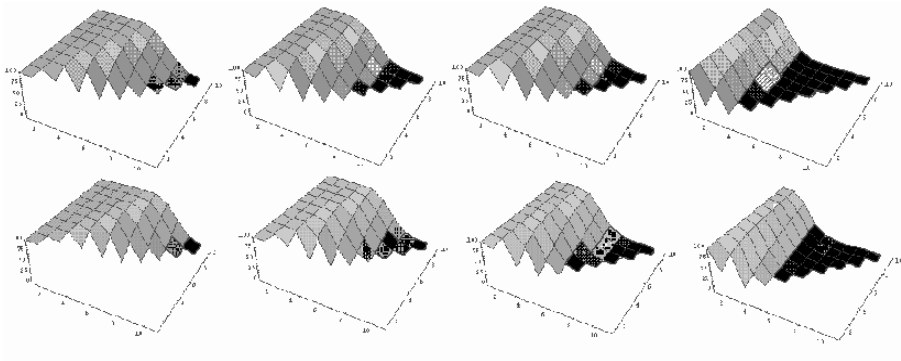


Fig. 2. The probability a given test has a n length patch with k wrong answers. The top row measures exams with $N = 50$, while the bottom row exams with $N = 100$. Student performance varies from left to right along each row for $T = 60\%$, 70%, 80%, and 90%. The dark areas represent probabilities small enough to make the associated shifts detectable.

3.1 Computing $P(N, n, k, m)$

Computing $P(N, n, k, m)$ can be done in $O(N2^n)$ time by building an appropriate automaton, which simulates a traversal of all possible binary strings of length N. According to the recurrence given in (2) where $P(N, n, k, m) = Prob_n(N, k)$ having (1) as a termination condition, $Prob(E_{N-n}) = (N - m)/N$ the probability that E_{N-n} is 1, and $I(E_N) \in \{0, 1\}$ the identity of the Nth element. In order to supply $I(E_N)$ the automaton must retain as states all binary strings of length n for every stage in the traversal, giving a total of $(N - n)2^n$ states.

$$Prob_n(n-1, x) = 1 \quad \forall x \tag{1}$$

$$\begin{aligned} Prob_n(N, k) = {} & Prob_n(N-1, k+1) \cdot (1 - Prob(E_{N-n}) \cdot I(E_N) \\ & + Prob_n(N-1, k-1) \cdot Prob(E_{N-n}) \cdot (1 - I(E_N)) \\ & + Prob_n(N-1, k) \cdot (1 - Prob(E_{N-n})) \cdot (1 - I(E_N)) \\ & + Prob_n(N-1, k) \cdot Prob(E_{N-n}) \cdot I(E_N) \end{aligned} \tag{2}$$

To avoid the complexity of computing the full automaton for large n, we can probabilisticly approximate $I(E_N)$, by counting the number of 1s rather than

maintaining the entire n length string. This method reduces the number of states to $n(N-n)$ and produces an approximation that proves good in practice. This method was used in our experiments, and proves sufficiently accurate. By using an I/O probabilistic automaton as described in [29], we can symbolically precompute the recurrence in n^2 runs of the probabilistic machine. Details appear in the full paper.

3.2 Finding Suspicious Patches in the Double Scan Model

In the double scan model, we adjust the suspicious patch probabilities with respect to the difficulty of each question. First, we compute the probabilities $Q_1, Q_2, \cdots, Q_N$ that a random member of the class answered each question correctly, assuming no shift errors. We now adjust these probabilities to reflect the differences between students. For each exam E_i with score T_i we find a single constant C_i such that $\sum_{j=1}^{N} C_i \cdot Q_j = T_i$. Given this constant, we model the probability that student i correctly answered question j as $\min(C_i \cdot Q_j, 1)$. Using these probabilities, we can calculate the expected average of a patch s, denoted E_s. The relative level of difficulty of this patch s is then E_s/T_i for student i. We then adjust the suspicious patch probability according to this level of difficulty. Patches of low average will now have stricter suspiciousness thresholds than patches of high average. Details appear in the full paper.

4 Suspicious Patch Scoring

Suspicious patch detection discovers patches with abnormally many wrong answers, but that by itself it is not an effective discriminator between true and false shifts. We use suspicious patch detection in conjunction with a second probabilistic system which we call *suspicious patch scoring*. Suspicious patch scoring is derived from the observation that correcting a real shift should result in a significant increase in the score of the exam. We considered two different models:

- *The Independent Probability Model.* It is natural to assume that a shifted block of answers should score as much as a random sequence, since each answer was intended for a neighboring position. The probability that a block of length N on alphabet α yields at least B correct answers is well defined and computed by Equation 3. This equation is independent of the length of the exam or its score. Table 1 gives insight into the strength of this method.

$$Prob(N, B, \alpha) = \sum_{k=B}^{N} \binom{N}{k} (\frac{1}{\alpha})^B (\frac{\alpha - 1}{\alpha})^{N-B} \tag{3}$$

- *The Answer Key Dependent Model.* However, the independent model does not exploit all information we have about the benefit probability of a shifted patch. A shifted patch's behavior depends on the exam structure and is not necessarily random. Consider a true-false exam ($\alpha = 2$) whose answer key

Table 1. The probability that a random patch of length n has c correct answers in it. For $\alpha = 5$, $2 \leq n \leq 10$ and $1 \leq c \leq n$. Probabilities low enough to be useful for our shift detection are in bold face.

$n \setminus c$	1	2	3	4	5	6	7	8	9	10
2	.360	**.040**								
3	.488	.104	**.008**							
4	.590	.181	**.027**	**.002**						
5	.672	.263	**.058**	**.007**	**.000**					
6	.738	.345	.099	**.017**	**.002**	**.000**				
7	.790	.423	.148	**.033**	**.005**	**.000**	**.000**			
8	.832	.497	.203	**.056**	**.010**	**.001**	**.000**	**.000**		
9	.866	.564	.262	**.086**	**.020**	**.003**	**.000**	**.000**	**.000**	
10	.893	.624	.322	**.121**	**.033**	**.006**	**.001**	**.000**	**.000**	**.000**

alternates true and false. Any low scoring patch will yield a high benefit when shifted, and will be considered shifted by the independent model, so the independent probability model will score these events as rarer than they in fact are. Our answer sequence dependent model takes in account the structure of the answer key A, as well as the size of the alphabet. This model is based on the following recurrence, where S is the patch, and $Prob(S, B) = Prob(N, B, \alpha)$. $Prob(0, B) = 0$ for $B \neq 0$, and 1 otherwise.

$$\begin{aligned} Prob(S, B) &= Prob(S_{|S|} \neq A_{|S|+1}) \cdot Prob(S-1, B) + Prob(S_{|S|} \\ &= A_{|S|+1}) \cdot Prob(S-1, B-1) \end{aligned}$$

where

$$Prob(A_{i+1} = S_i) = Prob(S_i = A_i | A_i = A_{i+1}) + \frac{1}{\alpha - 1} \cdot Prob(S_i \neq A_i | A_i \neq A_{i+1})$$

$$Prob(A_{i+1} \neq S_i) = Prob(S_i \neq A_i | A_i = A_{i+1}) + Prob(S_i \neq A_i | A_i = A_{i+1}) \quad (4)$$

5 Experimental Results

We conducted experiments on five different sets of scanned multiple-choice exams given at Stony Brook, for a a total of 684 papers. The characteristics of the five exams are reported in Table 2; they differ significantly in length, alphabet, and difficulty. These experiments are designed to answer two questions - (1) how well do our methods detect shifted exams and (2) how often do actual exams contain uncorrected shift errors.

To answer the first question, we constructed simulated shift errors and measured how well we detect them. To simulate shift errors, we picked patches of each exam at random and performed a shift, producing N l-shifts of each exam, for $3 \leq l \leq 10$. For simplicity, we limit attention to shifts with $\Delta = \pm 1$, where

Table 2. Average, number of questions and number of students in the exams used for performance testing.

	Questions	alphabet	Students	Average
Exam 1	33	5	204	81.4%
Exam 2	50	5	204	70.7%
Exam 3	50	4	101	68.5%
Exam 4	50	5	66	65.4%
Exam 5	30	4	109	61.8%

each exam contains at most one shift error. We seek a confidence threshold which identifies all significant simulated shifts while identifying few if any shifts in the original tests. Observe that a non-trivial fraction of these random shifts left the scores unchanged or even higher than the original, so no system could hope to identify all of these simulated shifts. For example, of the generated shifts of length 6, 11.3% did not reduce the student's score and thus should not be detected/corrected.

Table 3 discusses our detection success rate according to method used. The dynamic programming method proves relatively insensitive, detecting only 30% of simulated shifts of length 10 patches. Both the single and double scan methods did considerably better, correctly identifying roughly 90% of such shifts, and a substantial fraction of shifts as small as three answers. The double-scan method performed only slightly better than the single-scan method, perhaps not enough to justify the additional complexity. In order to appreciate the detection level described in Table 3, a closer examination of the detected shifts is needed. For example, 62% of the shifts of length 6 were detected in exam 2. This corresponds to 99% of all shifts that caused a loss greater than 2 questions, and 82% of shifts that caused a loss greater than 1 question.

To assess how often actual exams contain uncorrected shift errors, we note that at these probability thresholds, roughly 1-2% of the original exams scored as having shift errors. Properly interpreting our results requires justifying whether our detected shifts in the original tests reflect actual student clerical errors. We believe that they do, and have provided the details of all our detected shifts in Table 4 for inspection.

6 Adversary Strategies and Adaptability

It is possible for a clever but devious student to increase their exam score by taking advantage of the shift-correction machinery described in this paper. For example, a student who is very confident in her answers for a sequence of questions but must guess the answers for all of a preceding run of questions may find it reasonable to deliberately shift their sequence of correct answers. Assuming the system discovers the shift, it may find it advantageous to also shift some of the guesses, thus in effect giving two chances for each guess.

Table 3. The portions of detected original and synthetic shifts according to length, and broken down by exam. Shifts were generated with lengths 3 to 10, and detected using the dynamic programming, single scan and double scan models.

Dynamic Prog	Exam1	Exam2	Exam3	Exam4	Exam5
Original exams	.010	.000	.018	.010	.000
Shift length 3	.016	.004	.027	.023	.015
Shift length 4	.028	.012	.035	.029	.018
Shift length 5	.048	.034	.052	.036	.033
Shift length 6	.083	.098	.084	.043	.040
Shift length 7	.133	.169	.144	.060	.055
Shift length 8	.212	.263	.208	.105	.096
Shift length 9	.266	.366	.253	.148	.145
Shift length 10	.326	.456	.314	.216	.209

Single Scan	Exam1	Exam2	Exam3	Exam4	Exam5
Original exams	.019	.000	.010	.015	.019
Shift length 3	.429	.171	.209	.127	.128
Shift length 4	.638	.321	.342	.258	.233
Shift length 5	.743	.453	.463	.376	.353
Shift length 6	.836	.611	.584	.498	.470
Shift length 7	.883	.696	.674	.587	.566
Shift length 8	.911	.766	.739	.702	.650
Shift length 9	.929	.809	.789	.752	.713
Shift length 10	.943	.842	.817	.797	.750

Double Scan	Exam1	Exam2	Exam3	Exam4	Exam5
Original exams	.019	.000	.010	.015	.019
Shift length 3	.446	.178	.211	.137	.138
Shift length 4	.649	.334	.352	.268	.246
Shift length 5	.759	.470	.476	.389	.366
Shift length 6	.839	.620	.587	.503	.477
Shift length 7	.884	.700	.675	.595	.570
Shift length 8	.913	.768	.740	.704	.656
Shift length 9	.930	.813	.789	.754	.718
Shift length 10	.943	.843	.818	.801	.755

Table 4. The putative shifts that were detected and corrected by our system in original exams. Patch scores are the number of correct answers in the original vs. corrected patch, and final scores are the original vs. corrected exam scores.

Exam	Exam1	Exam1	Exam1	Exam1	Exam4
Answer key	CBDCB	ACBDCB	DCBA	DABBBA	ABCBCBBDE
Student's answer	BDDBC	CBAABD	ADCB	ABBBAD	CBBCDCABD
Corrected answer	BDDB	CBAAB	DCB	ABBBA	BBCDCABD
patch scores (O/C)	1 / 3	0 / 3	0 / 3	2 / 5	1 / 5
final scores (O/C)	85% / 92%	85% / 95%	79% /89%	73% /83%	72% /80%

Exam	Exam3	Exam5	Exam5
Answer key	AABCCBACBADCBDCB	BBDDBBCCADBDAC	BCCADBDA
Student's answer	ABCCCABBADDCDCBA	DABDCABBCAABDB	BBCAADBD
Corrected answer	ABCCCABBADDCDCB	ABDCABBCAABDB	BCAADBD
patch scores (O/C)	5 / 11	1 / 7	2 / 5
final scores (O/C)	30% / 42%	17% / 37%	73% / 83%

The expected score of any randomly filled n-question exam is clearly n/α. However this expected score will increase if one employs too generous a method to correct for possible shift errors. We consider the interesting problem of determining the "right" way to "randomly" answer questions if the most generous possible method of shift correction is employed. We use the terminology developed by Dancik [8]. The *adaptability* of a string S is the expected length of the longest common subsequence of S and R relative to $n = |S| = |R|$, where R is a random string over the same α-sized alphabet Σ as S, i.e.

$$\text{adaptability}(S) = \frac{\sum_{S_i \in \Sigma^n} |LCS(S, S_i)|}{n\alpha^n} \tag{5}$$

It follows that adaptability$(S) \geq 1/\alpha$, since the length of the LCS is at least as large as the Hamming distance between two strings. We will be interested in adaptability$_\alpha(n)$, the maximum adaptability over all strings length n on an alphabet of size α. Observe that a string's adaptability increases monotonically with n towards a limit. Let AB denote the concatenation of strings A and B. Note that the adaptability$(SS) \geq$ adaptability(S) since $LCS(SS, AB) \geq LCS(S, A) + LCS(S, B)$ for all S, A, and B. The question identifying the highest adaptability string for infinite exams has been considered by Dancik [8]. Using approximating automata, he determined upper and lower bounds on maximum adaptability $\lim_{n\to\infty}$ adaptability$_\alpha(n)$ for small alphabet sizes, given in Table 5. Further, Dancik conjectured that the string $L = (01101001)*$ is the string of maximum adaptability, i.e. the best guesses for our student to make in long exams.

Here, we report the results of a computer search for higher adaptability strings. Our computational methods will be discussed in the full version of the paper. Table 6 reports the highest adaptability strings of length up 16. Table 7 lists the adaptability of four interesting sets of strings from lengths 16 to 32,

Table 5. Lower and Upper bounds on maximum adaptability for small alphabets [8].

α	Lower Bound	Upper Bound
2	0.80	0.88
3	0.67	0.80
4	0.58	0.75
5	0.53	0.70
6	0.50	0.66
7	0.46	0.63
8	0.44	0.60

and highlights the best performing string of each length. These are $(01)*$, L, and repetitions of the optimal strings of length 14 and 16. The string defined by $(01)*$ performs poorly in comparison to the others. Although we found strings for specific lengths that where were better than L, we were unable to find a regular expression that has higher adaptability than L for all or for most strings of length 8 or more. L appears to beat simple alternating forms like $(01)*$ because it can exploit the longer runs of identical characters which become more likely as length increases. We conjecture that the language of maximum adaptability strings is not regular.

Table 6. The strings of highest adaptability for a given length. Their complement string (not listed) is of the same adaptability.

Length	Adaptability	Highest Adaptability Strings
2	0.6250	01
3	0.6667	010
4	0.6875	0110 0101
5	0.7000	01101 01010 01001
6	0.7161	011001
7	0.7232	0110010 0100110
8	0.7344	01100110
9	0.7376	011001101 010011001
10	0.7453	0110100110 0110010110
11	0.7482	01101001101 01001101001
12	0.7542	011001011001
13	0.7562	0110010110010 0100110100110
14	0.7602	01101001011001 01100101101001
15	0.7618	011001011010010 010010110100110
16	0.7658	0110010110100110

Table 7. The adaptability of 3 high adaptability strings according to length. The string of highest adaptability of the three for each length appears in bold.

Length\String	(01)∗	(0110010110100110)∗	(01101001011001)∗	(01101001)∗
16	0.7344	**0.7658**	0.7643	0.7649
17	0.7353	0.7631	0.7618	**0.7662**
18	0.7361	**0.7688**	0.7684	0.7686
19	0.7368	0.7663	**0.7692**	0.7669
20	0.7375	0.7713	0.7712	**0.7726**
21	0.7381	0.7687	0.7697	**0.7734**
22	0.7386	0.7731	0.7748	**0.7751**
23	0.7391	0.7736	**0.7754**	0.7735
24	0.7396	0.7750	0.7770	**0.7780**
25	0.7400	0.7730	0.7756	**0.7784**
26	0.7404	0.7775	**0.7797**	**0.7797**
27	0.7407	0.7781	**0.7786**	0.7783
28	0.7411	0.7793	0.7819	**0.7820**
29	0.7414	0.7782	0.7821	**0.7823**
30	0.7417	0.7817	0.7830	**0.7833**
31	0.7419	0.7808	0.7813	**0.7819**
32	0.7422	0.7837	0.7844	**0.7850**

References

[1] M. G. Aamodt and T. Mcshane. A meta-analytic investigation of the effect of various test item characteristics on test scores and test completion times. *Public Personnel Management*, 21:151–160, 1992.

[2] W. R. Balch. Item order affects performance on multiple choice exams. *Teaching of Psychology*, 16:75–77, 1989.

[3] J. A. Bath. Answer-changing behavior on objective examinations. *Journal of Educational Research*, 61:105–107, 1967.

[4] M. D. Beck. The effect of item response changes on scores on an elementary reading achievement test. *Journal of Educational Research*, 71:153–156, 1976.

[5] J. L. Carlson and A. L. Ostrosky. Item sequence and student performance on multiple choice exams - further evidence. *Journal of Economic Education*, 23:232–235, 1992.

[6] C. A. Clark. Should students change answers on objective tests? *Chicago Schools Journal*, 43:382–385, 1962.

[7] D. A. Copeland. Should chemistry students change answers on multiple-choice tests? *Journal chemical Education*, 49:258, 1972.

[8] V. Dancik. *Expected length of Longest Common Subsequences*. PhD thesis, University of Warwik, Warwick, UK, 1994.

[9] K. A. Edwards and C. Marshall. First impressions on tests: Some new findings. *Teaching Psychology*, 4:193–195, 1977.

[10] D. N. Harpp and J. J. Hogan. Crime in the classroom - detection and prevention of cheating on multiple choice exams. *Journal of Chemical Education*, 70:306–311, 1993.

[11] R. Hertzberg. Vice president of marketing research, scantron. Personal communication, September 1999.

[12] G. E. Hill. The effect of changed responses in true-false tests. *Journal of Educational Psychology*, 28:308–310, 1937.

[13] NCS Inc. Annual report. World Wide Web, `www.ncs.edu`, September 1999.
[14] S. S. Jacobs. Answer changing on objective tests: Some implications for test validity. *Educational and Psychology Measurement*, 32:1039–1044, 1972.
[15] R. F Jarrett. The extra-chance nature of changes in students' responses to objective test items. *Journal of General Psychology*, 38:243–250, 1948.
[16] E. E. Lamson. What happens when the second judgment is recorded in a true-false test? *Journal of Educational Psychology*, 26:223–227, 1935.
[17] M. L. Lowe and C. C. Crawford. First impressions versus second thought in true-false tests. *Journal of Educational psychology*, 20:192–195, 1929.
[18] D. O. Lynch and B. C. Smith. Item response changes: Effects on test scores. *Measurement and Evaluation in Guidance*, 7:220–224, 1975.
[19] C. O. Mathews. Erroneous first impressions on objective tests. *Journal of Educational Psychology*, 20:260–286, 1929.
[20] M. K. Matter. *The relationship between achievement test response changes, ethnicity, and family income*. PhD thesis, University of Texas, Austin, 1985.
[21] D. J. Mueller and A. Shuedel. Some correlates on net gain resultant from answer changing on objective achievement test items. *Journal of Educational Measurement*, 12:251–254, 1975.
[22] P. J. Pascale. Changing initial answers on multiple choice achievement tests. *Measurement and Evaluation in Guidance*, 6:236–238, 1974.
[23] G. Prieto and A. R. Delgado. The effect of instructions on multiple-choice test scores. *European Journal of Psychological assessment*, 15:143–150, 1999.
[24] P. J. Reile and L. J. Briggs. Should students change their initial answers on objective type tests? more evidence regarding an old problem. *Journal of Educational Psychology*, 43:110–115, 1952.
[25] E. Reiling and R. Taylor. A new approach to the problem of changing initial responses to multiple choice questions. *Journal of Educational Measurement*, 9:67–70, 1972.
[26] R. C. Sinclair, A. S. Soldat, and M. M. Mark. Effective cues and processing strategy: Color-coded examination forms influence performance. *Teaching of Psychology*, 25:130–132, 1998.
[27] A. Smith and J. C. Moore. The effects of changing answers on scores on non-test-sophisticated examinees. *Measurement and Evaluation in Guidance*, 8:252–254, 1976.
[28] M. Smith, K. P. white, and R. H. Coop. The effect of item type on the consequences of changing answers on multiple choice tests. *Journal of Educational Measurement*, 16:203–208, 1979.
[29] E. W. Stark and G. Pemmasani. Implementation of a compositional performance analysis algorithm for probabilistic i/o automata. *PAPM99*, September 1999.
[30] G. R. Stoffer, K. E. Davis, and J. B. Brown. The consequences of changing initial answers on objective tests: A stable effect and a stable misconception. *Journal of Educational research*, 70:272–277, 1977.

An Upper Bound for Number of Contacts in the HP-Model on the Face-Centered-Cubic Lattice (FCC)

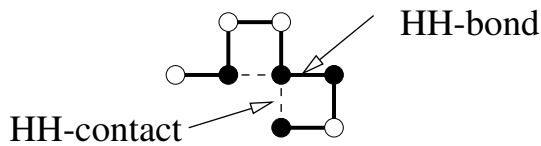
Rolf Backofen

Institut für Informatik, LMU München
Oettingenstraße 67, D-80538 München
backofen@informatik.uni-muenchen.de

Abstract. Lattice protein models are a major tool for investigating principles of protein folding. For this purpose, one needs an algorithm that is guaranteed to find the minimal energy conformation in some lattice model (at least for some sequences). So far, there are only algorithm that can find optimal conformations in the cubic lattice. In the more interesting case of the face-centered-cubic lattice (FCC), which is more protein-like, there are no results. One of the reasons is that for finding optimal conformations, one usually applies a branch-and-bound technique, and there are no reasonable bounds known for the FCC. We will give such a bound for Dill's HP-model on the FCC.

1 Introduction

Simplified protein models such as lattice models are used to investigate the protein folding problem, the major unsolved problem in computational biology. An important representative of lattice models is the HP-model, which has been introduced by [7]. In this model, the 20 letter alphabet of amino acids (called monomers) is reduced to a two letter alphabet, namely H and P. H represents *hydrophobic* monomers, whereas P represent *polar* or hydrophilic monomers. A *conformation* is a self-avoiding walk on the cubic lattice. The energy function for the HP-model simply states that the energy contribution of a contact between two monomers is -1 if both are H-monomers, and 0 otherwise. Two monomers form a *contact* in some specific conformation if they are not connected via a *bond*, and the euclidian distance of the positions is 1. One searches for a conformation which maximizes the number of contacts, which is a conformation whose hydrophobic core has minimal surface. Just recently, the structure prediction problem has been shown to be NP-hard even for the HP-model [4,5] on the cubic lattice. A sample conformation for the sequence PHPPHHPH in the two-dimensional lattice with energy -2 is the following:



R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 277–292, 2002.

The white beads represent P, the black ones H monomers. The two contacts are indicated via dashed lines.

For investigating general properties of protein-folding, one needs an algorithm which is guaranteed to find a conformation with maximal number of contacts (at least for some sequences, since the problem is NP-hard in general). Although there are approximation algorithms for the HP-model in the cubic lattice [6] and FCC [1], the need of an optimal conformation in this case implies that one cannot use approximate or heuristic algorithms for this purpose. To our knowledge, there are two algorithm known in the literature that find conformations with maximal number of contacts (optimal conformations) for the HP-model, namely [8,2]. Both use some variant of Branch-and-Bound.

The HP-model is original defined for the cubic lattice, but it is easy to define it for any other lattice. Of special interest is the face-centered-cubic lattice (FCC), which models protein backbone conformations more appropriately than the cubic lattice. When considering the problem of finding an optimal conformation, the problem occurs that no good bound on the number of contacts for the face-centered cubic lattice is known, in contrast to the HP-model. Both known algorithm for finding the optimal conformation search through the space of conformations using the following strategy:

- fix one coordinate (say x) of all H-monomers first
- calculate an upper bound on the number of contacts, given fixed values for the H-monomers.

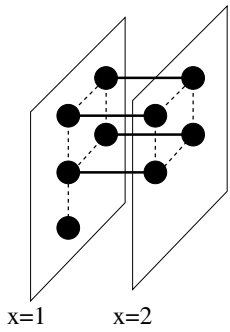


Fig. 1. H-Positions

An upper bound can easily be given in the case of the HP-model, if only the number of H-monomers are known in every plane defined by an equation $x = c$ (called x-layer in the following). For this purpose, one counts the number of HH-contacts and HH-bonds (since the number of HH-bonds is constant, and we do not care in which layer the HH-bonds actually are). Let us call this generalized contacts in the following. Then one distinguishes between generalized contacts within an x-layer, and generalized contacts between x-layers. Suppose that the positions occupied by H-monomers are given by black dots in Figure 1. Then we have 5 H-monomers in layer $x = 1$, and 4 H-monomers in $x = 2$. Furthermore, we have 4 generalized contacts between the layer $x = 1$ and $x = 2$ (straight lines), 5 contacts within $x = 1$ and 4 contacts within $x = 2$ (dashed lines). This coincide with upper bound given 5 H-monomer in $x = 1$, and 4 H-monomers in $x = 2$, which is calculated as follows. For the number of interlayer contacts, we know that every interlayer contact consumes 1 H-monomer in every layer. Hence, the maximal number of interlayer contacts is the minimum of the number of H-monomer in each layer, in this case $\min(5, 4) = 4$. The upper bound for the layer contacts is a bit more complicated, since it uses the concept of a frame. Consider some layer with n H-monomers.

Let $a = \lceil \sqrt{n} \rceil$ and $b = \lceil \frac{n}{a} \rceil$. (a, b) is the minimal rectangle (frame) around n H-monomers. Then the maximal number of contacts within this layer is upper bound by $2n - a - b$. In our example, we get for the first layer $n = 5$, $a = 3$ and $b = 2$, and the maximal number of layer contacts is then $10 - 3 - 2 = 5$, as it is the case in our example. For $n = 4$, we get $a = 2$, $b = 2$ and the maximal number is then $8 - 2 - 2 = 4$, as in our case. For details, see [8,2].

The problem with the face-centered-cubic lattice is, that there is no similar bound known (given the number of H-monomers in every x-layer). Part of the problem is that the interlayer contacts are much more complex to determine. The reason is that the face-centered cubic lattice has 12 neighbors (position with minimal distance), whereas the cubic lattice has only 6. Thus, in any representation of FCC, we have more than one neighbor in the next x-layer for any point $\boldsymbol{p}$, which makes the problem complicated. Such an upper bound will be given in this paper.

2 Preliminaries

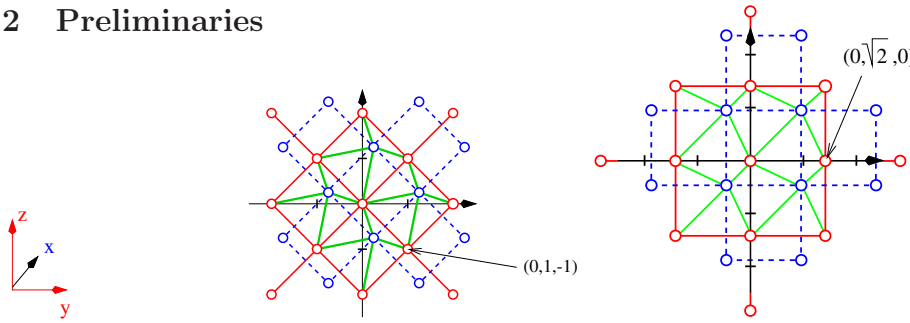


Fig. 2. In the first figure, we have shown two x-layers (where the x-axis is shown as the third dimension). The red circles are the lattice points in the first x-layer, where the red lines are the nearest neighbor connections. The blue circles are the points in the second x-layers. The green lines indicate the nearest neighbor connections between the first and the second x-layer. The second figures shows FCC after rotation by 45°

Given vectors $\boldsymbol{v_1}, \ldots, \boldsymbol{v_n}$, the lattice generated by $\boldsymbol{v_1}, \ldots, \boldsymbol{v_n}$ is the minimal set of points L such that $\forall \boldsymbol{u}, \boldsymbol{v} \in L$, both $\boldsymbol{u} + \boldsymbol{v} \in L$ and $\boldsymbol{u} - \boldsymbol{v} \in L$. An *x-layer* in a lattice L is a plane orthogonal to the x-axis (i.e., is defined by the equation $x = c$) such that the intersection of the points in the plane and the points of L is non-empty. The *face-centered cubic lattice* (FCC) is defined as the lattice $D_3 = \{\begin{pmatrix} x \\ y \\ z \end{pmatrix} \mid \begin{pmatrix} x \\ y \\ z \end{pmatrix} \in \mathbb{Z}^3 \text{and } x + y + z \text{ is even}\}$. For simplicity, we use a representation of D_3 that is rotated by $\phi = 45°$ along the x-axis. Since we want to have distance 1 between successive x-layers, and distance 1 between neighbors in one x-layer, we additionally scale the y- and z-axis, but leave the x-axis as it is. A partial view of the lattice and its connections, as well as the rotated lattice is given in Figure 2. Thus, we can define the lattice D'_3 to be the lattice that consists of the following sets of points in real coordinates: $D'_3 = \{\begin{pmatrix} x \\ y \\ z \end{pmatrix} \mid \begin{pmatrix} x \\ y \\ z \end{pmatrix} \in \mathbb{Z}^3\} \uplus \{\begin{pmatrix} x+0.5 \\ y+0.5 \\ z+0.5 \end{pmatrix} \mid \begin{pmatrix} x \\ y \\ z \end{pmatrix} \in \mathbb{Z}^3\}$. The first is the set of points

in even x-layers, the second the set of point in odd x-layers. A generator matrix for D'_3 is given in [3].

The set $N_{D'_3}$ of minimal vectors connecting neighbors in D'_3 is given by $N_{D'_3} = \left\{\begin{pmatrix}0\\\pm1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\\pm1\end{pmatrix}\right\} \uplus \left\{\begin{pmatrix}\pm1\\\pm0.5\\\pm0.5\end{pmatrix}\right\}$. The vectors in the second set are the vectors connecting neighbors in two different successive x-layers. Two points $\boldsymbol{p}$ and $\boldsymbol{p'}$ in D'_3 are *neighbors* if $\boldsymbol{p} - \boldsymbol{p'} \in N_{D'_3}$.

Colorings. The positions occupied by H-monomers in some conformation of the HP-model can be defined by colorings. A *coloring* is a function $f : D'_3 \to \{0, 1\}$. We denote with points(f) the set of all points colored by f, i.e., points(f) = $\{\boldsymbol{p} \mid f(\boldsymbol{p}) = 1\}$. With num($f$) we denote |points($f$)|. Let f_1 and f_2 be colorings. With $f_1 \cup f_2$ we denote the coloring f with points(f) = points(f_1) $\cup$ points(f_2). Two colorings f_1, f_2 are *disjoint* if their set of points are disjoint. $f_1 \uplus f_2$ denotes the disjoint union of colorings. Given a coloring f, we define the number of contacts con(f) of f by con(f) = $\frac{1}{2}|\{(\boldsymbol{p}, \boldsymbol{p'}) \mid f(\boldsymbol{p}) = 1 = f(\boldsymbol{p'}) \wedge (\boldsymbol{p} - \boldsymbol{p'}) \in N_{D'_3}\}$.

A coloring f is called a *coloring of the plane* $x = c$ if $f(x, y, z) = 1$ implies $x = c$. We say that f is a *plane coloring* if there is a c such that f is a coloring of plane $x = c$. We define $\text{Surf}_{pl}(f)$ to be the surface of f in the plane $x = c$. I.e., $\text{Surf}_{pl}(f)$ is the number of pairs $(\boldsymbol{p}, \boldsymbol{p'})$ such that $\boldsymbol{p} - \boldsymbol{p'} \in N_{D'_3}$ (i.e., they are neighbors), $f(\boldsymbol{p}) = 1$, $f(\boldsymbol{p'}) = 0$, and $\boldsymbol{p'}$ is also in plane $x = c$. With $\min_y(f)$ we denote the integer $\min\{y \mid \exists z : f(c, y, z) = 1\}$. $\max_y(f)$, $\min_z(f)$ and $\max_z(f)$ are defined analogously.

3 Description of the Upper Bound

Our purpose is to give a upper bound on the number of contacts, given that n_c H-monomers are in the x-layer defined by $x = c$. Thus, we need to find a function $b(n_1, \ldots, n_k)$ with

$$b(n_1, \ldots, n_k) \geq \max\left\{ \text{con}(f) \middle| \begin{array}{l} f = f_1 \uplus \ldots \uplus f_k,\ \text{num}(f_c) = n_c \\ \text{and } f_c \text{ is a coloring of plane } x = c \end{array} \right\}.$$

To develop $b(n_1, \ldots, n_k)$, we distinguish between contacts $(\boldsymbol{p}, \boldsymbol{p'})$ where both $\boldsymbol{p}$ and $\boldsymbol{p'}$ are in one x-layer, and contacts $(\boldsymbol{p}, \boldsymbol{p'})$ where $\boldsymbol{p}$ is in an layer $x = c$, and $\boldsymbol{p'}$ is in the layer $x = c+1$. The contacts within the same x-layer are easy to bound by bounding the surface $\text{Surf}_{pl}(f_c)$. Since every point in layer $x = c$ has four neighbors, which are either occupied by an colored point, or an uncolored point, we get $4 \cdot \text{num}(f) = \text{Surf}_{pl}(f_c) + 2 \cdot LC$, where LC is the number of layer contacts. The hard part is to bound the number of contacts between two successive layers.

For defining the bound on the number of contacts between two successive layers, we introduce the notion of a i-point, where $i = 1, 2, 3, 4$. Given any point in $x = c + 1$, then this point can have at most 4 neighbors in the plane $x = c$. Let f be a coloring of the plane $x = c$. Then a point $\boldsymbol{p}$ in plane $x = c + 1$ is a *i-point for f* if it has i neighbors in plane $x = c$ that are colored by f (where $i \leq 4$). Of course, if one colors a i-point in plane $x = c + 1$, then this point

generates i contacts between layer $x = c$ and $x = c + 1$. In the following, we will restrict ourself to the case where $c = 1$ for simplicity. Of course, the calculation is independent of the choice of c.

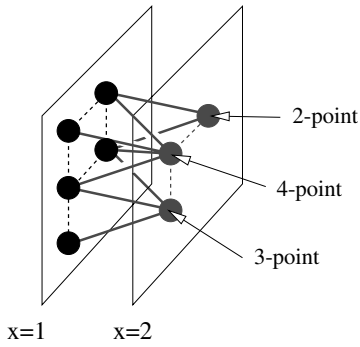


Fig. 3. H-Positions in FCC

Consider as an example the two colorings f_1 of plane $x = 1$ and f_2 of plane $x = 2$ as shown in Figure 3. f_1 consists of 5 colored points, and f_2 of 3 colored points. Since f_2 colors one 4-point, one 3-point and one 2-point of f_1, there are 9 contacts between these two layers. It is easy to see that we generated the most contacts between layers $x = 1$ and $x = 2$ by first coloring the 4-points, then the 3 points and so on until we reach the number of points to be colored in layer $x = 2$.[1]

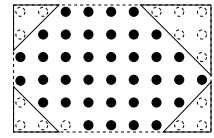
For this reason, we are interested to calculate the maximal number of i-points (for $i = 1, 2, 3, 4$), given only the number of colored points n in layer $x = 1$. But this would overestimate the number of possible contacts, since we would maximize the number of 4-, 3-, 2- and 1- point independently from each other. We have found a dependency between these numbers, which requires to fix the side length (a, b) of the minimal rectangle around all colored points in layer $x = 1$ (called the *frame*). In our example, the frame is $(3, 2)$. Of course, one has to search through all "reasonable frames" to find the maximal number of contacts between the two layers. This will be treated in a later section.

Denote with $max_i(a, b, n)$ the maximal number of i-points in layer $x = 2$ for any coloring of layer $x = 1$ with n-colored points and frame (a, b). Then we have found that

$$max_4(a,b,n) = n + 1 - a - b \qquad max_2(a,b,n) = 2a + 2b - 2\ell - 4$$
$$max_3(a,b,n) = \ell \qquad max_1(a,b,n) = \ell + 4.$$

The remaining part is to find $\ell = max_3(a, b, n)$, which is a bit more complicated. Define $k = \text{edge}(a, b, n) = \max\{k \in \mathbb{N} \mid ab - 4\frac{k(k+1)}{2} \geq n\}$, and $r = \text{ext}(a, b, n) = \lfloor \frac{ab - 4\frac{k(k+1)}{2} - n}{k+1} \rfloor$. Then $max_3(a,b,n) = \begin{cases} 4k + r & \text{if } 4k + r < 2(a-1) \\ 2(a-1) & \text{else.} \end{cases}$

$4k + r$ has a geometric interpretation. It is the maximal sum of the side length (one short side) of *uncolored* triangles at the corners of the frame for colorings with n points and frame (a, b). Consider the coloring



[1] Note that this might not necessarily be the coloring with the maximal number of contacts, since we might loose contacts within the layer $x = 2$; although this could be included in the calculation of the upper bound, we have excluded this effect in for simplicity

with $n = 38$, $a = 6$ and $b = 9$. The sum of the triangle side lengths (one short side) is $2+2+2+3 = 9$. This is exactly the same as $4\text{edge}(a,b,n) + \text{ext}(a,b,n)$.

Plan of the Paper. In Section 4, we will determine the number of points having n possible contacts, given some parameter of the coloring f of plane $x = c$. The parameters are the surface $\text{Surf}_{pl}(f)$, and the number of points with 3 possible contacts.

In Section 5, we will then show how we can determine the number of points having 3 possible contacts, given $\text{Surf}_{pl}(f)$. $\text{Surf}_{pl}(f)$ is determined by the minimal rectangle (called frame) around all points colored by f. Thus, we get an upper bound for both the contacts in the plane $x = c$, and the contacts between $x = c$ and $x = c + 1$ by enumerating all possible frames for f. Of course, we cannot enumerate *all* frames. Thus, we introduce in Section 6 a concept of "sufficiently filled frames", i.e. frames that are not too big for the number of points to be colored within the frame. These frames will be called normal. We then prove that it is sufficient to enumerate only the normal frames to get an upper bound. In fact, this is the most tedious part of the construction.

4 Number of Points with 1, 2, 3, 4-Contacts

In the following, we want to handle caveat-free, connected colorings. A coloring is connected if for every two points colored by f, there is a path $\boldsymbol{p} = \boldsymbol{p_1} \dots \boldsymbol{p_n} = \boldsymbol{p'}$ such that all points $\boldsymbol{p_1} \dots \boldsymbol{p_n}$ are colored by f, and $\boldsymbol{p_i} - \boldsymbol{p_{i+1}} \in N_{D_3'}$. Let f be a coloring of plane $x = c$. A *horizontal caveat in* f is a k-tuple of points $(\boldsymbol{p_1}, \dots, \boldsymbol{p_k})$ such that $f(\boldsymbol{p_1}) = 1 = f(\boldsymbol{p_k})$, $\forall 1 < j < k : f(\boldsymbol{p_j}) = 0$ and $\forall 1 \leq j < k : \left(\boldsymbol{p_{j+1}} = \boldsymbol{p_j} + \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}\right)$. A *vertical caveat* is defined analogously satisfying $\forall j < k : \left(\boldsymbol{p_{j+1}} = \boldsymbol{p_j} + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}\right)$ instead. We say that f contains a *caveat* if there is at least one horizontal or vertical caveat in f.

Definition 1. *Let f be a coloring of plane $x = c$. We say that a point $\boldsymbol{p}$ is a 4-point for f if $\boldsymbol{p}$ is in plane $x = c + 1$ or $x = c - 1$ and $\boldsymbol{p}$ has 4 neighbors $\boldsymbol{p_1}, \dots, \boldsymbol{p_4}$ in plane $x = c$ with $f(\boldsymbol{p_1}) = \dots = f(\boldsymbol{p_4}) = 1$. Analogously, we define 3-points, 2-points and 1-points. Furthermore, we define $\#4_{c-1}(f) = |\{\boldsymbol{p} \mid \boldsymbol{p}$ is a 4-point for f in $x = c - 1\}|$. Analogously, we define $\#4_{c+1}(f)$ and $\#i_{c\pm1}(f)$ for $i = 1, 2, 3$.*

Trivially, we get for any coloring f of plane $x = c$ that $\forall i \in [1..4] : \#i_{c-1}(f) = \#i_{c+1}(f)$. Hence, we define for a coloring f of plane $x = c$ that $\#i(f) = \#i_{c-1}(f)$ $(= \#i_{c+1}(f))$ for every $i = 1 \dots 4$. For calculating the number of i-points for a coloring f of plane $x = c$, we need the additional notion of x-steps for f. An x-step f consists of 3 points in $x = c$ that are sufficient to characterize one 3-point.

Definition 2 (X-Step). *Let f be a coloring of plane $x = c$. An x-*step for f *is a triple $(\boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{p_3})$ such that $f(\boldsymbol{p_1}) = 0$, $f(\boldsymbol{p_2}) = 1 = f(\boldsymbol{p_3})$, $\boldsymbol{p_1} - \boldsymbol{p_2} = \pm \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$ and $\boldsymbol{p_1} - \boldsymbol{p_3} = \pm \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$. With* xsteps$(f)$ *we denote the number of x-steps of f.*

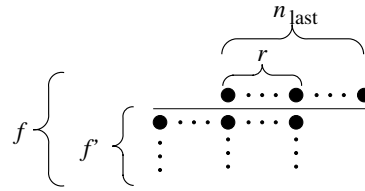
Lemma 1. *Let f be a connected, horizontal caveat-free coloring of the plane $x = c$. Then the following equations are valid:*

(a) $\#4(f) = n + 1 - \frac{1}{2}\mathrm{Surf}_{pl}(f)$ (b) $\#3(f) = \mathrm{xsteps}(f)$

(c) $\#2(f) = 2n - 2\#4(f) - 2\#3(f) - 2$ (d) $\#1(f) = \mathrm{xsteps}(f) + 4$

Proof (sketch). The proof is by induction on the number of rows. We will handle only the simple case of plane colorings where the different rows do overlap (since we can show that non-overlapping colorings are not optimal). The base case is trivial. For the induction step, let f be a plane coloring of height $h + 1$. Let the coloring f' be f with the row $z = \max_z(f)$ deleted.

Equation (a)**:** Let n_{last} be the number of points introduced in the last row of f, and let r be the number of points where the last row of f overlaps with the last row of f' as shown below:

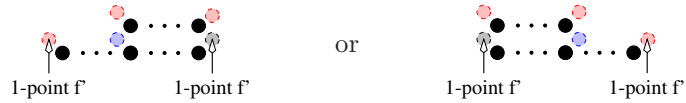


By our assumption of overlapping colorings we get $r > 0$. Then

$$\#4(f) = \#4(f') + (r - 1)$$
$$\text{and} \qquad \mathrm{Surf}_{pl}(f) = \mathrm{Surf}_{pl}(f') + 2(n_{\mathrm{last}} - r) + 2.$$

Using these values and the induction hypotheses, a simple calculation proves the claim.

Eqns. (b) **and** (d)**:** We will show only one case for the arrangement of the last two rows. Assume that the last two rows are of the forms



where the points colored by f are given in black. By this arrangement of the last two rows of f, f contains one more x-steps than f'. Thus, one can easily check that

$$\#3(f) = \#3(f) + 1$$
$$\text{and} \qquad \#1(f) = \#1(f) + 1,$$

from which one can verify Eqns. (b) and (d) by the induction hypotheses.

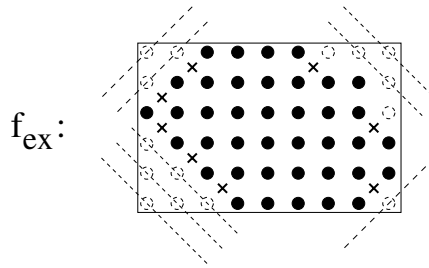
Equation (d) follows from the other equations and the equation

$$4n = 4 \cdot \#4(f) + 3 \cdot \#3(f) + 2 \cdot \#1(f) + 1 \cdot \#1(f).$$

With this lemma we need only to bound $\mathrm{Surf}_{pl}(f)$ and xsteps(f) (which is a bound on the number of 3-points) to calculate bounds on the number of 4-, 3-, 2- and 1-points.

5 Bound on the Number of 3-Points.

Given a plane coloring f, then we denote with frame(f) the pair (a, b), where $a = \max_z(f) - \min_z(f) + 1$ and $b = \max_y(f) - \min_y(f) + 1$. a is called the *height of* f, and b is called the *width of* f. The frame gives us the possibility to given a lower bound on the surface of a plane coloring, which is then an upper bound on the layer contacts. We need more characteristics of a coloring than the frame to generate a bound for xsteps(f), which will be captured by the notion of a detailed frame. The formal definition can be found in [3]. In principle, the detailed frame just counts for every corner, how many diagonals we can draw (starting from the corner) without touching a point that is colored by f. E.g., consider the following plane coloring f_{ex} given by the black dots:



Note that there are 8 positions in the next layer that are 3-points for this coloring. We have indicated these points with a $\times$. We can draw 3 diagonals from the left-lower corner, 2 from the left upper, 1 form the right lower, and 2 from the right upper corner. Note that the number of 3-points near every corner is exactly the same. We will prove this relationship later.

The detailed frame of a coloring f is the tuple $(a, b, i_{lb}, i_{lu}, i_{rb}, i_{ru})$, where (a, b) is the frame of f, and i_{lb} is the number of diagonals that can be drawn from the left-bottom corner. i_{lu}, i_{rb}, i_{ru} are defined analogously. For f_{ex}, the detailed frame is $(6, 9, 3, 2, 1, 2)$. The interesting part is that the the number of diagonals to be drawn gives an upper bound for the number of points to be colored *and* for the number of x-steps.

Proposition 1. *Let* $(a, b, i_1, i_2, i_3, i_4)$ *be the detailed frame of a plane coloring* f. *Then* $\mathrm{num}(f) \leq ab - \sum_{j=1}^{4} \frac{i_j(i_j+1)}{2}$.

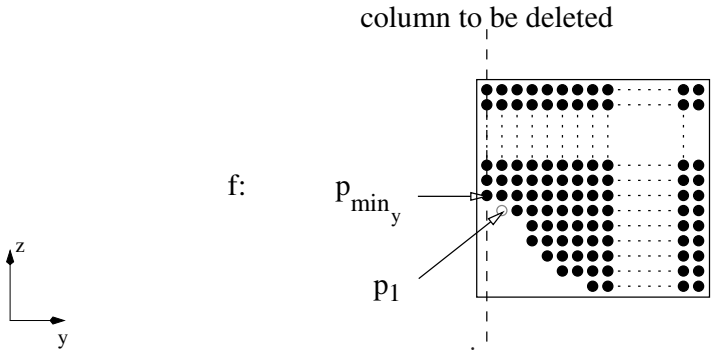
Definition 3 (Diagonal Caveat). *A* diagonal caveat *in* f *is a* k*-tuple of points* $(\boldsymbol{p_1}, \ldots, \boldsymbol{p_k})$ *of* D'_3 *with* $k \geq 3$ *such that* $f(\boldsymbol{p}_1) = 1 = f(\boldsymbol{p}_k)$, $\forall 1 < j < k : f(\boldsymbol{p}_j) = 0$ *and* $\forall 1 \leq j < k : \left(\boldsymbol{p_{j+1}} = \boldsymbol{p_j} + \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}\right) \ \vee \ \forall 1 \leq j < k : \left(\boldsymbol{p_{j+1}} = \boldsymbol{p_j} + \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}\right)$.

The number of diagonal caveats in f is denoted by diagcav(f). The next lemma gives us a good bound on the number of 3-points of a plane coloring f, given its edge characteristics. Recall the above example coloring f_{ex} with characteristic $(6, 9, 3, 2, 1, 2)$. Since the coloring does not have any diagonal caveats, the next lemma will show that xsteps(f) is given by $3 + 2 + 1 + 2 = 8$, as we have indicated.

Lemma 2. *Let f be a connected, caveat-free coloring of the plane $x = c$ which has a detailed frame $(a, b, i_1, i_2, i_3, i_4)$. Then*

$$\text{xsteps}(f) = \sum_{j \in [1..4]} i_j - \text{diagcav}(f).$$

Proof (sketch). By induction on the number of columns. The base case is trivial. For the induction step, we will illustrate only the case where the left lower corner of f is not completely filled. I.e., the coloring f is of the form



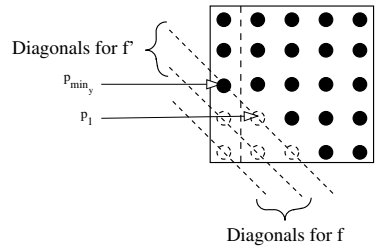
Note that this implies that f has the characteristics $(a, b, i_{lb}, 0, 0, 0)$, and that we have to show that xsteps(f) = i_{lb} − diagcav(f). Let f' be generated from f by deleting the first column (as indicated above). Let $\boldsymbol{p}_{min_y}$ be the lowest colored point in the first column of f. We distinguish the following cases for

$$\boldsymbol{p}_1 = \boldsymbol{p}_{min_y} - \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}.$$

1. $f(\boldsymbol{p}_1) = 0$ and $\boldsymbol{p}_{min_y}$ does not start a diagonal caveat in f. This implies that we cannot delete a diagonal caveat by deleting the first column. Additionally, we cannot delete or add an x-step. Hence,

$$\begin{aligned} \text{diagcav}(f') &= \text{diagcav}(f) \\ \text{xsteps}(f') &= \text{xsteps}(f). \end{aligned}$$

For determining the characteristics of f', consider the following example coloring f for this case:



Then by deleting the first column, we loose the diagonal through the left lower corner, but we can draw an additional one through $\boldsymbol{p}_1$. Thus, f' has the characteristics $(a, b-1, i_{lb}, 0, 0, 0)$. From this the claim follows from the induction hypotheses.

2. $f(\boldsymbol{p}_1) = 0$ and $\boldsymbol{p}_{min_y}$ starts a diagonal caveat. Then we get

$$\begin{aligned} \text{diagcav}(f') &= \text{diagcav}(f) - 1 \\ \text{xsteps}(f') &= \text{xsteps}(f) \end{aligned}$$

Since $\boldsymbol{p}_{min_y}$ starts a diagonal caveat, we know that $\boldsymbol{p}_1$ is part of this diagonal caveat. Hence, we cannot draw an additional diagonal in f' through $\boldsymbol{p}_1$ as in the previous case. Thus, f' has the characteristics $(a, b-1, i_{lb}-1, 0, 0, 0)$, and we get the claim again by the induction hypotheses.

3. $f(\boldsymbol{p}_1) = 1$. By deleting the first column, we deleted one x-step, but do not introduced or deleted a diagonal caveat. Hence,

$$\begin{aligned} \text{diagcav}(f') &= \text{diagcav}(f) \\ \text{xsteps}(f') &= \text{xsteps}(f) - 1 \end{aligned}$$

Furthermore, f' has the characteristics $(a, b-1, i_{lb}-1, 0, 0, 0)$. From this the claim follows from the induction hypotheses.

Proposition 2. *Let f be a caveat-free and connected coloring of plane $x = c$ with frame (a, b). Then* $\text{xsteps}(f) \leq 2(\min(a, b) - 1)$.

Proof (sketch). This proposition follows from the fact that in every connected and caveat-free coloring, there can be at most 2 x-steps in very row (resp. in very column).

6 Number of Contacts between Two Planes

As already mentioned in Section 2, for every coloring f we need to distinguish between contacts, where both points are in the same layer, and contacts, where the two corresponding points are in successive layers. The first one are called *layer contacts of f* (denoted by LC_f^c), whereas the later one are called *interlayer contacts*. Since we can split every coloring into a set of plane colorings, we define this notions for plane colorings.

Layer and Interlayer Contacts. Let f be a coloring of plane $x = c$. Since all colored points of f are in plane $x = c$, we can define the *layer contacts* LC_f^c of f in the plane $x = c$ by $\mathrm{LC}_f^c = \mathrm{con}(f)$. We define $\mathrm{LC}_{n,a,b}$ to be the maximum of all LC_f^c with $\mathrm{num}(f) = n$, f has frame (a, b) and f is a coloring of some plane $x = c$.

Proposition 3. *Under assumption of caveat-free colorings,* $\mathrm{LC}_{n,a,b} = 2n-a-b$.

Proof. Let f be a coloring of an arbitrary plane $x = c$. If f is caveat-free, then the surface of f in the plane $x = c$ is $2a + 2b$. Now we know that each of the n points has 4 neighbors, which are either occupied by another point, or by a surface point. Hence, we get $4n = 2\mathrm{LC}_{n,a,b} + 2a + 2b$.

Let f be a coloring of plane $x = c$, and f' be a coloring of plane $x = c'$. If $c' = c + 1$ (resp. $c - 1$), then we define the *interlayer contacts* $\mathrm{IC}_f^{f'}$ to be the number of contacts between plane $x = c$ and $x = c + 1$ (resp. $x = c - 1$) in the coloring $f \uplus f'$, i.e.: $\mathrm{IC}_f^{f'} = \left|\left\{ (\boldsymbol{p}, \boldsymbol{p'}) \middle| f(\boldsymbol{p}) = 1 \wedge f'(\boldsymbol{p'}) = 1 \wedge \boldsymbol{p'} - \boldsymbol{p} = \begin{pmatrix} \pm 1 \\ \pm 0.5 \\ \pm 0.5 \end{pmatrix} \right\}\right|$. Otherwise, $\mathrm{IC}_f^{f'} = 0$.

Let f be a coloring of plane $x = c$. With $\mathrm{contacts}_{\max}(f, n)$ we denote the maximal number of contacts between plane $x = c$ and $x = c+1$ by placing n points in $x = c+1$. I.e. $\mathrm{contacts}_{\max}(f, n) = \max \left\{ \mathrm{IC}_f^{f'} \middle| \begin{array}{l} f' \text{ is a plane coloring of } x = c + 1 \\ \text{with } \mathrm{num}(f') = n \end{array} \right\}$

Lemma 3. *Let f be a plane coloring of $x = c$. With $\delta_0(k)$ we denote* $\max(k, 0)$. *Then*

$$\mathrm{contacts}_{\max}(f, n) = 4\min(n, \#4(f)) + 3\min(\delta_0(n - \#4(f)), \#3(f))$$
$$+ 2\min(\delta_0(n - \sum_{i=3}^{4} \#i(f)), \#2(f)) + 1\min(\delta_0(n - \sum_{i=2}^{4} \#i(f)), \#1(f))$$

Next, we want not to consider a special coloring, but only the frame the coloring has. With $\mathrm{MIC}_{n_1,a_1,b_1}^{n_2,a_2,b_2}$ we denote

$$\max \left| \left\{ \mathrm{IC}_{f_1}^{f_2} \middle| \begin{array}{l} \mathrm{num}(f_1) = n_1 \wedge \mathrm{frame}(f_1) = (a_1, b_1) \wedge \\ \mathrm{num}(f_2) = n_2 \wedge \mathrm{frame}(f_2) = (a_2, b_2) \end{array} \right\} \right|$$

We define $\mathrm{MIC}_{n_1,a_1,b_1}^{n_2} = \max\limits_{a_2,b_2} \mathrm{MIC}_{n_1,a_1,b_1}^{n_2,a_2,b_2}$.

Proposition 4. $\mathrm{MIC}_{n_1,a_1,b_1}^{n_2} = \max \left\{ \mathrm{contacts}_{\max}(f, n_2) \middle| \begin{array}{l} f \textit{ is a plane coloring} \\ \textit{with frame } (a_1, b_1) \\ \textit{and } \mathrm{num}(f) = n_1. \end{array} \right\}$

Normal Colorings. Now we proceed as follows. We will first consider the case that the frame is sufficiently filled (where we define what this means in a moment). After that, we will show that we do not have to consider the frames which are not sufficiently filled (the pathological cases). We start with defining what "sufficiently filled" means.

Let a, b, n be positive numbers such that $ab \geq n$. We define $\text{edge}(a, b, n)$ by $\text{edge}(a, b, n) = \max\{k \in \mathbb{N} \mid ab - 4\frac{k(k+1)}{2} \geq n\}$. Let $k = \text{edge}(a, b, n)$. Then we define $\text{ext}(a, b, n) = \lfloor \frac{ab - 4\frac{k(k+1)}{2} - n}{k+1} \rfloor$. Intuitively, $\text{edge}(a, b, n)$ is the lower bound for the indent from the corners of a coloring of n points with frame (a, b), if we try to make the indents as uniform as possible (since uniform indents generate the maximal number of x-steps). $\text{ext}(a, b, n)$ is the number of times we can add 1 to $\text{edge}(a, b, n)$. Using this definitions, we can say what sufficiently filled means.

Proposition 5. $0 \leq \text{ext}(a, b, n) \leq 3$

Definition 4 (Normal). *Let n be an integer, (a, b) be a frame with $a \leq b$. Furthermore, let $k = \text{edge}(a, b, n)$ and $r = \text{ext}(a, b, n)$. We say that n is* normal for (a, b) *if either $4k+r < 2(a-1)$, or $4k+r = 2(a-1)$ and $ab - 4\frac{k(k+1)}{2} - r(k+1) = n$.*

The reason for using this notion is that if n is normal for (a, b), $\text{edge}(a, b, n)$ and $\text{ext}(a, b, n)$ yield a good bound on the number of x-steps of a plane coloring f. This will be shown in the next two lemmas.

Lemma 4. *If n is normal for (a, b), then there exists a caveat-free, connected plane coloring f such that $\text{xsteps}(f) = 4k + r$, where $k = \text{edge}(a, b, n)$ and $r = \text{ext}(a, b, n)$.*

Lemma 5. *Let (a, b) be a frame of a caveat-free and connected plane coloring f with $a \leq b$. Let $k = \text{edge}(a, b, \text{num}(f))$ and $r = \text{ext}(a, b, \text{num}(f))$. Then*

$$\text{xsteps}(f) \leq \begin{cases} 4k + r & \text{If } 4k + r < 2(a - 1) \\ 2(a - 1) & \text{else} \end{cases}$$

Definition 5 (Upper Bound for $\text{MIC}^{n'}_{n,a,b}$). *Let n be a number and $a \leq b$ with $ab \geq n \geq (a + b) - 1$. Let $k = \text{edge}(a, b, \text{num}(f))$ and $r = \text{ext}(a, b, \text{num}(f))$, and let*

$$l = \begin{cases} 4k + r & \text{if } 4k + r < 2(a - 1) \\ 2(a - 1) & \text{else.} \end{cases}$$

We define $max_4(a,b,n) = n+1-a-b$ $max_2(a,b,n) = 2a+2b-2l-4$
$max_3(a,b,n) = l$ $max_1(a,b,n) = l+4$.
With $\delta_0(n)$ *we denote* $max(n,0)$. *Now we define*

$$\begin{aligned}\mathrm{BMIC}^{n'}_{n,a,b} =& 4\min(n', max_4(a,b,n))\\ &+ 3\min(\delta_0(n' - max_4(a,b,n), max_3(a,b,n))\\ &+ 2\min(\delta_0(n' - \textstyle\sum_{i=3}^{4} max_i(a,b,n), max_2(a,b,n))\\ &+ 1\min(\delta_0(n' - \textstyle\sum_{i=2}^{4} max_i(a,b,n), max_1(a,b,n)).\end{aligned}$$

Theorem 1. *Under the condition given in the last definition,* $\mathrm{BMIC}^{n'}_{n,a,b}$ *is an upper bound for* $\mathrm{MIC}^{n'}_{n,a,b}$, *i.e.,* $\mathrm{MIC}^{n'}_{n,a,b} \leq \mathrm{BMIC}^{n'}_{n,a,b}$. *If* n *is normal for* (a,b), *then the above bound is tight, i.e.,* $\mathrm{BMIC}^{n'}_{n,a,b} = \mathrm{MIC}^{n'}_{n,a,b}$.

Proof. That $\mathrm{BMIC}^{n'}_{n,a,b}$ is an upper bound for $\mathrm{MIC}^{n'}_{n,a,b}$ follows from Lemmas 1, 3, 5 and from the fact that all plane colorings f with frame (a,b) satisfy $\mathrm{Surf}_{pl}(f) \geq 2a+2b$. That the bound is tight if n is normal for (a,b) follows from Lemma 4.

Note that any frame (a,b) for a connected, caveat-free coloring f with $\mathrm{num}(f) = n$ will satisfy $ab \geq n \geq (a+b)-1$, which is the reason for the bound on n in the above definition. We need to investigate properties of frames with respect to normality in greater detail. The next lemma just states that normality is kept if we either add additional colored points without changing the frame, or we switch to a smaller frame for the same number of colored points.

Lemma 6. *Let* n *be normal for* (a,b). *Then all* $n \leq n' \leq ab$ *are normal for* (a,b). *Furthermore, for all* (a',b') *such that* $a' \leq a \wedge b' \leq b$ *with* $a'b' \geq n$, *we have* n *is normal for* (a',b').

Clearly, we want to search only through the normal frames in order to find the frame (a,b) which maximizes $\mathrm{MIC}^{n'}_{n,a,b}$, given n and n'. This will be subject of Theorem 2.

Restriction to Normal Colorings. For this purpose, we define colorings which have

- maximal number of x-steps for given frame (a,b) (i.e., $\mathrm{xsteps}(f) = 2\min(a,b)-1$,
- maximal number of colored points under the above restriction.

To achieve $\mathrm{xsteps}(f) = 2\min(a,b)-1$, we must have 2 x-steps in every line. By caveat-freeness, this implies that these maximal colorings are as given in Figure 4.

The definition of these colorings is achieved by defining maximal line number distributions (where maximal refers to maximal x-steps). Line number distributions just counts for every line (row) in the coloring, how many colored points are on that line. The maximal line number distribution for a frame (a,b) is given

by $D^{a,b}_{\max 3}$, and which has the property that below the line with maximal number of colored points, we add 2 points from line to line, and after the maximal line we subtract 2 points. For every line number distribution D, we have defined a canonical coloring $f_{\mathrm{can}(D)}$. $\mathrm{num}(D)$ is the number of colored points of D, which is the same as the points colored by $f_{\mathrm{can}(D)}$. The precise definitions can be found in [3]. Figure 4 gives examples of the corresponding canonical colorings with maximal number of x-steps for the frames $(5,5)$, $(5,6)$, $(5,7)$ and $(6,7)$.

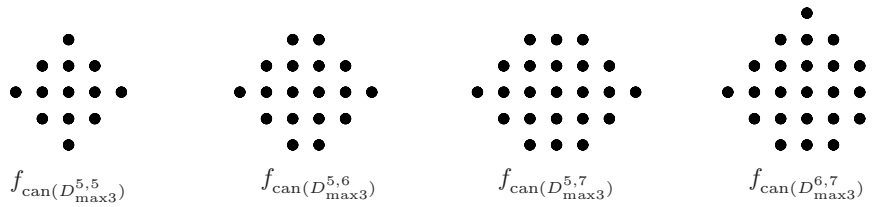


Fig. 4. Canonical colorings for the elements $(5,5)$, $(5,6)$, $(5,7)$ and $(6,7)$ of M.

Now we want to find for a given n a minimal frame (a_m, b_m) such that (a_m, b_m) has maximal number of x-steps. For this purpose, we define a set of tuples

$$M = \bigcup \{\{(n,n),(n,n+1),(n,n+2),(n+1,n+2)\} \mid n \text{ odd}\}$$

Note that M is totally ordered by the lexicographic order on tuples. Hence, we can define $\mathrm{MinF}(n)$ to be the minimal element $(a,b) \in M$ such that $\mathrm{num}(D^{a,b}_{\max 3}) \geq n$.

Note that we have excluded the case (n,n) with n even in the set M. The reason is that in this case, any coloring f of this frame which has maximal number of x-steps (namely $2(n-1)$) is not maximally overlapping. This implies that we can reduce this to a smaller frame. Figure 5 shows an example.

Lemma 7. *Let n be a number and (a,b) be $\mathrm{MinF}(n)$. Then*

- *There is a plane coloring f with frame (a,b) such that $\mathrm{num}(f) = n$ and $\mathrm{xsteps}(f) = 2(a-1)$.*
- *n is normal for (a,b) or $(a,b-1)$.*

Theorem 2 (Existence of Optimal Normal Frame). *Let n be an integer. Then for all frames (a',b') there is a frame (a,b) such that $a \leq a' \wedge b \leq b'$, n is normal for (a,b), $(a,b-1)$ or $(a-1,b)$ and $\forall n' : \mathrm{MIC}^{n'}_{n,a,b} \geq \mathrm{MIC}^{n'}_{n,a',b'}$.*

Proof (sketch). The main idea of this theorem is the following. Fix n and n'. Let (a,b) be a frame for n with maximal number of possible x-steps (i.e., there is a plane coloring f with $\mathrm{num}(f) = n$, f has frame (a,b), and $\mathrm{xsteps}(f) =$

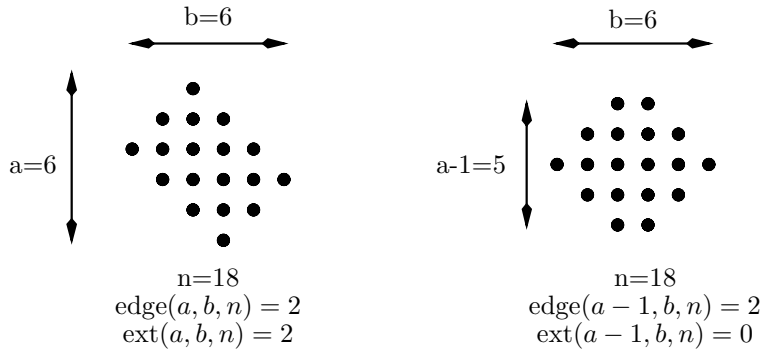


Fig. 5. Special case that a is even and $4\text{edge}(a,b,n) + \text{ext}(a,b,n) = 2(a-1)$. The first picture is the coloring for (a,b), the second for $(a-1,b)$.

$2\min(a,b)-1$). Then we know that $\text{MIC}^{n'}_{n,a+1,b} \leq \text{MIC}^{n'}_{n,a,b}$ since be enlarging the frame, we loose one 4-point by lemma 1, but can win at most one x-step by Proposition 2. The same holds for $\text{MIC}^{n'}_{n,a,b+1}$. Thus, it is sufficient to consider the minimal frame (a_m, b_m) which has maximal number of possible x-steps. But we can show that in this case, n is normal for (a_m, b_m), $(a_m, b_m - 1)$ or $(a_m - 1, b_m)$.

This theorem states, that we need only to consider all frames that are within distance one from a normal frame in order to find the frame (a,b) with that maximizes $\text{MIC}^{n'}_{n,a,b}$ for a given n and n'. Now we are able to summarize the results.

Theorem 3. *Let f be a connected, caveat-free coloring with $f = f_1 \uplus \ldots \uplus f_k$, where f_i is a coloring of the plane $x = i$. Let $a^m_k = \lceil \sqrt{n_k} \rceil$ and $b^m_k = \lceil \frac{n_k}{a^m_k} \rceil$. Then*

$$\text{con}(f) \leq \sum_{i=1}^{k-1} \max \left\{ \text{BMIC}^{n_{i+1}}_{n_i,a_i,b_i} + \text{LC}_{n_i,a_i,b_i} \mid \begin{array}{l} a_i b_i \geq n_i \text{ and } n_i \text{ is} \\ \text{normal for } (a_i,b_i), \\ (a_i-1,b_i) \text{ or } (a_i,b_i-1) \end{array} \right\} + \text{LC}_{n_k,a^m_k,b^m_k},$$

References

1. Richa Agarwala, Serafim Batzoglou, Vlado Dancik, Scott E. Decatur, Martin Farach, Sridhar Hannenhalli, S. Muthukrishnan, and Steven Skiena. Local rules for protein folding on a triangular lattice and generalized hydrophobicity in the hp model. *Journal of Computational Biology*, 4(2):275–296, 1997.
2. Rolf Backofen. Constraint techniques for solving the protein structure prediction problem. In Michael Maher and Jean-Francois Puget, editors, *Proceedings of 4^{th} International Conference on Principle and Practice of Constraint Programming (CP'98)*, volume 1520 of *Lecture Notes in Computer Science*, pages 72–86. Springer Verlag, 1998.

3. Rolf Backofen. *Optimization Techniques for the Protein Structure Prediction Problem*. Habilitationsschrift, University of Munich, 1999.
4. B. Berger and T. Leighton. Protein folding in the hydrophobic-hydrophilic (HP) modell is NP-complete. In *Proc. of the Second Annual International Conferences on Compututational Molecular Biology (RECOMB98)*, pages 30–39, New York, 1998.
5. P. Crescenzi, D. Goldman, C. Papadimitriou, A. Piccolboni, and M. Yannakakis. On the complexity of protein folding. In *Proc. of STOC*, 1998. To appear. Short version in *Proc. of RECOMB'98*, pages 61–62.
6. William E. Hart and Sorin C. Istrail. Fast protein folding in the hydrophobid-hydrophilic model within three-eighths of optimal. *Journal of Computational Biology*, 3(1):53 – 96, 1996.
7. Kit Fun Lau and Ken A. Dill. A lattice statistical mechanics model of the conformational and sequence spaces of proteins. *Macromolecules*, 22:3986 – 3997, 1989.
8. Kaizhi Yue and Ken A. Dill. Forces of tertiary structural organization in globular proteins. *Proc. Natl. Acad. Sci. USA*, 92:146 – 150, 1995.

The Combinatorial Partitioning Method

Matthew R. Nelson[1], Sharon L. Kardia[2⋆], and Charles F. Sing[3]

[1] Esperion Therapeutics, Inc., Dept. of Genomics, Ann Arbor, MI 48108, USA
[2] University of Michigan, Dept. of Epidemiology, Ann Arbor, MI 48109, USA
[3] University of Michigan, Dept. of Human Genetics, Ann Arbor, MI 48109, USA

Abstract. Recent advances in genome technology have led to an exponential increase in the ability to identify and measure variation in a large number of genes in the human genome. However, statistical and computational methods to utilize this information on hundreds, and soon thousands, of variable DNA sites to investigate genotype–phenotype relationships have not kept pace. Because genotype–phenotype relationships are combinatoric and non–additive in nature, traditional methods, such as generalized linear models, are limited in their ability to search through the high–dimensional genotype space to identify genetic subgroups that are associated with phenotypic variation. We present here a combinatorial partitioning method (CPM) that identifies partitions of higher dimensional genotype spaces that predict variation in levels of a quantitative trait. We illustrate this method by applying it to the problem of genetically predicting interindividual variation in plasma triglyceride levels, a risk factor for atherosclerosis.

1 Introduction

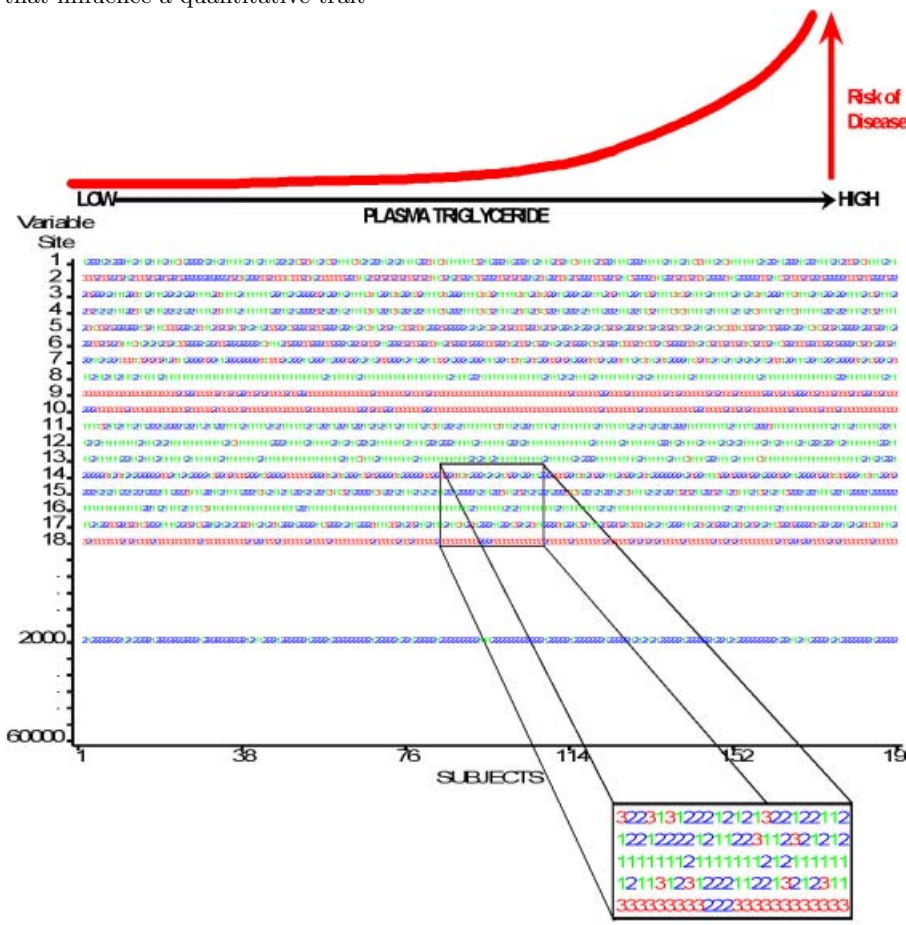
Genetic research seeks to understand how information encoded in the DNA molecule influences phenotypic variation in traits that are relevant in biology, agriculture, and medicine. The structure of DNA and its organization into functional gene units was revealed during the last half of the 20th century [12]. Within the next two years, the order of the four nucleotide bases that make up the DNA sequence of the entire human genome (approximately 3 billion bases) is expected to be known. The nucleotide order of the full sequence of the genomes of the major agriculture crops and animals will not be far behind. Currently, the first studies of complete gene sequences suggest that a variable DNA site may occur every 100 to 200 [10] nucleotide base pairs. Hundreds of thousands to millions of variable DNA sites are expected for every eukaryotic species. Moreover, rapid advances in molecular biology promise to make available early in the 21st century cheap and fast measurements of an individual's genotype at each of the variable DNA sites in the many genes that are thought to influence interindividual variation in measures of human health such as blood pressure, serum cholesterol, and glucose levels. The analytical challenge for the geneticists of the 21st century will be to identify which variable sites, in which genes,

⋆ Presenting Author.

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 293–304, 2002.

influence which traits, and in which environmental strata within which populations. Few variable sites are expected to have singular effects on trait variation that are biologically and statistically independent of the effects of other variable sites. Variations in combinations of sites will likely be of the greatest importance in determining the contribution of genotypic variation to interindividual trait variation.

Fig. 1. Illustration of the problem of identifying combinations of variable DNA sites that influence a quantitative trait



The scope of the analytical problem that we face is illustrated by the fictitious data set presented in Figure 1. Each row corresponds to the genotypes at a variable DNA site for a sample of 190 subjects. Each column corresponds to the genotypes of an individual at each of 60,000 measured variable DNA sites. In this illustration, the individuals are ranked according to their plasma

triglyceride level, which is related to their risk of heart disease. The problem is to identify the combination of variable loci sites that predict the quantitative phenotypic differences among individuals, and hence their risk of disease. Traditional statistical methods based on linear mathematical models [4,7] will not be appropriate for the task of identifying relevant patterns of DNA sequence variations [8,9] because the relationship between genotype and phenotype is known to be combinatoric and non–additive. Methods of data mining are more likely to be successful in revealing and sorting out the signals resident in the very large dimensional data sets available to geneticists. In this paper we present an analytical strategy—the combinatorial partitioning method (CPM)—that we are developing for searching through high dimensional genetic information to find genetic predictors of quantitative measures of human health.

2 The CPM Method

2.1 Method Overview

The objective of the combinatorial partitioning method is to identify sets of partitions of multi–locus genotypes that predict quantitative trait variability. To ease the description of the method for the remainder of this paper, we introduce the following notation for defining and discussing the CPM. Let L be the set of DNA variable sites (herein called 'loci') that are measured for a sample, where the number of loci in L is given as l. Let M be a subset of L loci, i.e. $M \subseteq L$, and the number of loci in set M be m. For a particular subset M, the set of observed m–locus genotypes is denoted as G_M with size g_M. We define a genotypic partition as a partition that includes one or more of all possible genotypes from the set G_M. A set of genotypic partitions, denoted K with size k, is a collection of two or more disjoint genotypic partitions. In the CPM, every one of the possible m–locus genotypes is included in one, and only one, of the disjoint genotypic partitions that make up a set, K. The collection of all possible sets of partitions of the genotypes G_M for all subsets M of L total loci into genotypic partitions defines the state space that is evaluated by the CPM.

Traditional genetic analyses of variation in trait levels begin by assuming that the number of genotypic partitions, which we denote as k, to be equal to the number of genotypes g_M on M. An analysis of variance among genotype means is typically followed by *a posteriori* comparisons to identify genotypes that have significantly different trait means. The objective of the CPM is to simultaneously identify the combination of loci that predict trait variability and group genotypes that are phenotypically similar into genotypic partitions. The motivation for partitioning can be illustrated using a single locus, two allele example. If we assume that the A allele is dominant to the a allele at a locus, we would partition the set of genotypes $\{AA,\ Aa,\ aa\}$ into subsets $\{AA,\ Aa\}$ and $\{aa\}$ based on the phenotypic similarity among genotypes within partitions as well as the differences between partitions. The higher dimensional application of the CPM is designed to identify the subset of $m \leq l$ loci that divide g_M

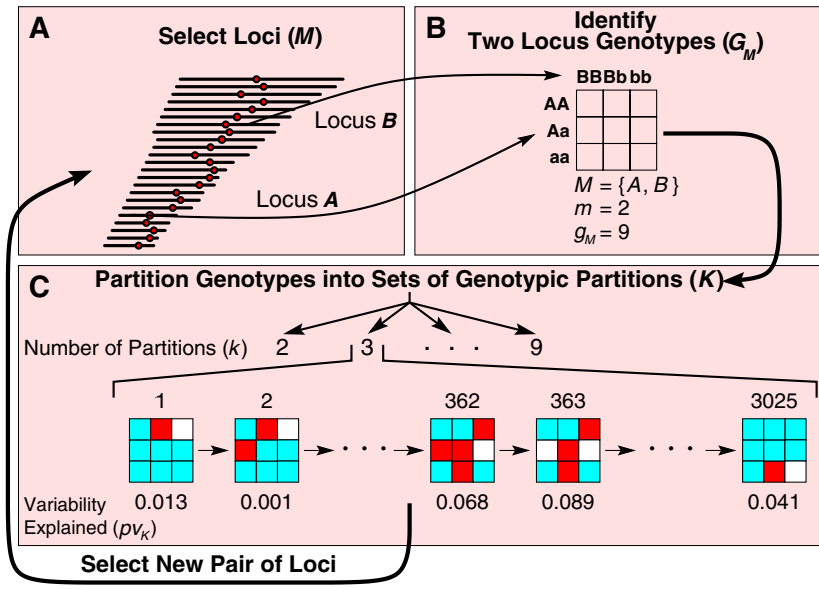
genotypes into k partitions that are similar within and most dissimilar between partitions for the mean of a quantitative trait.

This method can be broken down into three steps. The first step is to conduct the primary evaluation of the state space of sets of genotypic partitions for statistical measures of the phenotypic characteristics of the k partitions of genotypes. In this presentation we have used the phenotypic variance explained by the k class means as our statistical measure. The sets of genotypic partitions that predict more than a pre–specified level of trait variability are set aside. The second step is to validate each of the sets of genotypic partitions retained in Step 1 by cross validation methods. The third step is to select the best sets of genotypic partitions, based on the results of the cross validation from Step 2, and proceed to draw inferences about the relationships between the distribution of phenotypes and the distribution of the g_M multi–locus genotypes, for the selected sets of genotypic partitions.

2.2 Step 1: Searching and Evaluating the State Space

Searching the state space of all possible ways to partition m–locus genotypes into sets of genotypic partitions for all subsets of l loci can be separated into two nested combinatorial operations, illustrated in Figure 2 for the case when $m = 2$. The first operation, illustrated in Figure 2A, consists of systematically

Fig. 2. The combinatorial nature of the CPM applied to two loci at a time ($m = 2$) over a range of k.



selecting all possible subsets of loci for the desired values of m, of which there are $\binom{l}{m}$ ways. For each subset M, the m–locus genotypes are identified by the 3–by–3 grid for two diallelic loci A and B. This is illustrated in Figure 2B. Table 1 shows the number of combinations of loci that would need to be considered for a range of values in l and m.

Table 1. The number of possible combinations of loci for a range of values of m and l

	l				
m	20	50	100	1,000	10,000
2	1.9×10^2	1.2×10^3	5.0×10^3	5.0×10^5	5.0×10^7
3	1.1×10^3	2.0×10^4	1.6×10^5	1.7×10^8	1.7×10^{11}
4	4.8×10^3	2.3×10^5	3.9×10^6	4.1×10^{10}	4.2×10^{14}
5	1.6×10^4	2.1×10^6	7.5×10^7	8.3×10^{11}	8.3×10^{17}

Table 2. The number of possible sets of genotypic partitions for a range of values of m and k, assuming that for each m, $g_M = 3^m$

	k				
m	2	3	4	5	6
2	2.6×10^2	3.0×10^3	7.8×10^3	7.0×10^3	2.6×10^3
3	6.7×10^7	1.3×10^{12}	7.5×10^{14}	6.1×10^{16}	1.4×10^{18}
4	1.2×10^{24}	7.4×10^{37}	2.4×10^{47}	3.4×10^{54}	1.5×10^{60}
5	7.1×10^{72}	1.5×10^{115}	8.3×10^{144}	5.9×10^{167}	1.7×10^{186}

The second operation, depicted in Figure 2C, is to evaluate the possible sets of genotypic partitions over the desired range of k. The number of ways to partition g_M genotypes into a set of k genotypic partitions is known as a Stirling number of the second kind [2], computed from the sum

$$S(g_M, k) = \frac{1}{k!} \sum_{i=0}^{k-1} (-1)^i \binom{k}{i} (k-i)^{g_M}. \tag{1}$$

For diallelic loci the maximum number of m–locus genotypes that could be observed is $g_M = 3^m$. Table 2 demonstrates the dominating power of this combinatoric with the number of sets of genotypic partitions possible for a range of values in m and k. Clearly, only in cases where $m = 2$ can the search space be exhaustively evaluated. In cases where $m > 2$, we are developing alternative heuristic optimization methods, such as simulated annealing [5] or genetic algorithms [3] to search the state space.

In this paper, to illustrate the CPM, we restrict our application to the two locus case ($m = 2$), and evaluate directly all possible sets of genotypic partitions.

This complete enumeration of the state space has at least two advantages. First, when there is no *a priori* knowledge of how the state space of sets of genotypic partitions (as defined above) relates to the measured phenotypic characteristics, enumeration in low dimensions allows us to examine this relationship directly. Second, knowledge of the nature of this relationship over the entire state space can aid in the evaluation of heuristic optimization methods that best suit CPM and address the questions of interest.

To conduct an evaluation of the state space we must select an objective function, which is a statistical function that provides a measure of the value of each set of genotypic partitions. The sum of the squared deviations of the phenotype means of the partitions from the overall sample mean (SS_K) is a natural choice for an objective function of a set of genotypic partitions. The value of this measure increases as the individuals within genotypic partitions become more phenotypically similar and as the phenotypic differences between partitions increase. A disadvantage of the partition sum of squares is that it will tend to improve as k increases, favoring the division of genotypes into a greater number of partitions such that $k = g_M$. The bias corrected estimate of genotypic variance [1] is used to compensate for this bias. It is written as

$$s_K^2 = \sum_{i=1}^{k} \frac{n_i \left(\overline{Y}_i - \overline{Y}\right)^2}{n} - \frac{k-1}{n} \sum_{i=1}^{k} \sum_{j=1}^{n_i} \frac{\left(\overline{Y}_{ij} - \overline{Y}_i\right)^2}{n-k} \tag{2}$$

$$= \frac{SS_K}{n} - \frac{k-1}{n} MS_W, \tag{3}$$

where n is the sample size, $\overline{Y}$ is the sample grand mean, n_i is the sample size and $\overline{Y}_i$ is the mean of partition i, Y_{ij} is the phenotype of the jth individual in the ith partition, and MS_W is the mean squared estimate of the phenotypic variability among individuals within genotypic partitions. This statistic was derived using the expected mean squares from a standard one way analysis of variance model to obtain an unbiased estimator of the variance associated with a set of genotypic partitions. This correction has the effect of penalizing the scaled partition sum of squares by a quantity that increases with k if the estimate of s^2 does not decrease as additional partitions are considered. For comparative purposes the proportion of variability explained by a set of genotypic partitions is preferred over s_K^2. The proportion, denoted pv_K, is computed as

$$pv_K = \frac{s_K^2}{s_K^2 + MS_W} = \frac{s_K^2}{s_P^2}, \tag{4}$$

where s_P^2 is the total phenotypic variance.

In this combinatorial setting, the number of sets considered can potentially be enormous, even for moderate values of l and $m = 2$. For this reason, some filter is needed to decide which sets of genotypic partitions should be retained for further consideration in Step 2 of the CPM. Many criteria could be used in setting this filter, including criteria based on test–wise significance [e.g. $MS_K/MS_W > F(\alpha - 0.005; k-1, n-k)$], biological significance (e.g. $pv_K > 0.05$), or some

proportion of all the sets considered (e.g. the top 1%, the top 100, or simply the overall best). In the applications described in Section 3, we use a cutoff based on the F–statistic.

2.3 Step 2: Validating the Selected Sets of Genotypic Partitions

The second step in the CPM is to validate the retained sets of genotypic partitions. We are currently using multi–fold cross validation [13], a common method of model validation. In our applications, we set the number of cross validation folds to ten [6]. This method simulates the process of going back into the population of inference and collecting an independent sample with which to validate the constructed models. To reduce the possibility that a particular random assignment of the sample into 10 cross validation groups might favor one set over another, the random assignment and cross validation was repeated 10 times, and the resulting cross validated within partition sums of squares were averaged for each set.

The cross validated estimate of the trait variability within each genotypic partition ($SS_{W,CV}$) was used to judge the predictive ability of a given set of genotypic partitions. For consistency and comparability, we used $SS_{W,CV}$ and $SS_{K,CV} = SS_{Total} - SS_{W,CV}$ in place of SS_W and SS_K in equations 3 and 4 to calculate a cross validated proportion of variability explained, denoted as $pv_{K,CV}$. The larger the value of $pv_{K,CV}$ is the more predictive set K is said to be. Note that $SS_{W,CV}$ must be greater than or equal to SS_W which implies that $pv_{K,CV}$ will be less than or equal to pv_K. Also, the expectations of the estimates of $pv_{K,CV}$ in this cross validated setting are not known. Further work is required to better understand the properties of this estimator under cross validation conditions. We use this statistic for evaluating the relative predictive ability of each set of genotypic partitions.

2.4 Step 3: Select the 'Best' Sets of Genotypic Partitions and Make Inferences about the Relationship between Phenotypic and Genotypic Variation

Steps 1 and 2 provide a strategy for identifying sets of genotypic partitions that predict variation in quantitative trait levels. The third step in this approach is to select some subset of the validated sets of genotypic partitions on which inferences can be made. Rather than select a single winner on the basis of the cross validation (Step 2) we prefer to scrutinize a subgroup containing multiple sets of partitions that may be almost as predictive as the overall most predictive set. With this subgroup we can ask questions such as: 1) How much trait variability does each of the selected sets explain? 2) Which combinations of loci are involved? 3) How do the genotype–phenotype relationships in one combination compare with another? and 4) How do the genotype–phenotype relationships compare with the relationships expected using traditional multi–locus genetic models? The subjective decision of which subgroup is selected to address such questions is left to the investigator.

3 Example Application

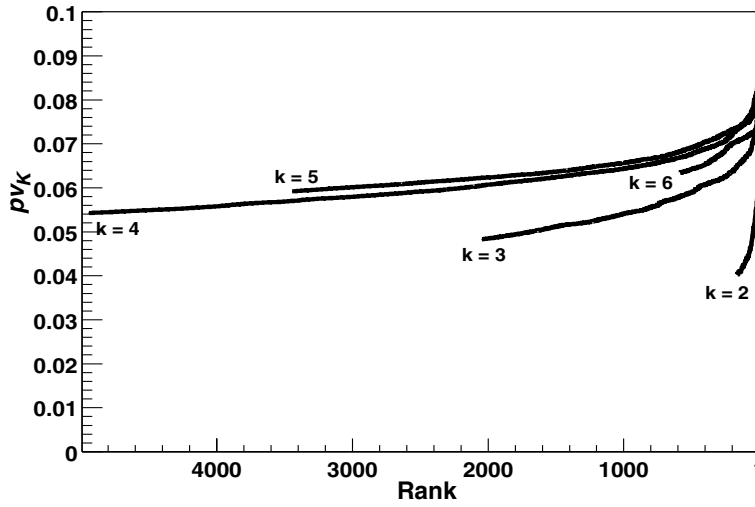
To illustrate the combinatorial partitioning method we present an application to identify sets of genotypic partitions of two locus combinations of 18 diallelic loci (designated as ID1, PstI, SstI, IDB, XbaI, MspI, EcRI, E112, E158, Pvu2, Hnd3, BstI, TaqI, StuI, Hnc2, Ava2, P55, and P192) located in six atherosclerosis susceptibility gene regions (*APOB*, *PON1*, *LPL*, *APOA1–C3–A4*, *LDLR*, *APOE*) that predict inter–individual variability in plasma triglycerides (Trig) levels, a known atherosclerotic risk factor. The sample of 241 women (age 20 to 60 yrs) used here comes from the Rochester Family Heart Study (RFHS), a population–based study of multi–generation pedigrees sampled without regard to health status from the Rochester, Minnesota population [14]. In accordance with common practice in the study of lipid metabolism, we applied a natural logarithm–transformation (lnTrig) and we adjusted lnTrig using a polynomial regression model that included age, height, and weight, each to the third order, as well as first order terms for waist–to–hip ratio and body mass index.

As discussed above in Section 2.2, we limited the state space in this application to all possible sets of genotypic partitions ($K : k = 2, 3, \ldots, g_M$) defined by all possible pairs ($m = 2$) of the $l = 18$ diallelic loci. Note that many of the loci used in the application have alleles that have a low relative frequency. When two such loci are considered in combination, the number of two–locus genotypes observed in a data set of this size is commonly smaller than nine (3^2). We set five as the lower bound for the number of individuals that must be present to constitute a valid partition. Sets containing a partition with fewer than five individuals were excluded from further evaluation. To filter the potentially millions of valid sets of genotypic partitions for validation in Step 2, we selected a criterion based on the test-wise significance of each set. All sets with an F statistic that exceeded the 0.9999 quantile of the F distribution corresponding to each k considered, i.e. $F(\alpha = 0.0001; k-1, n-k)$, were retained for further validation.

The application of the CPM to lnTrig in females resulted in the consideration of 880,829 sets of genotypic partitions. This number is considerably lower than the expected 3,235,338 possible sets associated with all two-locus combinations of 18 diallelic loci. There are two reasons for this: 1) most of the pairwise combinations of loci resulted in fewer than nine two-locus genotypes, and 2) approximately 10% of the possible sets contained partitions that contained less than 5 individuals and were therefore not evaluated. When the CPM was initially applied with a retention cutoff corresponding to $\alpha = 0.01$, more than 50,000 sets of genotypic partitions were retained. Setting the cutoff corresponding to $\alpha = 0.0001$ resulted in the retention of 11,268 sets (1.3%) with k ranging from 2 to 6. The proportion of variability explained by each of the retained sets is (in rank order, worst to best) shown in Figure 3.

The sets that explain the greatest proportion of variability by k were from $pv_K = 0.069$ for $k = 2$ up to $pv_K = 0.086$ for $k = 4$, a range of 0.017. There are a few notable features of the plot of these results given in Figure 3. First, the number of sets retained varies greatly by k and is most likely a function of the small size of the sample used in this example. Second, the shape of the

Fig. 3. The proportion of lnTrig variability explained in females by the 11,268 retained sets of genotypic partitions. Sets sorted by pv_K.

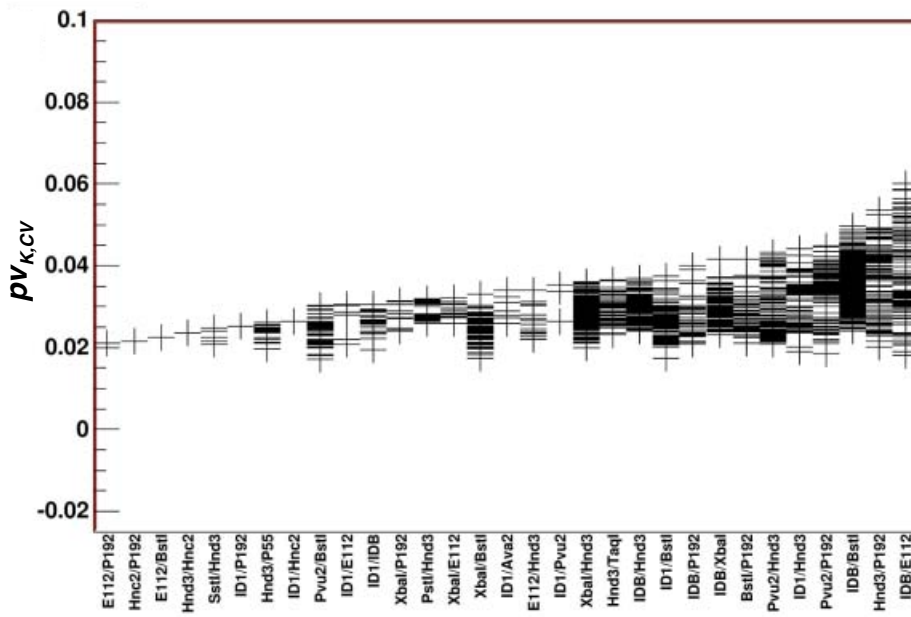


curves for each k show that there are many sets near the cutoff that explain approximately the same proportion of variability and relatively few sets that explain substantially more. It should be pointed out that when these 18 loci were analyzed independently, the greatest proportion of lnTrig variability explained by a single locus was 0.035.

After performing 10–fold cross validation on all retained sets we found that the most predictive set overall corresponds to $k = 3$. In Figure 4 we illustrate the cross validated estimates of the proportion on variability explained for all sets of size $k = 3$. The y–axis corresponds to the cross validated proportion of variability for each of the retained sets where $k = 3$. The x–axis corresponds to to each pair of loci with a retained set of genotypic partitions, sorted on the basis of the maximum value of $pv_{K,CV}$ for each locus pair. Each '+' on the graph represents the value of $pv_{K,CV}$ for a particular set within the indicated pair of loci. From this plot we see that the locus pair {IDB, E112} with $pv_{K,CV} = 0.060$ ($pv_K = 0.084$) had the best overall predictive ability. Figure 4 also illustrates several important features about the solution space of this type of application. First, the 11,268 retained sets are only distributed among 31 of the possible 144 pairs of loci. Second, most pairs of loci have many sets associated with partitions that predict similar levels of Trig variability. Lastly, this figure illustrates that the CPM finds sets with good cross validation properties. For geneticists interested in the specifics of the genotype–phenotype relationship designated by a particular set of CPM solutions, we have developed a battery of data visualization procedures to

examine 1) the specific genotypic partitions, 2) the phenotypic means associated with a particular set or sets of solutions, 3) intersections between sets for a particular pairs of loci, or 4) intersections between sets across pairs of loci.

Fig. 4. The cross validated proportion on InTrig variability predicted in females by the retained sets of genotypic partitions with three genotypic partitions ($k = 3$). Sets are sorted by pairs of loci and $pv_{K,CV}$.



4 Discussion

Most traits of interest in human genetics have a complex, multifactorial etiology. Interindividual variation in these traits is influenced by variation in many factors, including interactions among many variable genetic loci and exposures to many environmental agents [11]. However, most studies of the genetic basis of complex traits ignore this reality, relying on analytical methods that are applicable only to Mendelian traits because they model the relationship between interindividual trait variability and genotype variability one locus at a time.

The motivation for the CPM is to develop a search algorithm through large multi–locus genotypic spaces to identify partitions that predict phenotypic variability that is not constrained by the limitations of generalized linear models.

The results of an application of the CPM to identifying genetic predictors of interindividual variation in triglyceride levels illustrates some of its features. A major advantage of the CPM is its ability to examine and estimate statistical relationships between trait variability and genotypic variability defined by more than one locus at a time. A distinct advantage of the CPM over traditional linear regression approaches is that the multi–locus genotype state space to be considered in the selection of genotypic partitions is not defined a priori by loci that have separate, independent phenotypic effects on trait variability. That is, this method allows for the non–additive relationships between loci to be the primary dominate causation of trait variability because the possible partitions are not constrained a priori to include only those loci that display average phenotypic effects when considered separately.

As with any new analytical approach, there are many aspects of the CPM that remain research issues. First, we are working on extending these methods to much higher dimensional genotype spaces using simulated annealing and genetic algorithms. Second, we are evaluating other methods of validating the selected sets of genotypic partitions to guard against characterizing artifacts that are unique to a particular data set. Finally, we are modifying the CPM to include prediction of variation in categorical traits (ordered and unordered) and the multivariate properties of vectors of quantitative traits to broaden its applicability to sorting out and characterizing those DNA variable sites that predict interindividual variation in particular traits in particular environmental strata within particular populations.

References

1. E. Boerwinkle and C. F. Sing, *The use of measured genotype information in the analysis of quantitative phenotypes in man. III. Simultaneous estimation of the frequencies and effects of the apolipoprotein E polymorphism and residual polygenetic effects on cholesterol, betalipoprotein and triglyceride levels*, Ann Hum Genet **51** (1987), 211–26.
2. L. Comtet, *Advanced combinatorics: The art of infinite expansions*, Reidel Pub. Co., Boston, MA, 1974.
3. D. E. Goldberg, *Genetic algorithms in search, optimization, and machine learning*, Addison–Wesley, Reading, MA, 1989.
4. D. L. Hartl and A. G. Clark, *Principles of population genetics*, third ed., Sinauer Associates, Sunderland, MA, 1997.
5. S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, *Optimization by simulated annealing*, Science **220** (1983), 671–680.
6. R. Kohavi, *A study of cross–validation and bootstrap for accuracy estimation and model selection*, Proc Int Joint Conf Artificial Intel, 1995, pp. 1137–1143.
7. B.-H. Liu, *Statistical genomics: Linkage, mapping, and QTL analysis*, CRC Press, Boca Raton, 1998.
8. M. R. Nelson, *A combinatorial partitioning method to identify multi–genic multi–locus models that predict quantitative trait variablility.*, Ph.D. thesis, University of Michigan, Department of Human Genetics, 1999.

9. M. R. Nelson, S. L. R. Kardia, R. E. Ferrell, and C. F. Sing, *A combinatorial partitioning method to identify multi–locus genotype partitions that predict quantitative trait variation.*, (Submitted).
10. D. A. Nickerson, S. L. Taylor, K. M. Weiss, A. G. Hutchinson, J. Stengard, V. Salomaa, E. Vartiainen, E. Boerwinkle, and C. F. Sing, *DNA sequence diversity in a 9.7 kb region of the human lipoprotein lipase gene.*, Nat. Genet. **19** (1998), 233–240.
11. C. F. Sing, M. B. Haviland, and S. L. Reilly, *Genetic architecture of common multifactorial diseases.*, Ciba Found Symp **197** (1996), 211–229.
12. G. S. Stent, *Genetics: An introductory narrative*, W.H. Freeman, San Francisco, 1971.
13. M. Stone, *Cross–validation: A review*, Math Operationsforsch Statist, Ser Statistics **9** (1978), 127–139.
14. S. T. Turner, W. H. Weidman, V. V. Michels, T. J. Reed, C. L. Ormson, T. Fuller, and C. F. Sing, *Distribution of sodium–lithium countertransport and blood pressure in Caucasians five to eighty–nine years of age*, Hypertension **13** (1989), 378–391.

Compact Suffix Array*

Veli Mäkinen

Department of Computer Science, P.O Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, Finland
vmakinen@cs.Helsinki.FI

Abstract. *Suffix array* is a data structure that can be used to index a large text file so that queries of its content can be answered quickly. Basically a suffix array is an array of all suffixes of the text in the lexicographic order. Whether or not a word occurs in the text can be answered in logarithmic time by binary search over the suffix array. In this work we present a method to compress a suffix array such that the search time remains logarithmic. Our experiments show that in some cases a suffix array can be compressed by our method such that the total space requirement is about half of the original.

1 Introduction

The string-matching problem considered in this paper is to determine for a pattern P and a text T, whether there is a substring inside text T that matches pattern P. If the same text is queried several times, it is worthwhile to build a data structure (index) for the text so that queries can be answered quickly. Optimal inquiry time is $O(|P|)$, as every character of P must be checked. This can be achieved by using the *suffix tree* [13,10,11] as the index. In a suffix tree every suffix of the text is presented by a path from the root to a leaf.

The suffix array [9,8] is a reduced form of the suffix tree. It represents only the leaves of the suffix tree in the lexicographic order of the corresponding suffixes. Although it saves space, the query times get worse. The tree traversal is simulated by a binary search; a branch in a suffix tree can be selected in constant time (or at least in time proportional to the size of the alphabet), but in a suffix array a branch must be searched by a binary search. The search time with the suffix array is therefore $O(|P| \log |T|)$. Manber and Myers [9] have shown that by saving additional information about consecutive suffixes of the binary search in the suffix array, it is possible to search pattern P in time $O(|P| + \log |T|)$.

A suffix tree contains a lot of repetitive information in different tree branches. The uncompacted version of the suffix tree called *suffix trie* can be transformed into a structure called *suffix automaton* or *DAWG (Directed Acyclic Word Graph)* [2] by combining its similar subtrees. If the method is applied straight to the suffix tree we get a *Compact DAWG* (*CDAWG*) [3,4,5,6]. This kind of combining can also be applied to the suffix array. As the suffix array

* A work supported by the Academy of Finland under grants 44449 and 22584.

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 305–319, 2000.

represents only the leaves of the suffix tree, there are less possibilities for combining.

In this paper we present a modification of the suffix array called *compact suffix array (CSA)*. The CSA can be considered as an analogue to the CDAWG; as the CDAWG is a minimized suffix tree, the CSA is a minimized suffix array. Despite the analogue, these two structures are not in one-to-one correspondence to each other, because there is different kind of repetitiveness in the suffix tree than in the suffix array due to different data representation.

A feature of the CSA is that the compression does not change the asymptotic search times; searching pattern P from the CSA takes still time $O(|P|+\log|T|)$. Moreover, there is a trade off between the compression performance and the search time in our algorithms. The trade off is controlled using certain constant parameters.

In Section 2 we define the CSA. In Section 3 we give an algorithm to construct a CSA in linear time from the suffix array. In Section 4 we give a general algorithm to search pattern P from the CSA in average time $O(|P| + \log|T|)$. In Section 5 we consider the space requirement and the trade off between compression and search times, and give a practical search algorithm that works in $O(|P|+\log|T|)$ time in the worst case. Experimental results are given in Section 6 both about the size of the structure and the search times. Conclusions are given in Section 7.

2 Compact Suffix Array

Let Σ be an ordered alphabet and $T = t_1 t_2 \ldots t_n \in \Sigma^*$ a text of length $n = |T|$. A *suffix* of text T is a substring $T_{i\ldots n} = t_i \ldots t_n$. A suffix can be identified by its *starting position* $i \in [1 \ldots n]$. A *prefix* of text T is a substring $T_{1\ldots i}$. A prefix can be identified by its *end position* $i \in [1 \ldots n]$.

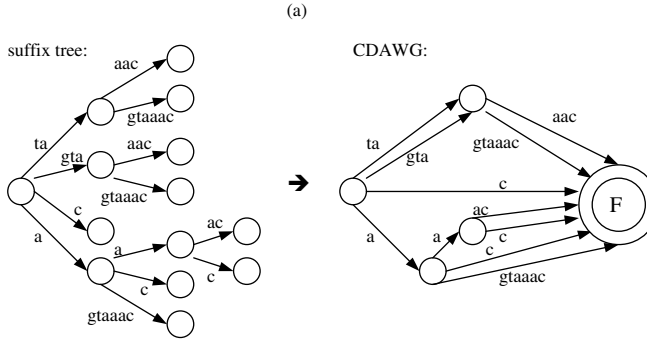
Definition 1. *The* suffix array *of text T of length $n = |T|$ is an array $A(1 \ldots n)$, such that it contains all starting positions of the suffixes of the text T such that $T_{A(1)\ldots n} < T_{A(2)\ldots n} < \ldots < T_{A(n)\ldots n}$, i.e. the array A gives the lexicographic order of all suffixes of the text T.*

The idea of compacting the suffix array is the following: Let $\Delta > 0$. Find two areas $a \ldots a+\Delta$ and $b \ldots b+\Delta$ of A that are repetitive in the sense that the suffixes represented by $a \ldots a+\Delta$ are obtained in the same order from the suffixes represented by $b \ldots b+\Delta$ by deleting the first symbol. In other words, $A(a+i) = A(b+i)+1$ for $0 \leq i \leq \Delta$. Then replace the area $a \ldots a+\Delta$ of A by a link, stored in $A(a)$, to the area $b \ldots b+\Delta$. This operation is called a *compacting operation*. Similarly define *uncompacting operation* such that a link to the area $b \ldots b+\Delta$ is replaced by suffixes $a \ldots a+\Delta$.

Basically a compacting operation on a suffix array corresponds to the operation of combining subtrees of a suffix tree to form a CDAWG. This is visualized in Fig. 1(a). Figure 1(b) shows the same operation with a suffix array. As can

be seen in Fig. 1, conversions from a suffix tree to a CDAWG, and from a suffix array to a CSA, are slightly different.

In Fig. 1(b), the link at entry 7 of the compact suffix array is to the area that itself is represented by a link. This means that to represent the starting point of the linked area, we need in fact two values: value x at entry a to represent the link to an existing entry b of the compacted array, and if entry b contains a link to another entry c, we need value Δx to tell which of the entries represented by the link at entry b we are actually linking to. In Fig. 1(b), we have $(x, \Delta x) = (5, 1)$ at the entry 7. The value $\Delta x = 1$ tells that the link at entry 7 points to the second entry of the area that is created by uncompacting link $(3, 0)$ at entry $x = 5$. In principle, we would need a third value to denote the length of the area represented by a link, but a link can also be uncompacted without it (see Sect. 4). However, we still need a boolean value for each entry to tell whether it contains a link or a suffix of the text.



(b)

text:	g	t	a	g	t	a	a	a	c
pos:	1	2	3	4	5	6	7	8	9

suffix array:

entry	suffix	pos
1	aaac	6
2	aac	7
3	ac	8
4	agtaaac	3
5	c	9
6	gtaaac	4
7	gtagtaaac	1
8	taaac	5
9	tagtaaac	2

→

entry	suffix	pos
1	aaac	6
2	aac	7
3	ac	8
4	agtaaac	3
5	→	(3)
6		
7	gtagtaaac	1
8	→	(6)
9		

→

compact suffix array:

entry	suffix	pos
1	aaac	6
2	aac	7
3	ac	8
4	agtaaac	3
5	→	(3,0)
6	gtagtaaac	1
7	→	(5,1)

Fig. 1. (a) A suffix tree and a CDAWG for text "gtagtaaac" (b) A suffix array for text "gtagtaaac" and construction of a compact suffix array. Arrows in columns with title *suffix* tell that the value(s) in parenthesis in the next column denote a link to another area of the array.

Because the links can be chained, we have to extend the definitions of the compacting and uncompacting operations. A *compacting operation* compacts area $b \ldots b + \Delta$ such that some parts of the area can already be compacted using a compacting operation. A *uncompacting operation* retrieves area $a \ldots a + \Delta$ from an area $b \ldots b + \Delta$ that is compacted using a compacting operation.

Definition 2. *The* compact suffix array (CSA) *of text T of length $n = |T|$ is an array $CA(1 \ldots n')$ of length n', $n' \leq n$, such that for each* entry $1 \leq i \leq n'$, *$CA(i)$ is a triple $(B, x, \Delta x)$, where B is a boolean value to determine whether x denotes a suffix of the text ($B = false$), or whether x and Δx denote a link to an area obtained by a compacting operation ($B = true$). If $B = true$, $x + \Delta x$ denotes the entry of CA where the linked area starts* after *uncompacting $CA(x)$ with a uncompacting operation.*

The relationship between the suffix array and the CSA can be characterized by the fact that the suffix array of the text T can be constructed by uncompacting the links of the CSA. The *optimal* CSA for T is such that its size n' is the smallest possible.

We use $CA(i).B$, $CA(i).x$, and $CA(i).\Delta x$ to denote the value of B, x, and Δx at entry i of the array CA.

3 Constructing a CSA from a Suffix Array

The idea of the algorithm to construct a CSA from a suffix array is shown in Fig. 1(b). First, the areas that can be replaced by a link are marked in the original array. This is done by advancing from the first entry of the array to the last, and finding for each suffix $A(i)$ the entry x that contains suffix $A(i)+1$. The entry x of suffix $A(i)+1$ could be found by a binary search, but it can also be found in constant time by constructing the reverse A_r of array A in advance; the array A_r is easily created in linear time by setting $A_r(A(i).x) = i$ for all $1 \ldots n$. After the entry x of suffix $A(i)+1$ is found, the repetitiveness is checked between the areas starting at i and x, and in the positive case, the area starting at x is marked to be replaced by a link to the area starting at i. The second step is to construct the compacted array by replacing marked areas by links, and changing links to refer to the entries of the compact array. Both steps, that are shown in Fig. 2, work clearly in linear time.

In *MarkLinksToSuffixArray*, the reason for the conditions in the line (8) is to prevent *cyclic links*. For example a simple cyclic link would be that area $\mathcal{A}$ is represented by a link to area $\mathcal{B}$ and area $\mathcal{B}' : \mathcal{B}' \cap \mathcal{B} \neq \emptyset$ is represented by a link to area $\mathcal{A}' : \mathcal{A}' \cap \mathcal{A} \neq \emptyset$ (for example, the CSA of text "ababa" could contain such). Using the array *IsPointed* we choose the areas that can be linked and prevent cyclic links. For example, with the simple cyclic link we choose so that area $\mathcal{A}$ can be replaced by a link to area $\mathcal{B}$, but area $\mathcal{B}'$ can not be replaced by a link to area $\mathcal{A}'$. Unfortunately, this prevents all the other links to the area $\mathcal{A}'' : \mathcal{A}'' \cap \mathcal{A} \neq \emptyset$, too, and results in a structure that is not optimally compressed. Thus, the algorithm does not construct the optimal CSA.

```
MarkLinksToSuffixArray(in A,in n,out A',out n'):
(1)  for each 1 ≤ i ≤ n do A'(i).x ← −1;
(2)  for each 1 ≤ i ≤ n do A_r(A(i).x) ← i;
(3)  i ← 1; n' ← 1;
(4)  while i ≤ n do begin
(5)       x ← A_r(A(i).x + 1);
(6)       j ← 0;
(7)       while x + j < n and i + j < n and
(8)                 x + j ≠ i and i + j ≠ x and not IsPointed(x + j) and
(9)                 A(x + j).x ← A(i + j).x + 1 do begin
(10)           if A'(i + j).x < 0 then A'(i + j) ← A(i + j);
(11)           if j > 0 then begin
(12)                A'(x + j) ← (true, i, j);
(13)                IsPointed(i + j) ← true end;
(14)           j ← j + 1 end;
(15)      if j > 1 then begin {update first linked suffix}
(16)           IsPointed(i) ← true;
(17)           A'(x) ← (true, i, 0);
(18)           i ← j + 1 end
(19)      else begin {copy suffix}
(20)           if A'(i).x < 0 then A'(i) ← A(i);
(21)           i ← i + 1 end;
(22)      n' ← n' + 1 end;

CompactMarkedArray(in A',in n,in n',out CA):
(1)  j ← 0;
(2)  for each 1 ≤ i ≤ n do begin
(3)       if A'(i).Δx = 0 then begin
(4)            CA(j) ← A'(i);
(5)            j ← j + 1;
(6)            mapiToj(i) ← j end
(7)       else mapiToj(i) ← j − 1 end;
(8)  for each 1 ≤ i ≤ n' such that CA(i).B = true do
(9)       CA(i) ← (true, mapiToj(i), A'(CA(i).x).Δx);
```

Fig. 2. Algorithm to construct a CSA. For simplicity, arrays A and A' use the same representation as array CA.

4 Searching

The proposed compacting method maintains the important property that the CSA still allows a fast search. The search algorithm works in two phases. First, a binary search is performed over the entries that do not contain a link. This search gives us an area of the array that must be checked. Secondly, checking is done by uncompacting every link of the area, and checking the suffixes formed

this way against the pattern. To be able to make a binary search over the CSA, we need some modifications to its original definition.

Definition 3. *The* binary searchable compact suffix array (BSCSA) *of text* T *is an array* $CA(1 \ldots n')$ *which is as in the definition of the CSA with following limitations:* $CA(1).B = CA(n').B = false$ *and for each* $i \in [2 \ldots n' - 1]$ *if* $CA(i).B = true$ *then* $CA(i+1).B = false$.

On other words, only every other entry of a BSCSA can be a link. The algorithm *MarkLinksToSuffixArray* in Fig. 2 can be easily modified to handle this limitation. From this on, we assume that a CSA is actually a BSCSA. The search algorithm that utilizes this property is shown in Fig. 3.

```
SearchFromCSA(in P,in CA,in T,in n,in n',out matches):
    {Make two binary searches to find left limit LL and right limit RR}
    {Check the area LL..RR}
(1) if (LL = RR and CA(LL).B = true) or (LL < RR) then
(2)     matches ← matches + CheckArea(CA, LL, RR, P)
(3) else if LL = RR then matches ← 1;

CheckArea(in CA, in LL, in RR, in P):
(1) for each LL ≤ i ≤ RR do begin
(2)     if CA(i).B = true then begin
(3)         LimitSet ← LimitSet ∪ {i};
(4)         UCA ← UnCompactLink(CA, i);
(5)         LimitSet ← LimitSet \ {i};
(6)         if i = LL or i = RR then UCA ← CopyEqual(UCA, P);
(7)         matches ← matches + |UCA| end
(8)     else matches ← matches + 1 end
(9) return matches;
```

Fig. 3. Search algorithm for a CSA

Algorithm *SearchFromCSA* makes two binary searches to find left and right limits of the area to be checked. The only difference compared to the normal binary search, is the special treatment of the entries containing a link; if the middle of the area is a link, we can choose the entry on its left side instead. Also, if the last middle point of the search loop is a link, we have to make some additional checkings. After finding left and right limits, we call function *CheckArea* to calculate the sum of the matches.

The reason for the use of the *LimitSet* in *CheckArea* is to prevent function *UnCompactLink* from trying to uncompact cyclic links (i.e. prevents from entering into the areas of the array that cannot be part of the link being uncompacted). Function *UnCompactLink* returns an array that represents the linked area of

the original suffix array. The resulting array does not necessarily match totally to the pattern, so function *CopyEqual* returns the matching part. Finally, the number of the matches is incremented by the length of the uncompacted array.

The difficult part of the search algorithm is the function *UnCompactLink*. As the links between the areas of the CSA can be chained, it is natural to write a recursive function to uncompact a link. We first write a procedure that only handles the situation where the linked area does not contain links (i.e. the situation at the end of the recursion). The resulting algorithm is shown in Fig.4.

```
SimpleUnCompactLink(in CA, in x):
(1)  y ← CA(x).x + CA(x).Δx;
(2)  prev ← CA(x − 1).x;
(3)  i ← 0;
(4)  while T_{CA(y+i).x+1...n} < T_{CA(x+1).x...n} and
                T_{CA(y+i).x+1...n} > T_{prev...n} do begin
(5)      prev ← CA(y + i).x + 1;
(6)      UCA(i + 1).x ← prev;
(7)      i ← i + 1 end
(8)  return UCA;
```

Fig. 4. The simple case of the algorithm to uncompact a link.

It is easy to see that the simple case of the *UnCompactLink* works right in its proper environment: The conditions in the while-loop are for detecting the end of the linked area. If suffix $T_{CA(y+i).x+1...n}$ is greater than previously uncompacted suffix and smaller than $T_{CA(x+1).x...n}$, it must belong to the area being uncompacted. Otherwise it would have been in the place of the suffix $T_{CA(x+1).x...n}$ in the original suffix array, and would have ended up to be in $CA(x + 1)$, in which it is not.

Now we can write the full recursive version of the same algorithm, which can handle the chained links; the algorithm is shown in Fig. 5.

In *UnCompactLink*, procedure *AddAndCheck(A,B,inc)* adds the area B to the end of area A and adds *inc* to each added suffix. It also checks whether the end of the linked area is reached with the same conditions as in *SimpleUnCompactLink*. The *LimitSet* is to prevent *UnCompactLink* from trying to uncompact cyclic links; the construction algorithm of Sect. 3 does not create cyclic links, but the uncompacting phase could enter into a cycle while searching the end of the linked area.

The only significant difference between the recursive version and the simple case of the *UnCompactLink* is the treatment of the value Δx: in the recursive version the start of the linked area is not known, and all the links starting after entry x must be checked. When the start of the linked area is found, the end of the area is checked almost similarly in both versions. In the recursive version

checking is done in parts: entries that contain a suffix are handled as in the simple case, and entries that contain a link are first uncompacted and then the suffixes in the uncompacted area are handled as in the simple case.

UnCompactLink(in CA, in x):
(1) $y \leftarrow CA(x).x$; $\Delta x \leftarrow CA(x).\Delta x$; $i \leftarrow 0$;
(2) **while** $(y+i) \notin LimitSet$ **do begin**
(3) **if** $CA(y+i).B = true$ **then begin** $\{y+i$ contains a link$\}$
(4) $LimitSet \leftarrow LimitSet \cup \{y+i\}$;
(5) $UCA \leftarrow$ UnCompactLink$(CA, y+i)$;
(6) $LimitSet \leftarrow LimitSet \setminus \{y+i\}$;
(7) **if** AddAndCheck$(Area, UCA(\Delta x+1 \ldots |UCA|),1)$ **then break**;
(8) $\Delta x \leftarrow \max(0, \Delta x - |UCA|)$) **end**
(9) **else begin** $\{y+i$ contains a suffix$\}$
(10) **if** $\Delta x > 0$ **then** $\Delta x \leftarrow \Delta x - 1$
(11) **else if** AddAndCheck$(Area, CA(y+i).x, 1)$ **then break end**;
(12) $i \leftarrow i+1$ **end**;
(13) **return** $Area$;

Fig. 5. Recursive algorithm to uncompact a link.

4.1 The Search Time

The first part of the algorithm *SearchFromCSA* of Fig. 3, that finds the area to be checked, works in time $O(|P| \log n')$, as the pattern P is compared to each suffix of the binary search. We can, however, store LCP values for the entries that contain suffixes (see Sect. 5.1). With this information, the first part of the algorithm *SearchFromCSA* can be implemented to work in time $O(|P| + \log n')$.

The time needed for the recursive subprocedure *UnCompactLink* is presented in the following lemma:

Lemma 4. *The average time needed to uncompact a link of CSA is* $O((\frac{2n-n'}{n'})^2 \log n)$, *where* $n = |T|$ *and* n' *is the length of the CSA.*

Proof. The structure of the CSA guarantees that at most every other entry contains a link. We start the average case analysis by considering the worst case, i.e. every other entry contains a link. This gives us an equation $\frac{n'}{2}x + \frac{n'}{2} = n$ to the average length of the linked area x. The value $\frac{n'}{2}x = n - \frac{n'}{2}$ is the sum of the lengths of the linked areas. If we assume that every entry of the original suffix array is equally likely to be the start of a linked region, we can write the probability of a link path of length k as $\mathbb{P}$("*link path of length* k")$= (\frac{n-\frac{n'}{2}}{n})^{k-1}(1 - \frac{n-\frac{n'}{2}}{n}) = (1 - \frac{n'}{2n})^{k-1}\frac{n'}{2n}$ (i.e. $k-1$ times a link leads to

an entry containing a link and the last link leads to an entry containing a suffix). This is a geometric probability distribution and its expectation value is $\mathbb{E}(\text{"}\textit{link path of length } k\text{"}) = \frac{2n-n'}{n'}$.

As each link represents on the average $\frac{2n-n'}{n'}$ suffixes and the length of the link path to each such suffix is on the average $\frac{2n-n'}{n'}$, and every such suffix is compared to some other suffixes (time $O(\log n)$), the total time used to uncompact the suffixes presented by a link is on the average $O((\frac{2n-n'}{n'})^2 \log n)$.

In addition to uncompacting the suffixes represented by a link, the algorithm of Fig. 5 spends time finding the start of the linked area. This must be done only when an area is linked inside an already linked area. In such cases, there is on average $\frac{2n-n'}{2n'}$ suffixes to uncompact to find the start of the linked area, because in the average case a link points in the middle of the already linked area. As the length of the link path is on the average $\frac{2n-n'}{n'}$, it takes $O(\frac{1}{2}(\frac{2n-n'}{n'})^2 \log n)$ to find the start of the linked area.

The total time to uncompact a link is therefore $O(\frac{3}{2}(\frac{2n-n'}{n'})^2 \log n)$. □

Theorem 5. *The average time needed to search all occurrences of pattern P from CSA of text T is $O((\frac{2n-n'}{n'})^2(|P| + k \log n))$, where $n = |T|$, n' is the length of the CSA, and k is the number of occurrences.*

Proof. The binary search of the area of possible occurrences takes time $O(|P| + \log n')$. If k is the number of occurrences, there will be at most $k/3$ links to uncompact (when each link represents the minimum two suffixes). Therefore the result of Lemma 4 must be multiplied by $k/3$. In addition, the suffixes represented by the first and the last links must be checked against the pattern P (other suffixes already match due to the binary search). The total search time is the sum of these three factors. □

Theorem 6. *The average time needed to determine whether there is an occurrence of pattern P inside text T, using CSA is $O((\frac{2n-n'}{n'})^2(|P| + \log n))$, where $n = |T|$ and n' is the length of the CSA.*

Proof. There is at most one link to uncompact, because if there were two (or more) links to uncompact, there would be a suffix between them that matches P. Therefore we get the time $O((\frac{2n-n'}{n'})^2(|P| + \log n))$. □

If $n' = \Theta(n)$, the value $(\frac{2n-n'}{n'})^2$ can be considered as a constant, and the results of Lemma 5 and Theorem 6 can be improved. This is the case for typical texts, as can be seen from the experimental results in Sect. 6.

5 Practical Implementation

5.1 Space Requirement

Although a CSA has fewer entries than the corresponding suffix array, an entry of a CSA takes more space than an entry of a suffix array. An entry of a suffix array

contains only one integer value, while in a CSA there are two integer values and one boolean value in an entry (triple $(B, x, \Delta x)$). This can be overcome by the fact that we can limit the length of the linked region at the cost of slightly smaller compression. If we limit the length of the linked region so that $0 \leq \Delta x \leq 255$, we can use one byte to represent the value of Δx[1]. Also the boolean value B can be embedded into the value x at the cost of only being able to handle texts that are smaller than 2^{31} bytes.

Fact 7. *An entry of a CSA takes* $5/4$ *times the space of an entry of a suffix array.*

Since the value Δx is only needed for the entries that contain a link, the wasted space can be used to store the LCP information of consecutive suffixes of the binary search. As each suffix has two possible preceding suffixes in a binary search, there are two values to be saved for each suffix. There was suggested in [9], that saving only the maximum of these two values is adequate, because the other value can be achieved during the search. However, we need one bit to tell which value is actually saved. There remains 7 bits to save the LCP information, i.e the LCP values must be in the range $0 \ldots 127$. In an average case this is an acceptable limitation[2]. The special case when a LCP value does not fit into a byte can be handled separately (costing in search times).

Fact 8. *An entry of a suffix array with the LCP information of consecutive suffixes of the binary search takes the same space as an entry of a CSA with the LCP information.*

5.2 An Implementation with Improved Worst Case Bounds

Although the algorithm in Fig. 5 has reasonably good average case bounds, its worst case behavior is a problem; finding the start of the linked area may require uncompacting the whole array.

Therefore we make the following modifications to the structure of the CSA and construction algorithm of Fig. 2:

- The length of the linked area is added to the structure of the CSA. That is, each $CA(i)$ in Definition 2 contains $(B, x, \Delta x, \Delta)$, where Δ is the length of the linked area (i.e. $\Delta = \Delta_2 + 1$, where Δ_2 denotes the value Δ introduced in Sect. 2).

[1] We assume an environment, where an integer takes 32 bits and consists of four bytes.

[2] The largest value of a LCP of two suffixes of a text T equals to the length of the *longest repeat* in T, i.e. the longest substring of T that occurs at least twice. As the probability of a repeat of length m is $\frac{1}{|\Sigma|^m}$ and there are approximately $|T|^2$ places to start the repeat, we can write the expectation value $\mathbb{E}$("*number of repeats of length* m") $\approx \frac{|T|^2}{|\Sigma|^m}$. If the longest repeat is unique, its length m_l should satisfy $1 \approx \frac{|T|^2}{|\Sigma|^{m_l}}$, and therefore $m_l \approx 2 \log_{|\Sigma|} |T|$ (see the more detailed analysis in [7,12]). Thus, the values of LCPs are expected to be in the range $0 \ldots 127$.

- The length of the linked area is limited by a constant $\mathcal{C}$ by adding condition $j < \mathcal{C}$ in while-loop at line (7) of procedure *MarkLinksToSuffixArray* in Fig. 2. Also, the value of j must be copied to the value $A'(x).\Delta$ at line (17).
- The length of the link path is limited by a constant $\mathcal{D}$ by using array $pathdepth(1 \ldots n)$ in procedure *MarkLinksToSuffixArray*. The array entries are initialized to zero and condition $pathdepth(i + j) < \mathcal{D}$ is added in the line (7). The array $pathdepth$ is updated by adding $pathdepth(x + j) \leftarrow pathdepth(i + j) + 1$ in line (12) and $pathdepth(x) \leftarrow pathdepth(i) + 1$ in line (16).

Storing the value Δ similarly as Δx (using 8 bits) would imply that an entry of the CSA takes 6/5 times the size of an entry of the suffix array with LCP information. This is impractical, and therefore we store both values using only 4 bits (i.e. $\mathcal{C} = 15$, $\Delta \leq \mathcal{C}$, and $\Delta x \leq \mathcal{C}$). This does not affect the compression significantly, as can be seen from the experimental results in Sect. 6.

With the value Δ in the CSA, we can now replace the algorithm *UnCompactLink* by a more efficient version, that is shown in Fig. 6.

```
UnCompactLinkPractical(in CA, in x):
(1)  y ← CA(x).x; Δx ← CA(x).Δx; Δ ← CA(x).Δ; i ← 0;
(2)  while Δx > 0 do begin
(3)      if CA(y + i).B = false then begin
(4)          i ← i + 1; Δx ← Δx − 1 end
(5)      else begin
(6)          if CA(y + i).Δ ≤ Δx then begin
(7)              Δx = Δx − CA(y + i).Δ; i ← i + 1 end
(8)          else break end end;
(9)  while true do begin
(10)     if CA(y + i).B = true then begin {y + i contains a link}
(11)         UCA ← UnCompactLink(CA, y + i);
(12)         Add(Area, UCA(Δx + 1 ... min(|UCA|, Δx + Δ)),1);
(13)         Δx ← 0; end
(14)     else {y + i contains a suffix}
(15)         Add(Area, CA(y + i).x, 1);
(16)     if |Area| ≥ Δ then break;
(17)     i ← i + 1 end;
(18) return Area;
```

Fig. 6. A version of *UnCompactLink* with improved worst case bounds.

Theorem 9. *The algorithm in Fig. 6 uncompacts a link of the CSA using* $O(\mathcal{C}^2(\mathcal{D} - \log_2 \mathcal{C}))$ *time in the worst case, where* $\mathcal{C}$ *is a constant limiting the length of the linked area and* $\mathcal{D}$ *is a constant limiting the length of the link path.*

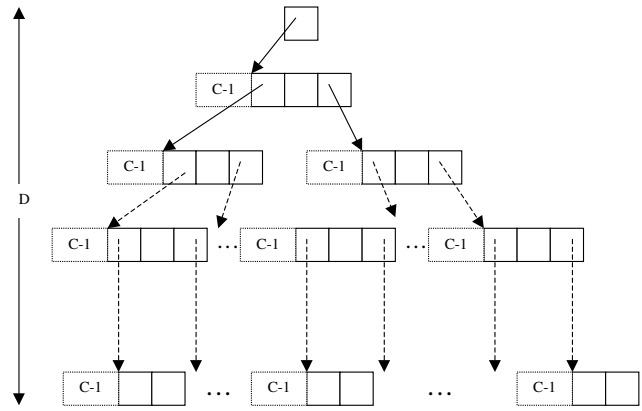


Fig. 7. The recursive search tree to uncompact a link in the worst case.

Proof. The recursive search tree of the worst case is visualized in Fig. 7.

The first call to *UnCompactLink* requires to skip at most $\mathcal{C}-1$ suffixes (when $\Delta x = \mathcal{C}-1$). After that, there is a link, a suffix, and a link to represent the linked area, so that the tree will grow as much as possible and code as little information as possible. Because a link can represent at most $\mathcal{C}$ suffixes, there is a limit to the growth of the tree. On the other hand, there could be link paths that do not code any information. As we have limited the length of the link path by a constant $\mathcal{D}$, the worst case is achieved when the links at the last level of the search tree each represent the minimum two suffixes that are located at the end of the link path of length $\mathcal{D}$.

The amount of widening levels (denoted by x) in the worst case search tree can be solved from the following equation:

$$\sum_{i=0}^{x} 2^i + 2 \cdot 2^{x+1} = \mathcal{C}, \tag{1}$$

where the summation denotes the suffixes at levels $1 \ldots x$ and the last term denotes the suffixes at the end of the link path. The equation (1) has a solution $x = \log_2 \frac{\mathcal{C}+1}{3} - 1$. We have to limit to the values of $\mathcal{C}$ that result a whole number solution to the value of x (for other values, the worst case search tree of Fig. 7 would be unbalanced, and thus slightly harder to analyze).

To calculate the time needed to uncompact a link, we have to multiply each node of the search tree with $\mathcal{C}-1$. Therefore the time requirement is:

$$(\mathcal{C}-1)(\sum_{i=0}^{x} 2^i + 2^{x+1}(\mathcal{D}-x)), \tag{2}$$

which can be written as follows by substituting the above value for x:

$$(\mathcal{C}-1)(\frac{\mathcal{C}+1}{3}-1+\frac{\mathcal{C}+1}{3}(\mathcal{D}-\log_2\frac{\mathcal{C}+1}{3}+1))). \quad (3)$$

□

As a corollary of Theorem 9, we have $O(|P|+\log n')$ worst case search time from the CSA (when LCP values are stored in integer fields). Although, the constant factor is quite big compared to the original suffix array. The experimental results in next section show that the constants $\mathcal{C}$ and $\mathcal{D}$ can be set relatively small without significant loss in compression.

6 Experimental Results

Two important properties of the CSA were tested: how much CSA saves space as compared to the suffix array, and how much the search times suffer from compression.

The space requirements of some files from the Calgary and Canterbury corpora [1] are shown in Table 1. The SA_LCP means the size of a suffix array with the LCP information, i.e. five times the size of the text. The CSA is constructed as described in Sect. 5 using values $\mathcal{C}=15$ and $\mathcal{D}=4$ (the values in parentheses in the last column of Table 1 show the reduction percentage when $\mathcal{C}=255$ and $\mathcal{D}$ is not used). As can be seen, the corresponding CSA takes about half the size of the suffix array with two texts: *world192.txt* and *progp* (Pascal-program code). This is quite good result as normal compression routine like *zip* compresses these texts by 71% and 77%. The text *book1* does not have much repetitiveness and is therefore compressed poorly even with *zip* (59%). The worst text to the CSA is *e.coli* containing DNA, though it compresses well with *zip* (71%). This is due to the nature of the repetitiveness; e.g. string "aggggt" may occur frequently in DNA, but so will occur strings "cggggt", "gggggt", and "tggggt". When a suffix array is constructed, suffixes starting with "aggggt" will be before suffixes starting with "cggggt", "gggggt", and "tggggt". Now, suffixes starting with "ggggt" cannot be represented as links to any of these suffixes starting with "aggggt", "cggggt", "gggggt", or "tggggt", because they will occur in different order. This is in contrast to the word "procedure" in *progp*; there will be no other words containing word "rocedure" and so all suffixes starting with "rocedure" can be replaced by a link to the area of suffixes starting with "procedure".

The search times are compared in Table 2. The total time to search 10000 patterns is reported for each text. For natural language texts, the patterns are chosen randomly from different sources. For *e.coli*, all patterns of length 6 are generated and they are combined with some patterns of length 7 to get the total amount of 10000. Each search is repeated 10, 100, or 1000 times depending on the size of the text. The first line on each text denotes the time needed to determine whether there is an occurrence of the pattern (the column *occurrences* reports the amount of patterns that were found). The second line on each text

Table 1. Space requirement

text	—text—	SA_LCP	CSA	reduction %
book1	768 771	3 843 855	2 502 785	35% (36%)
e.coli	4 638 690	23 193 450	18 722 175	19% (19%)
progp	49 379	246 895	129 560	48% (52%)
world192.txt	2 473 400	12 367 000	6 590 370	47% (52%)

Table 2. Search times

text	—text—	occurrences	SA_LCP (ms)	CSA (ms)
book1	768 771	4 466	27	39
book1	768 771	530 950	32	112
e.coli	4 638 690	10 000	34	39
e.coli	4 638 690	6 328 248	90	1054
progp	49 379	1 224	21	49
progp	49 379	34 151	21	81
world192.txt	2 473 400	5 632	34	47
world192.txt	2 473 400	1 283 892	44	278

denotes the time needed to report all the occurrences. Times are milliseconds on a Pentium III 700 MHz machine with 774 MB main memory running Linux operating system.

The search algorithms use LCP values, and thus work in $O(|P|+\log|T|)$ time in the average case. The LCP values were calculated using a simple algorithm that uses $O(|T|\log|T|)$ time in the average case (the worst case is not a problem, because we have limited the lengths of the LCPs to be smaller than 128).

It can be seen that with the basic string matching query, the CSA is not significantly slower than the suffix array. As predicted, the more occurrences there are, the more time takes searching from the CSA. The reason why the search time also increases in the suffix array, is that matches were reported (not printed, but calculated one by one) for the sake of comparison.

7 Conclusion

We have considered an efficient way to compact a suffix array to a smaller structure. A new search algorithm has been developed for the string-matching problem using this compact suffix array. It has been shown that a basic string-matching query can be answered in average time $O(|P| + \log|T|)$ on typical texts. The same bound can be achieved in the worst case, although with a larger constant factor and smaller compression.

The algorithm in Sect. 3 that constructs a CSA from a suffix array does not necessarily give the smallest CSA, because the method used for preventing cycles, can also prevent some non-cyclic links. It is an open problem to develop

an efficient algorithm to construct the smallest possible CSA (a Union-Find approach for preventing cycles could be possible). Another open problem is to construct a CSA straight from the text. This would save space at construction time, as our algorithm needs space for two suffix arrays and some additional structures during the construction.

Acknowledgement

I wish to thank Esko Ukkonen and Juha Kärkkäinen for their comments and advice.

References

1. R. Arnold and T. Bell, *A corpus for the evaluation of lossless compression algorithms*, Proceedings of the Data Compression Conference, 1997, pp. 201-210. `http://corpus.canterbury.ac.nz`
2. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas, *The smallest automaton recognizing the subwords of a text*, Theor. Comp. Sci., 40 (1985), pp. 31-55.
3. A. Blumer, J. Blumer, D. Haussler, and R. McConnell, *Complete inverted files for efficient text retrieval and analysis*, Journal of the ACM, 34:3 (1987), pp. 578-595.
4. A. Blumer, D. Haussler, A. Ehrenfeucht, *Average sizes of suffix trees and dawgs*, Discrete Applied Mathematics, 24 (1989), pp. 37-45.
5. M. Crochemore, *Transducers and repetitions*, Theor. Comp. Sci., 45 (1986), pp. 63-86.
6. M. Crochemore and Renaud Vérin, *Direct Construction of Compact Directed Acyclic Word Graphs*, In Proc. of the CPM, LNCS 1264 (1997), pp. 116-129.
7. Erdös and Rényi, *On a new law of large numbers*, J. Anal. Math. 22 (1970), pp. 103-111.
8. G. H. Gonnet, R. A. Baeza-Yates, and T. Snider, *Lexicographical indices for text: Inverted files vs. PAT trees*, Technical Report OED-91-01, Centre for the New OED, University of Waterloo, 1991.
9. U. Manber and G. Myers, *Suffix arrays: A new method for on-line string searches*, SIAM J. Comput., 22 (1993), pp. 935-948.
10. E. M. McCreight, *A space economical suffix tree construction algorithm*, Journal of the ACM, 23 (1976), pp. 262-272.
11. E. Ukkonen, *On-line construction of suffix-trees*, Algorithmica, 14 (1995), pp. 249-260.
12. M. S. Waterman, *Introduction to Computational Biology*, Chapman & Hall, University Press, Cambridge, UK, 1995, pp. 263-265.
13. P. Weiner, *Linear pattern matching algorithms*, In Proc. IEEE 14th Annual Symbosium on Switching and Automata Theory, 1973, pp. 1-11.

Linear Bidirectional On-Line Construction of Affix Trees

Moritz G. Maaß

TU München, Fakultät für Informatik
maass@informatik.tu-muenchen.de

Abstract. Affix trees are a generalization of suffix trees that is based on the inherent duality of suffix trees induced by the suffix links. An algorithm is presented that constructs affix trees on-line by expanding the underlying string in both directions and that has linear time complexity.

1 Introduction

Suffix trees are a widely studied and well known data structure. They can be applied to a variety of text problems and have become more popular with their application in problems related to molecular biology. There exist many linear algorithms for their construction [Ukk95], [McC76]. They all make use of auxiliary edges called suffix links. Usually, when taking edges in the suffix tree, the string represented by the current node is lengthened or shortened at the end. Suffix links are used to move from one node to another so that the represented string is shortened at the front. This is extremely useful since two suffixes of a string always differ by the characters at the beginning of the longer suffix. Thus successive traversal of all suffixes can be sped up greatly by the use of suffix links.

As proven by Giegerich and Kurtz in [GK97], there is a strong relationship between the suffix tree built from the reverse string (often called reverse prefix tree) and the suffix links of the suffix tree built from the original string.

Obviously, the suffix links form a tree by themselves (with edges pointing to the root, see Figures 1c, 1d, and 2). Giegerich and Kurtz have shown that the suffix links of the atomic suffix tree (AST - sometimes also referred to as suffix trie) for a string t are exactly the edges of the AST for the reverse string t^{-1} (i.e., the reversed suffix links labeled with the letters they represent and the same nodes are the AST for the reverse string). This property is partially lost when turning to the construction of the compact suffix tree (CST - most often referred to simply as suffix tree). The CST for a string t contains a subset of the nodes of the CST for the reverse string (which we will call compact prefix tree (CPT)) and the suffix links of the CST represent a subset of the edges of the CPT.

A similar relationship was already shown by Blumer et al. in [BBH$^+$87] for compact directed acyclic word graphs (c-DAWG). As shown there, the DAWG contains all nodes of the CPT (where nodes for nested prefixes are also inserted,

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 320–334, 2000.

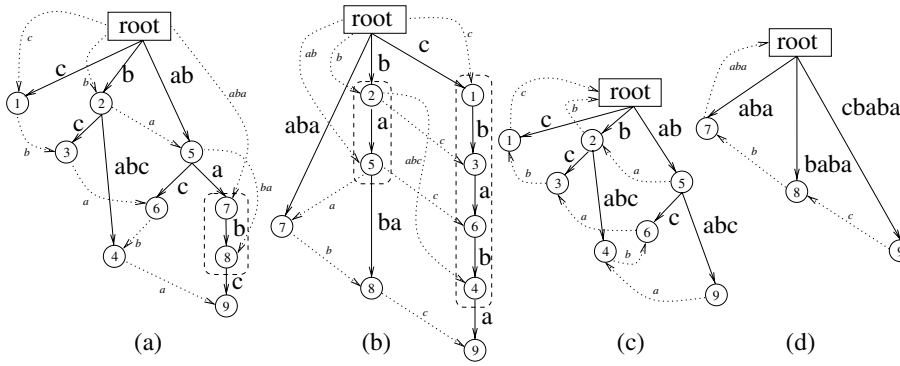


Fig. 1. Affix tree for `ababc` in suffix (a) and prefix (b) view (dotted lines represent the prefix edges, dashed boxes mark the paths) and suffix tree for `ababc` (c) and `cbaba` (d) (dotted lines represent the suffix links).

although they are not branching) and uses the CPT's edges as suffix links. The c-DAWG is the "edge compressed" DAWG. Because the nodes of the c-DAWG are invariant under reversal one only needs to append the suffix links as reverse edges to gain the symmetric version. Although not stated explicitly, Blumer et al. in essence already observed the dual structure of suffix trees as stated above because the c-DAWG is the same as a suffix tree where isomorphic subtrees are merged.

The goal of this paper is to show how a tree that incorporates both the CST and the CPT can be built on-line in linear time by appending letters in any order at either side of the underlying string. This tree will be called compact affix tree (CAT). It has been introduced by Stoye in his Master's thesis [Sto95] (an English translation is available [Sto00]). Our algorithm essentially has the same steps as Stoye's algorithm, but we will use additional data collected in paths (see section 2.2 and Figures 1a and 1b) to assure a suffix-tree-like view. Stoye could show that the CAT takes linear space, but he was unable to show linear time complexity of his algorithm. The basic algorithmic idea is to merge an algorithm for building CSTs on-line (Ukkonen's algorithm) with an algorithm for building CSTs anti-on-line (appending to the front of the string similar to Weiner's algorithm [Wei73]).

The core problem for achieving linear complexity is that Ukkonen's algorithm (or any other suffix tree algorithm) expects suffix links of inner nodes to be atomic (i.e., having length one). As can easily be seen in Figures 1a, 1b, and 2, this is the case for suffix trees, but not for affix trees (nodes 7, 8 in Figure 1a and 4, 5 in Figure 1b). Stoye tried to solve this problem by traversing the tree to find the appropriate atomic suffix links. In general, this approach might lead to quadratic behavior, which is the reason why Stoye could not prove linear time complexity of his algorithm. We will use additional data collected in paths to

establish a view of the CAT, as if it were the corresponding CST (this can be seen comparing Figures 1a, 1b with Figures 1c, 1d).

We will continue by briefly describing the data structures in section 2. Then the algorithms for constructing CSTs in linear time on-line by adding characters either always to the front or to the end of the string are given in section 3. These are merged to form the linear bidirectional on-line algorithm for CATs in section 4. The linearity for bidirectional construction is proven in section 5. We finish with a section on the implementation of paths.

2 Data Structures

In the following, let Σ be an arbitrary finite alphabet. Let Σ^* denote the set of all finite strings over Σ, let λ be the empty string and let $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$ denote the set of all non-empty strings over Σ. If $t \in \Sigma^*$ is a string, then let $|t|$ be its length and let t^{-1} be the reverse string. If $t = uvw$ for any $u, v, w \in \Sigma^*$, then u is a *prefix*, w a *suffix*, and v a *substring* of t. A suffix w of t is called *nested* if $t = uwv$ for $v \in \Sigma^+$ and $u \in \Sigma^*$ (nested prefix is defined analogously). A substring w of t is called *right branching* if $a, b \in \Sigma$ are two distinct characters ($a \neq b$) and $t = uwav$ and $t = u'wbv'$ for any $u, u', v, v' \in \Sigma^*$ (left branching is defined analogously).

2.1 Trees

We will define the affix tree data structure in terms of [GK97] beginning with Σ^+-trees.

Definition 1 (Σ^+-Tree). *A Σ^+-tree T is a rooted, directed tree with edge labels from Σ^+. For each $a \in \Sigma$, every node in T has at most one outgoing edge whose label starts with a. A Σ^+-tree is called* atomic*, if all edges are labeled with single letters. A Σ^+-tree is called* compact*, if all nodes other than the root are either leaves or branching nodes.*

We represent the edge labels by means of pointers into the underlying string, thus keeping the size for each label constant. Each path from the root to an arbitrary node n in T represents a unique string $w = \mathsf{path}(n)$. If $u = \mathsf{path}(n)$, we can identify the node by the string and we will write $\overline{u} = n$. A string $u \in \Sigma^*$ is in $words(T)$ iff there is a node n in T with $\mathsf{path}(n) = uv$ with $v \in \Sigma^*$. If a string u is represented in T by a node n (i.e., $u = \mathsf{path}(n)$), we call the location of the string explicit. Otherwise, if the string u is represented but not by a node (i.e., there exists a node n and a string $v \in \Sigma^+$ s.t. $uv = \mathsf{path}(n)$), we say the location of the string is implicit.

A *suffix link* is an auxiliary edge from node n to node m where m is the node s.t. $\mathsf{path}(m)$ is the longest proper suffix of $\mathsf{path}(n)$ represented by a node in T. Suffix links can be labeled similarly to edges, but with the string label reversed (see Figure 1). For example, a suffix link leading from a node representing the string `cbab` to a node representing the string `b` will be labeled `abc`.

Definition 2 (Reverse Tree T^{-1}). *The reverse tree T^{-1} of a Σ^+-tree T augmented with suffix links is defined as the tree that is formed by the suffix links of T, where the direction of each link is reversed, but the label is kept.*

Definition 3 (Suffix Tree). *A suffix tree of string t is a Σ^+-tree with $words(T) = \{u | u \text{ is a substring of } t\}$.*

As stated above, Giegerich and Kurtz have proven in [GK97] that the AST T for the reverse string t^{-1} is the same as the reverse AST T^{-1} for the string t.

Affix trees can be defined as 'dual' suffix trees.

Definition 4 (Affix Tree). *An affix tree T of a string t is a Σ^+-tree s.t. $words(T) = \{u | u \text{ is a substring of } t\}$ and $words(T^{-1}) = \{u | u \text{ is a substring of } t^{-1}\}$.*

Note that a string might have two different locations in CAT if it is not explicit. The following lemma presents an easily verifiable fact about CSTs.

Lemma 1. *In the CST for string t, each node represents either the empty string (the root), a right branching substring, or a non-nested suffix of t.*

The CAT contains all nodes that would be part of the CST and the CPT. With this lemma, each node can easily be checked to belong either to the suffix part, the prefix part, or both, thus defining a node type. For a node to be a suffix node it needs to be the root or have zero or more than one suffix child (because it is a leaf or because it is branching). The attribute needs not be saved explicitly but can be reconstructed on-line. Edges are either suffix or prefix edges. In CATs the suffix links are the (reversed) prefix edges and vice versa. All edges need to be traversable in both directions.

Compare Figures 1a and 1b with Figures 1c and 1d to see the differences between CAT and CST. In the following let CAT(t) denote the CAT for the string t (CST(t), CPT(t) analogously).

2.2 Additional Data for the Construction of CAT

Let $\alpha(t)$ be the longest nested suffix of string t and $\hat{\alpha}(t)$ the longest nested prefix of t. There are some important locations in CAT(t) that are needed for the building process. These locations are (the string arguments are dropped if they are clear from context):

- The active suffix point $asp(t)$, which represents the suffix location of the longest nested suffix (the active suffix) $\alpha(t)$.
- The active suffix leaf $asl(t)$, which represents the shortest non-nested suffix of t.
- The active prefix point $app(t)$, which represents the prefix location of the longest nested prefix (the active prefix) $\hat{\alpha}(t)$.
- The active prefix leaf $apl(t)$, which represents the shortest non-nested prefix of t.

The *asl* has a special role regarding suffix links.

Lemma 2. *In $CST(t)$ all suffix nodes except for the asl and the root have atomic suffix links.*

The proof should be obvious and follows from Lemma 1.

To keep track of *asp* and *app*, which might both be implicit, we will use *reference pairs* (rps) (similarly defined by Ukkonen [Ukk95]). A rp is a pair (n, u) where n is a node (called the base of the rp), $u \in \Sigma^*$ is a string, and $\mathsf{path}(n)u$ is the represented location. Since there are always two implicit locations of any string not represented by a node in CAT (e.g. bab in Figure 1a and 1b), the rps will have a type. A suffix rp will refer to a suffix location (which might be implicit in a suffix edge) and use a suffix node as base. With *canonical reference pair* (crp) we will refer to a rp (n, u) that is closest to the represented location while still being of the referenced type (i.e., if (n, u) is a suffix crp, then n is a suffix node and there exists no rp (n', u') for the same location with $|u'| < |u|$ and where n' is also a suffix node). For *asp* a suffix crp and for *app* a prefix crp is kept during all iterations.

By Lemma 2, all nodes that belong to the suffix part are connected by atomic suffix links. If they belong exclusively to the suffix part, they cannot be branching in the prefix part and can therefore be combined to clusters of "suffix only" nodes connected with atomic prefix edges in the prefix part. Each such cluster will be combined in a *path* (*asl* might have a non-atomic edge, but that will be dealt with in section 6). See the dashed boxes marking the paths in Figure 1a and 1b. The paths contain all nodes that would not be part of the corresponding CST (compare Figures 1a and 1b with 1c and 1d). The same holds true for nodes that only belong to the prefix part. The paths allow the following operations in constant time, thus behaving like edges:

- Access the first and the last node from either side.
- Access the i-th node.
- Split the path similar to edges at a given position (e.g. making node 6 of Figure 1b a prefix node).
- Append or prepend a node to a path.

With these paths, the CAT can be traversed ignoring nodes of the reverse kind: if a prefix only node is hit while traversing the suffix part, it must have a pointer to a path. The last node of that path is again followed by a suffix node. When taking a suffix view and using paths like edges the CAT behaves just like a CST. This is the key to solving the problem of Stoye's algorithm and to achieve linear time.

We will also use *open edges* as introduced by Ukkonen [Ukk95]. The edges leading to leaves will be labeled with the starting point and an infinite end point (i.e., (i, ∞)) that represents the end of the underlying string. Such an edge will always extend to the end of the string.

3 On-Line Linear Construction of Compact Suffix Trees

We will briefly describe how to construct a CST by expanding the underlying string to the right and how to construct a CST by expanding to the left. To solve these problems we can make use of two already existing algorithms, namely Ukkonen's algorithm [Ukk95] and Weiner's algorithm [Wei73]. Weiner's algorithm is rather complicated, so we will assume some additional information readily available in the later algorithm for constructing CATs.

3.1 Normal On-Line CST Construction

Ukkonen's algorithm constructs a CST on-line from left to right, building CST(ta) from CST(t). It essentially works by lengthening all suffixes represented in CST(t) by the character a and inserting the empty suffix. It should be obvious that the resulting tree is CST(ta). To keep the linear time bound two ideas help: Through the use of open edges the suffixes representing leaves are automatically lengthened. A reference to the longest represented nested suffix (asp) is always kept by means of a crp. It is then checked whether the lengthened suffix is still represented in the tree. If not, the suffix is not nested any more and a leaf is inserted. Such a suffix is called a *relevant suffix* (every non-nested suffix sa of string ta is a relevant suffix iff s is a nested suffix of t). After that the next shortest nested suffix is examined. As soon as one suffix is still nested after lengthening it by a all shorter suffixes will also stay nested and the update is complete. The location of that first nested suffix s of t, where sa is also nested in ta, is called the *endpoint*. sa is then $\alpha(ta)$, the new active suffix. To reach the next shortest suffix the base of the crp is replaced by the node its suffix link points to. After that the new rp is canonized by moving the base down the tree as far as possible thus shortening the string part. The number of nodes traversed in this way can be estimated by the length change of the string part leading to an overall linear complexity.

Theorem 1. *Ukkonen's algorithm constructs CST(t) on-line in time $\mathcal{O}(|t|)$.*

The proof and the algorithmic details can be found in [Ukk95]. An example of one iteration can be seen in Figure 2c and 2d. The procedure for one iteration as described above will be called update-suffix.

3.2 Anti-on-Line CST Construction with Additional Help

We will describe how to build CST(at) from CST(t). CST(t) already represents all suffixes of CST(at) but at. The leaf for at will branch at the location representing $\hat{\alpha}(at)$ in CST(t) (which is $app(at)$). There are three possibilities. The node location may be represented by an inner node, the location is implicit in some edge, or the location is represented by a leaf. In the last case the new leaf does not add a new branch to the tree, but extends an old leaf that now represents a nested suffix. The node representing the old leaf needs to be deleted after attaching the new leaf to it (see Lemma 1).

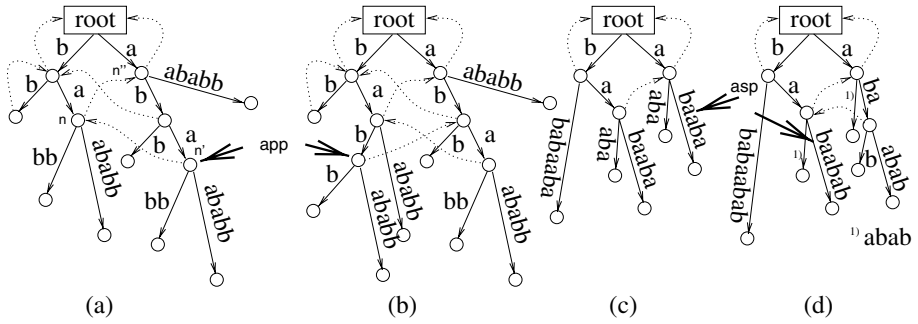


Fig. 2. CST(abaababb) and CST(babaababb) with *app* marked and CST(bbabaaba) and CST(bbabaabab) with *asp* marked.

See Figure 2a and 2b for an example of one iteration where a new branching node is inserted. To find the location of $app(at)$ let (n, u) be a crp for $app(at)$ and (n', u') a crp for $app(t)$ in CST(t). $app(at)$ needs to be found in CST(t). If n is not the root, it has a suffix link to some node n'' that is on the path from n' to the root. Hence n can be found starting from n' by moving up the tree until a node n'' is found that is target of an atomic suffix link that lengthens the represented string by a at the front (this means traversing the suffix link backwards). These three nodes are marked in Figure 2a. If we knew the final length of $\hat{\alpha}(at)$, we could easily determine the length of u. Because we will later be constructing CATs we allow the additional information of knowing that length. The length is also used to identify the node that might be target of an additional suffix link for a possibly inserted branching inner node.

Clearly, $|\hat{\alpha}(t)| + 1 \geq |\hat{\alpha}(at)|$ and there exists a string $v \in \Sigma^*$ s.t. $\hat{\alpha}(t) = vv'$ and $\hat{\alpha}(at) = av$ with $v' \in \Sigma^*$. The number of traversed nodes is limited by the length of the active prefix, so the following theorem is established easily.

Theorem 2. *With the additional information of knowing the length of $\hat{\alpha}(s)$ for any suffix s of t before inserting it, it takes $\mathcal{O}(|t|)$ time to construct CST(t) in an anti-on-line manner.*

Proof. Inserting the leaf, inserting an inner branching node, or deleting an old leaf takes constant time per step. To find the base n_i of the crp for the new *app* from the old base n_{i-1}, a number of nodes need to be traversed. For each node encountered the string represented by the new base will be at least one character shorter. Taking the reverse suffix link lengthens the path by one. If s_i is the number of nodes encountered in iteration i, the following inequality therefore holds:

$$|\mathsf{path}(n_i)| \leq |\mathsf{path}(n_{i-1})| - s_i + 1 \Leftrightarrow s_i \leq |\mathsf{path}(n_{i-1})| - |\mathsf{path}(n_i)| + 1 \quad (1)$$

Thus the total number of nodes traversed is $\sum_{i=1}^{|t|} s_i \leq \sum_{i=1}^{|t|} |\mathsf{path}(n_{i-1})| - |\mathsf{path}(n_i)| + 1 \leq |t| + |\mathsf{path}(n_0)| - |\mathsf{path}(n_{|t|})| \leq |t| = \mathcal{O}(|t|)$ □

A more detailed discussion of Weiner's algorithm using the terms of Ukkonen's algorithm can be found in [GK97]. The procedure for one iteration as described above will be called update-prefix.

4 Constructing Compact Affix Trees On-Line

Obviously, the algorithm is dual. The necessary steps to construct CAT(ta) from CAT(t) are the same as for constructing CAT(at) from CAT(t). For clarity we will assume the view of constructing CAT(ta) from CAT(t). We call this a suffix iteration, while CAT(at) from CAT(t) is called a prefix iteration. The part representing CST(t) will be referred to as suffix part and the part representing CST(t^{-1}) as prefix part.

We will merge the above given algorithms (section 3) to get the final algorithm for the construction of CATs. Remember that both algorithms rely heavily on the fact that all suffix links are atomic. With the use of paths we can establish a CST-like view, which guarantees atomic suffix links by Lemma 2. When inserting new nodes that belong only to one part it is important to add them to paths (either new or existing ones). Given the above path properties (see section 2.2), this will pose no problem. Another modification common to both parts is the use of paths and of suffix and prefix crps so that the CAT behaves like a CST (respectively CPT) for each algorithm part.

To update the suffix part we apply a modified update-suffix. Except for the general modifications the leaves need to get suffix links, too. This is where the *asl* is needed. For each new leaf l, *asl* will get a suffix link to l, and l will become the new *asl*. *asl* initially has a suffix link to some node s. The link will be removed before lengthening the text since *asl* will "grow" because of the open edge leading to it and its suffix link cannot grow, leading to an inconsistency. Nodes s and *asl* will be used for the next part.

To update the prefix part we apply a modified update-prefix, where – in order not to switch the view – we talk of prefixes and not of suffixes (replace suffix by prefix and vice versa in section 3.2). Recall that the prefix edges are the reversed suffix links. Note that the suffix leaves form a chain connected by suffix links that represent a prefix of t and hence also of ta (see Figure 1a and 1b). After update-suffix they still form a chain representing a (possibly longer) prefix of ta, thus representing part of the prefix that needs to be inserted in the prefix part. By removing the suffix link from *asl* to s, the prefix location for t was lost. The *asl* has not been relinked yet. Hence the prefix location for t needs to be reinserted and *asl* needs to be suffix linked correctly to form the prefix location for ta. The first step is obvious because we have node s and we know that the prefix leaf for t is suffix parent to the prefix leaf for ta (see Figure 1; the node representing the whole string is easily kept globally, since it never changes due to the open edges). The second step involves the same search as in update-prefix to find the prefix location of *asp*(ta). The additional information is easy to give because we already know the length of $\alpha(ta)$ from update-suffix. Furthermore, it may happen that we take a starting point that is above the base n' for the crp

for $\alpha(t)$ in the prefix part because we have inserted relevant suffixes. In Figure 2d the endpoint of update-suffix is $\overline{ba}$, which is $\overline{ab}$ in Figure 2a and lies between n' and n''. In case the location of $\alpha(ta)$ is already an explicit prefix node we can even omit the search, since it will be the $|u|$-th element of the path starting after the base n (or n if $|u| = 0$), if (n, u) is the crp for *asp*(ta) (see Figure 3d, where the root is the base for the crp to *asp*, and nodes 2, 4, and 6 form a path starting at the root). Section 4.1 describes how the new starting point is found.

Because the algorithm is bidirectional we need to update *app* and *apl*. *apl* only changes if a new "shortest" leaf is inserted (i.e., if the previous *apl* was t, which is only the case if $t = a^k$ for $a \in \Sigma$), or if *app* grows. *app* might change in accordance with a lemma by Stoye [Sto95] (Lemma 4.8):

Lemma 3. *The active prefix will grow in the iteration from t to ta iff the new active suffix $\alpha(ta)$ is represented by a prefix leaf in CAT(t).*

That is the case when a node needs to be deleted.

These are essentially the same steps that Stoye's algorithm performs. (The reason behind each step is more elaborately described in [Sto95], [Sto00].) The important difference is that our algorithm makes use of paths (as described in section 2.2 and section 6) in order to skip over prefix nodes when updating the suffix part and vice versa. In summary, the six necessary steps of each iteration of the algorithm are:

1. Remove the suffix link from *asl* to s.
2. Lengthen the text, thereby lengthening all open edges.
3. Insert the prefix node for t as suffix parent of $\overline{ta}$ and link it to s.
4. Insert relevant suffixes and update suffix links (update-suffix).
5. Make the location of the new active suffix $\alpha(ta)$ explicit and add a suffix link from the new *asl* to it (main part of update-prefix).
6. Update the active prefix, possibly deleting a node.

Step 4 is nearly the same as update-suffix, so we will clarify the other steps by means of a small example. Figure 3 shows the CAT for baa (a), the tree after step 2 (b) (node 3 is the *asl*, node 5 is s, which is only accidentally the same as *asp*), the tree after step 3 (c) (node 6 was inserted to represent the prefix baa and it was linked to s), the tree after step 5 (d) (node 7 was inserted for the relevant suffix ab, the root was the endpoint, *asp* was already an explicit prefix node, it is node 2, the new *asl* is node 7, it was prefix linked to node 2), and finally the new CAT for baab after step 6 (e) (node 2 had to be deleted because it is neither part of CST nor of CPT for baab, which can easily be seen because it has exactly one incoming and one outgoing edge of either type and it was a leaf before). b is now the active prefix and the active suffix.

4.1 Finding the New Starting Point for the Search for the Prefix Location of $\alpha(ta)$

If a starting point is needed it will always be a prefix node that represents the string u (where $ua = \alpha(ta)$), which is the endpoint of that iteration. The following lemma will be helpful:

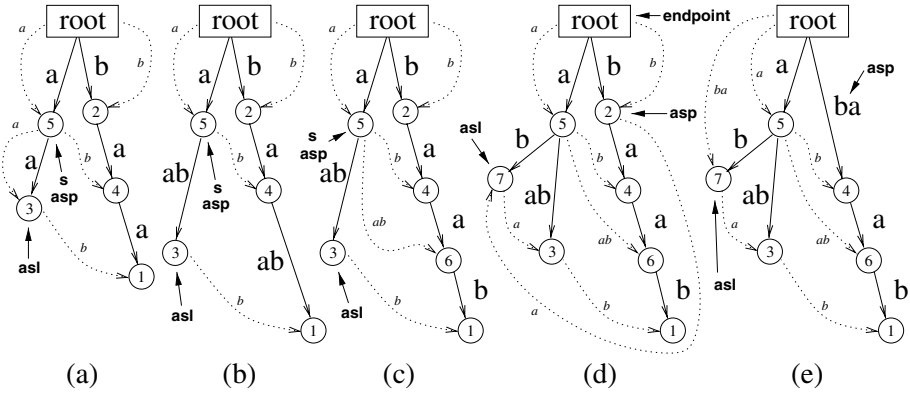


Fig. 3. Step-by-step construction of affix tree for `baab` from affix tree for `baa` in suffix view.

Lemma 4. *Assume that in the iteration from $CAT(t)$ to $CAT(ta)$ at least one relevant suffix is inserted before the endpoint g is reached. Let $\gamma = \mathsf{path}(g)$.*

1. *The location of the endpoint is always represented by a suffix node.*
2. *If the endpoint is not a prefix node, it is the active prefix of $CAT(t)$ and γa will be the new active prefix of $CAT(ta)$.*

Proof. Let $\alpha(t) = ud\gamma$ for some $u \in \Sigma^*$, $d \in \Sigma$. If relevant suffixes are inserted, then $t = vud\gamma$ (1), $t = v'ud\gamma bw$ (2) and $t = v''c\gamma aw'$ (3a) or $t = \gamma aw''$ (3b) where $v, w \in \Sigma^+$, $v', v'', w', w'' \in \Sigma^*$, $a, b, c, d \in \Sigma$ and $a \neq b, c \neq d$. Hence by (2) and (3a) / (3b) γ is right branching (proving 1).

If γa appears somewhere in t, where it is not a prefix of t, by (3a), γ must be left branching and $\overline{\gamma}$ is a prefix node. Otherwise γa is a prefix of t and no further γa in t, so γ is the active prefix of t. γa is also a suffix of ta, so γa will be the new active prefix of ta (proving 2). □

To find the prefix location of $\alpha(ta)$, we will distinguish the following cases:

1. No relevant suffixes were inserted, the active prefix got longer in the previous iteration.
2. No relevant suffixes were inserted, the active prefix did not change in the previous iteration.
3. Relevant suffixes were inserted, the endpoint is a prefix node.
4. Relevant suffixes were inserted, the endpoint is not a prefix node.

In case 1 the prefix location is already explicit in the prefix tree, it is a prefix leaf. In the previous iteration the active prefix got longer and was equal to the active suffix. By Lemma 3, a node that was previously a leaf got deleted (the active prefix is the longest nested prefix, so the next longer prefix must be a leaf). Since all leaves form a chain with inner distances one, the prefix location

of the *asp* must be the next leaf (i.e., the *apl*). It can be easily found by the path starting at the base of the suffix crp for *asp*.

In case 2 the active suffix was made an explicit prefix location in the previous iteration. This location will be the starting point for the search in the prefix tree.

In case 3 the endpoint is a prefix node as well as a suffix node. It will be the starting point for the search in the prefix tree.

Case 4 can be dealt with similar to case 1, since the endpoint is the active prefix by Lemma 4. Therefore, the prefix location is already explicit in the prefix tree as a prefix leaf (it is the *apl*).

Note that it is important to have a prefix node as starting point in the tree because this enables us to use the paths in the prefix tree (otherwise it might be necessary to search a prefix node first which would add extra complexity).

4.2 Complexity of the Unidirectional Construction

Since all operations of a single iteration are either constant (steps 1, 2, 3, 6) or have a linear amortized complexity by Theorem 1 (step 4) and Theorem 2 (step 5), the following theorem follows.

Theorem 3. *CAT*(t) *can be constructed in an on-line manner from left to right or from right to left in time* $\mathcal{O}(|t|)$.

5 Complexity of Bidirectional Construction

The affix tree construction algorithm is dual, so the tree can be constructed by any sequence of appending characters to any side of the string. We will prove the following:

Theorem 4. *Bidirectional construction of affix trees has linear time complexity.*

Because part of the complexity analysis is amortized, it is necessary to show that these parts do not interfere. The amortized complexity analysis depends upon step 4 and step 5. Their analyses are both based on the rps for the active suffix α_i and for the active prefix $\hat{\alpha}_i$ in iteration i. The amortized complexity analysis counts the nodes encountered using string lengths as an upper bound.

For the following lemmas and their proofs let $t^{(i)}$ be the string that is represented in the CAT after iteration i. Let (μ_i, ν_i) be the suffix crp for the suffix location of α_i, (u_i, v_i) be a (pseudo) prefix crp for the prefix location of α_i (pseudo because the prefix location of α_i might exist if it is a prefix node or when it will be inserted). Analogously, let $(\hat{u}_i, \hat{v}_i)$ $[(\hat{\mu}_i, \hat{\nu}_i)]$ be the suffix [prefix] crp for $\hat{\alpha}_i$, and let $\hat{\gamma}_i$ be the prefix endpoint in iteration i (which might be $\hat{\alpha}_{i-1}$). For $\hat{\gamma}_i$ let $(\hat{u}'_i, \hat{v}'_i)$ $[(\hat{\mu}'_i, \hat{\nu}'_i)]$ be the suffix [prefix] crp.

For step 4, additional work might occur when the active suffix grows in a prefix iteration s.t. the string distance from its location to the base of its crp grows (i.e., $|\nu_l| > |\nu_{l-1}|$). (Consider inserting $\mathtt{ba}^{\mathrm{k}+1}\mathtt{ba}^{\mathrm{k}+1}\mathtt{ac}$ from left to right versus inserting $\mathtt{a}^{\mathrm{k}+1}\mathtt{ba}^{\mathrm{k}+1}$ from left to right, appending b to the left and inserting ac on the right).

Lemma 5. *Additional k interim reverse (prefix) iterations add at most $\mathcal{O}(k)$ work to be accounted for in the amortized analysis of step 4.*

Proof. For step 4 we account the distance that is traversed by canonizing the rp for *asp*. For a given rp for *asp* we can determine the potential path (called *traversal path*) that would be taken in step 4 (e.g., if a previously unseen letter is appended). The nodes on this path make up the accounted complexity. The number of nodes is always less or equals to the string distance of the path (which we use in Theorem 1). We call a part of the path where the work is accounted by the traversed string distance *string-accounted*. If the work is accounted by the number of nodes, we call the part *node-accounted*. (Lemmas 6 and 7 introduce node-accounted parts in the analysis.) Furthermore, it is possible that the traversal path contains parts of the tree multiple times (the same nodes are traversed more than once, e.g. in the iteration from CAT(aab(ab)$^{\mathtt{i}}$) to CAT(aab(ab)$^{\mathtt{i}}$c)). We will call such parts of the path *nested*. We take a look at two cases. The case where the base of the suffix crp for the *asp* changes and the case where suffix nodes are inserted in reverse iterations.

For the first case, suppose the active suffix grows in iteration l, when appending from the left: $t^{(l)} = at^{(l-1)}$, $\hat{\alpha}_l = a\hat{\gamma}_l$, $\alpha_l = a\alpha_{l-1}$ and $\hat{\alpha}_l = \alpha_l \Rightarrow \hat{\gamma}_l = \alpha_{l-1}$. It follows that the crps must be equal: $(\mu_l, \nu_l) = (\hat{u}_l, \hat{v}_l)$ and $(\mu_{l-1}, \nu_{l-1}) = (\hat{u}'_l, \hat{v}'_l)$. Therefore, the number of nodes that are accounted for step 5 is exactly the number of nodes added to the traversal path of step 4. Since the active suffix grows by Lemma 3, the new suffix location is a leaf and can be found in constant time (see section 4.1, cases 1 and 4). Hence, the additional work imposed by this intermediate reverse iteration is already accounted for. A node-accounted part is added to the front of the traversal path.

For case two, we will distinguish the different parts of the tree and of the traversal path. Obviously, inserting a node into a string-accounted part does not matter. A node inserted into a node-accounted part of the path that is not nested will add $\mathcal{O}(1)$ work. Only if a nested, node-accounted part of the traversal path existed, would a node add more than $\mathcal{O}(1)$ work. Node-accounted parts can be appended to the front of the traversal path by case 1. For a part of the tree to be added a second time, the base of the rp for *asp* needs to be moved back down below that part without the part being traversed in step 4. This can only happen when the base is moved down in the tree over reverse suffix links in reverse iterations. It takes at least k iterations to move the base down by string distance k. For each iteration we can save a coin and use the coins to turn a previously node-accounted part of the traversal path string-accounted. Thus, before adding a part for the second time to the traversal path the first instance in the traversal path can be turned string-accounted. As a result, there cannot be node-accounted, nested parts of the traversal path.

Hence, the overall cost for k reverse (prefix) iterations is $\mathcal{O}(k)$. □

The following lemma will give us a bound on the remaining work yet to be done in step 5 by means of the current data structure.

Lemma 6. *For every sequence of suffix iterations from $CAT(t^{(i)})$ to $CAT(t^{(j)})$, $i < j$, let $\dot{s}_i$ be the number of prefix nodes on the prefix path from u_i to the root in iteration i and let $\dot{s}_j$ be the number of prefix nodes on the prefix path from u_j to the root in iteration j. Let s_k be the number of prefix nodes traversed in step 5 of iteration k, then $\dot{s}_i + (j - i) \geq \dot{s}_j + \sum_{k=i+1}^{j} s_k$.*

Proof. By Lemma 2, every prefix node w on the path from u_j to the root has an atomic prefix link starting a chain of atomic prefix links leading to the root. All nodes on the path are prefix nodes. The chain has length $|\mathsf{path}(w)|$. Every suffix iteration moves u_j over a reverse prefix link to a new prefix branch (step 5). Obviously, u_i and u_j have a distance of $(j-i)$ prefix links. Thus, for at most $(j - i)$ nodes $|\mathsf{path}(w)| \leq (j - i)$ and the chain ends at the root before reaching the path from u_i to the root. Every other node with $|\mathsf{path}(w)| > (j - i)$ can be coupled with a node on the path from u_i to the root by following its prefix link chain. Hence $\dot{s}_i + (j - i) \geq \dot{s}_j$.

Let $j = i + 1$ and s_j nodes be traversed from u_i in step 5 until the atomic prefix link to u_j is found. Then $\dot{s}_i + 1 \geq \dot{s}_j + s_j = \dot{s}_{i+1} + s_{i+1}$. Continuing the argument, $\dot{s}_i + 2 \geq \dot{s}_{i+1} + 1 + s_{i+1} \geq \dot{s}_{i+2} + s_{i+2} + s_{i+1}$. Induction easily yields $\dot{s}_i + (j - i) \geq \dot{s}_j + \sum_{k=i+1}^{j} s_k$. □

With this lemma, the number of prefix nodes on the prefix path from u_i to the root in iteration i can be used to limit the work accounted for in all subsequent steps 5 and limit the complexity of the unidirectional construction in terms of nodes: $\dot{s}_0 + n \geq \dot{s}_n + \sum_{k=1}^{n} s_k \implies \sum_{k=1}^{n} s_k \leq n - \dot{s}_n$.

Lemma 7. *The insertion of n_k prefix nodes in k interim reverse (prefix) iterations adds at most $\mathcal{O}(n_k + k)$ work for any following steps 5 of a suffix iteration.*

Proof. Suppose after k interim reverse iterations $|\mathsf{path}(u_{i+k})| > |\mathsf{path}(u_i)|$. Since the active suffix can only be enlarged along prefix edges k times, only k existing prefix nodes may be between $\overline{\alpha_{i+k}}$ (or its location) and u_i. All other prefix nodes must be nodes inserted in the prefix iterations. Hence a maximum of $n_k + k - 1$ prefix nodes can lie between u_{i+k} and u_i. Let $\dot{s}_i$ be the number of prefix nodes above u_i in iteration i and $\dot{s}_{i+k}$ be the number of prefix nodes above u_{i+k} after the k interim iterations. It follows that $\dot{s}_{i+k} - \dot{s}_i \leq n_k + k - 1$.

With Lemma 6, after iteration i the amortized work for i suffix iterations and additional m subsequent suffix iterations can be limited by $\sum_{j=1}^{i} s_j + \dot{s}_i + m$, where $\sum_{j=1}^{i} s_j$ is already performed and $\dot{s}_i + m$ is remaining. With k interim reverse iterations, the accounted work is then $\sum_{j=1}^{i} s_j + \dot{s}_{i+k} + m \leq \sum_{j=1}^{i} s_j + \dot{s}_i + m + n_k + k - 1$. The remaining work increased by $n_k + k - 1$.

□

Proof (Proof of Theorem 4).

Let the CAT T be constructed by l suffix iterations interleaved with k prefix iterations. Let $n = l + k$.

By Lemma 5, k additional reverse iterations add $\mathcal{O}(k)$ additional work for step 4. By Lemma 7, each preceding sequence of i prefix (suffix) iterations adds

at most $\mathcal{O}(n_i+i)$ additional work for all following suffix (prefix) iterations, where n_i is the number of prefix (suffix) nodes inserted in the sequence.

The total cost is therefore $\mathcal{O}(n)+\mathcal{O}(k+l)+\mathcal{O}(n_k+k+n_l+l) = \mathcal{O}(n+k+l+n_k+k+n_l+l) = \mathcal{O}(n)$ □

6 Implementation of Paths

We will show that paths with the properties used above can be maintained on-line. Each path is represented by two dynamic arrays which can grow dynamically with constant amortized cost per operation (see section 18.4 "Dynamic tables" in [CLR90]). With one array growing downward and the other one upward, a path can grow in either direction (the underlying string can be represented similarly).

Only the first and the last node of a path need to have access to the path. They hold a pointer to the double-sided dynamic array object, an offset, and a length. Thus the first, the last, and the i-th element can be retrieved in constant time. Appending or prepending elements can also be done easily in constant time. When the path is split somewhere in the middle, both new paths will use the same array with different lengths and offsets, thus this operation can be done in constant time, too. Since all nodes in a path are connected by atomic edges, the paths can never grow on a side where they were split, hence no problems will arise through the split operation. Only the *asl* might have a non-atomic suffix link or might be deleted, thus disturbing this property. Fortunately, this is not the case due to the following lemma:

Lemma 8. *The prefix parent of the active suffix leaf (asl) is (also) a prefix node.*

Proof (Idea). The proof is a purely technical case distinction and we will therefore only give the basic proof idea. The proof will be done by contradiction. We will assume *asl*'s parent $\overline{\omega}$ to be a suffix only node and make a case distinction over the letters preceding ω in all occurrences of ω in t (which are at least three, one because ω is a suffix of $\alpha(t)$, two other because ω is right branching). In the case where ω is a prefix of t, the characters preceding the other two occurrences must be examined iteratively until the left of the string is reached, leading to the contradiction in the final case. □

The total number of nodes contained in all paths is $\mathcal{O}(n)$. If a path is split in the middle a hole is in that path representing a prefix and suffix node. The total number of holes is also bounded by $\mathcal{O}(n)$, so the space requirement is linear.

7 Conclusion

Suffix trees are widely known and intensively studied. A less known fact is the duality of suffix trees with regard to the suffix links that is described in [GK97] and is implicitly stated in [BBH+87] and other papers. Stoye developed a first algorithm that could construct affix trees in an on-line manner, but could not

prove that his algorithm is linear [Sto95]. Through the introduction of paths, the algorithm could be improved and a linear behavior could be proven even for bidirectional construction.

Affix trees (especially augmented by paths) have the same properties as suffix trees and more. Construction is bidirectional and on-line. In affix trees, not only the suffixes, but also the prefixes of a string are given, and the tree view can be switched from suffixes to prefixes at any time and node, which cannot be achieved with two single suffix trees. At any position in the affix tree one can tell immediately whether the represented string is left branching or right branching or both.

Applications might be in pattern matching problems, where both the original and the reverse string are represented in one tree (e.g. finding all maximal repeats [Gus97]). Other new applications might come from the use of affix trees as two-head automaton with "edge compression" (see [GK97]).

Acknowledgements

I am in debt to Thomas Erlebach for thorough discussion and valuable criticism of this paper. I also want to thank Ernst W. Mayr for his help with the structuring of the final paper.

References

BBH+87. A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, July 1987.

CLR90. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1st edition, 1990.

GK97. R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. *Algorithmica*, 19:331–353, 1997.

Gus97. Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

McC76. Edward M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *J. ACM*, 23(2):262–272, April 1976.

Sto95. J. Stoye. Affixbäume. Master's thesis, Universität Bielefeld, May 1995.

Sto00. J. Stoye. Affix Trees. Technical Report 2000-04, Universität Bielefeld, Technische Fakultät, 2000,
`ftp://ftp.uni-bielefeld.de/pub/papers/techfak/pi/Report00-04.ps.gz`.

Ukk95. E. Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica*, 14:249–260, 1995.

Wei73. P. Weiner. Linear pattern matching. In *IEEE 14th Annual Symposium on Switching and Automata Theory*, pages 1–11. IEEE, 1973.

Using Suffix Trees for Gapped Motif Discovery*

Emily Rocke

CSE Department, University of Washington
Box 352350, Seattle, WA 98195-2350
ecrocke@cs.washington.edu

Abstract. Gibbs sampling is a local search method that can be used to find novel motifs in a text string. In previous work [8], we have proposed a modified Gibbs sampler that can discover novel gapped motifs of varying lengths and occurrence rates in DNA or protein sequences. The Gibbs sampling method requires repeated searching of the text for the best match to a constantly evolving collection of aligned strings, and each search pass previously required $\theta(nl)$ time, where l is the length of the motif and n the length of the original sequence. This paper presents a novel method for using suffix trees to greatly improve the performance of the Gibbs sampling approach.

1 Introduction

1.1 Motivation

The term *motif*, often used in biology to describe similar functional components that several proteins have in common, can also be used to describe any collection of similar subsequences of a longer sequence G, where the metric used to measure this similarity varies widely. In biological terms, these sets of similar substrings could be regulatory nucleotide sequences such as promoter regions, functionally or evolutionarily related protein components, noncoding mobile repeat elements, or any of a variety of kinds of significant pattern. More generally, if we find a collection of strings that are considerably more alike than a chance distribution could explain, then there is likely to be some underlying biological explanation, and if that explanation is not known for the motif in question, then investigating this motif is likely to be a fruitful line of questioning for biologists.

Although motif-finding tools exist that find *ungapped* motifs, the reality is that instances of biologically significant patterns can appear with insertions or deletions relative to other instances of the same underlying pattern. A tool to quickly and thoroughly discover approximate gapped motifs in long sequences would give a great deal of information to biologists on where to start looking for functional or critical regions. As the gap grows between the amount of available sequence data for various organisms and the relatively small percent of these

* This material is based on work supported in part by the National Science Foundation and DARPA under grant DBI-9601046, and by the National Science Foundation under grant DBI-9974498.

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 335–349, 2002.

sequences which are characterized, computational tools like this will become more and more useful in analyzing data as a supplement to work in the lab.

1.2 Overview

Gibbs sampling is a local search method that can be used to find novel motifs in a text string, introduced to computational biology by Lawrence *et al.* [5]. In previous work [8], we have proposed a modified Gibbs sampler that can discover novel gapped motifs of varying lengths and occurrence rates in DNA or protein sequences. This algorithm is powerful in practice, but also time-consuming, since each iteration requires searching the entire string for a best-match substring and its ideal alignment with the existing motif. This paper presents a method for using suffix trees to greatly improve the time performance of this approach.

Section 2 will give a brief introduction to Gibbs sampling and to suffix trees, and describe some previous work on finding motifs using suffix trees. Section 3 will introduce the basic outline of our algorithm, and sections 4 through 6 will describe the suffix tree search algorithm. Finally, section 7 will describe an implementation of the algorithm and report on its behavior in practice on genomic data.

2 Background and Previous Work

2.1 Gibbs Sampling

The original source for the algorithm presented here is Lawrence *et al.*'s Gibbs sampling algorithm for motif discovery[5]. The idea is to start with a random collection of length l substrings from the length n sequence G. For each iteration of the algorithm, discard one substring and search G for a replacement string. Any string in G is chosen with a probability proportional to its score according to some metric; in [5] the metric used is the relative entropy of the collection if that string were added:

$$\sum_{i=1}^{l}\sum_{j=1}^{A} q_{i,j}\log(\frac{q_{i,j}}{p_j}), \tag{1}$$

where A is the size of the alphabet, $q_{i,j}$ is the fraction of the i^{th} column of the collection consisting of letter α_j, and p_j is the fraction of G consisting of letter α_j. The basic idea of this scoring function is to reward distributions that are unlikely to occur by chance. The lowest score, 0, is found when an alignment column contains letters at exactly the proportion they occur in the sequence as a whole[1] (in other words, $q_{i,j}/p_j = 1$), while the highest score is obtained by a column consisting entirely of the letter $\alpha_{\min}$ which is *least* frequent in G, in which case the score for that column will be $\log(1/p_{\min})$. Each alignment column is scored independently, based only on the letters in that column, and the scores for each column are added together.

[1] The proof that 0 is the lowest possible score involves noting that $\ln(1/x) \geq 1 - x\ \forall x$, whence the proof can be derived algebraically.

Gapped Motifs. In previous work [8], we extended this algorithm to allow gapped motifs, requiring three main changes. First, we altered the scoring function to account for gaps, while retaining the basic relative entropy schema. Second, in order to allow gapped alignment, at each iteration we perform a dynamic programming search of G to simultaneously find the best string and its ideal alignment. To avoid an NP-complete multiple alignment problem, we make a simplifying assumption that at each iteration of the algorithm, the remaining strings are likely to already be in a good mutual alignment, since each one was aligned to the collection when it was originally added. Their alignment to one another can therefore be kept fixed, and only a 2-way alignment between the collection and the candidate string need be performed. Thus, our dynamic programming pass is in essence a standard Smith-Waterman 2-sequence alignment[10], albeit with a more complex scoring function. The third modification is that we take the highest scoring replacement string for each iteration, rather than any string with probability proportional to its score, due to the complexity of assigning scores not only to each string but to different possible alignments of the same string.

An additional feature is that our gapped Gibbs sampler decides as it progresses whether to lengthen the strings in the collection (if the context in G around the strings has high score), or to shorten them (if the edges of the strings have low score), and whether to add more strings to the collection or discard some. The result is a program that will flexibly find gapped motifs of varying lengths and number of occurrences—the user need not know in advance what kind of pattern is present—which makes it of considerable use in discovering novel features of DNA and amino acid sequences, as well as being applicable to areas outside of computational biology such as text processing, database searching, and other applications in which a long, error-prone data sequence must be analyzed.

However, a disadvantage is the speed at which this algorithm performs. Each pass through the dynamic programming step takes time $\theta(nl)$, where n is the length of G. In the end, a run of this sampling algorithm on a bacterial genome might take hours or days if it finds a long pattern in the sequence, and would be proportionally longer on more complex genomes. We would like to reduce the amount of unnecessary work performed in examining all of G each iteration, by doing a preprocessing step to store some of this information in advance. A perfect example of a data structure to compactly store this kind of information on a text is the suffix tree.

2.2 Suffix Trees

A suffix tree is a data structure which provides a powerful interface to quickly visit any given subsequence of a sequence $G = g_1 g_2 ... g_n$, and which can be constructed in $O(n)$ time and space. A thorough description of suffix tree construction and properties is beyond the scope of this paper, and can be found in [3].

However, to understand this paper it suffices to know three basic properties of suffix trees. First, a suffix tree is an A-ary tree, where A is the size of the alphabet, having a text "label" on each edge. Second, the (up to A) child edges from each node must each have an edge label beginning with a distinct letter in $\alpha_1, \ldots, \alpha_A$. Finally, let us say a path from the root of the tree to a leaf is labeled with some string S if the concatenation of the labels on each edge of the path, from root to leaf, is S. Then the key property of a suffix tree is that each "suffix" of G, that is, each subsequence $g_i g_{i+1} \ldots g_n$, appears in the suffix tree as a label from the root to some leaf, and conversely, every path from the root to a leaf is labeled with some suffix of G.

To understand this last property, notice how in Figure 1, a depiction of the suffix tree for the string `ACGACT`, each suffix (`ACGACT, CGACT, GACT, ACT, CT, T`) appears in the tree as a label on some path from root to leaf.

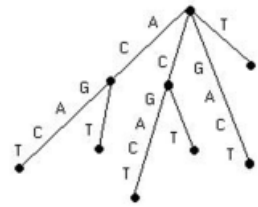


Fig. 1. An example suffix tree: the string "ACGACT"

An example use for this data structure is to query for a substring S in time $O(|S|)$. Note that any such S in G begins some suffix of G. Thus, if S is in G, some edge path from the root of the suffix tree begins with the label S. Simply by beginning at the root and walking down the correctly labeled edge from each node, we either find S or discover that it is not present. If searching for a number of patterns, each much shorter than G, a search time independent of n is a great time savings.

A detailed and accessible explanation of the $O(n)$ space bound and construction time can be found in [3].

2.3 Related Work

Approximate Suffix Tree Matching. Although the most obvious use of a suffix tree is to find exact matches, there are a number of known methods ([1,2,3,4,7,11,13,14] *etc.*) for using a suffix tree for approximate matching. However, these previous methods are generally not applicable to this problem domain, since the Gibbs sampling algorithm requires the search component to allow gaps, to be able to deal with a relative entropy score matrix (as opposed to, for example, a simple edit distance metric), and to be efficient even for the problem

of finding a single best-match string rather than an exhaustive listing of many good matches.

Known methods that do allow gapped alignment generally use dynamic programming in conjunction with a traversal of the suffix tree. A k-difference search—finding all strings whose edit distance from a given model is at most k—can be performed in $O(k \times n)$ time, but we are unaware of a result extending this time bound to an arbitrary score matrix instead of edit distance. Of work done on arbitrary score matrices, most concentrates on exhaustive listings, for example the "all-against-all" problem or the "P-against-all" problem, since in general there are few applications that require only a single best match, and the similar strings can be more efficiently found in bulk than individually.

Novel Motif Discovery with Suffix Trees. There are some recent algorithms ([6,9]) that use suffix trees to find novel motifs. [9] in particular applies to a similar problem domain to the one covered by this paper, that of exhaustively finding significant approximate motifs in a long sequence or collection of long sequences.[2] Rather than a sampling approach, this algorithm outputs a list of every string C and associated collection of strings A_C that meets the following conditions: A_C is a collection of at least T strings, every string in A_C is a substring of G differing from C in no more than k positions, and every such substring of G is included in A_C. C itself need not be in G. Here T and l are user-specified constants.

The Gibbs sampling algorithm presented here can have several advantages over this kind of exhaustive k-difference search in certain situations. By considering only patterns that actually occur in the sequence, it can look for motifs much longer than an enumerative approach can tolerate. It also inherently produces a classification of strings into best-probability motifs, rather than leaving it to the user to decide which of several k-difference sets a string belongs to is the best fit. Finally, dynamically varying the length of the strings is difficult with exhaustive algorithms, while a local search algorithm can easily change these parameters in the middle of a run, taking little or no more time than if the final length and size had been specified for that motif in advance.

On the other hand, there are disadvantages to a sampling method. One obvious difference—that with a sampling local-search method one can never be certain the program has found *all* the relevant motifs—does not turn out to be problematic in practice. In [8] we found that on the order of 90 out of 100 runs of the algorithm discovered the same handful of high-scoring motifs, which cover only a small fraction of the genome. This suggests that the sampling algorithm will, with high probability, find all the most significant motifs. However, exhaustive methods will probably remain more efficient to solve the problems of finding short motifs, ungapped motifs, or motifs with few errors, all of which will continue to be necessary for some biological problems. In particular, many regulatory sequences in DNA are fairly short (around 7 bases long), and could easily

[2] Although the algorithm in [9] does not allow gapped alignment as originally stated, the authors note in the paper that it can be modified to do so.

be found with an exhaustive k-difference search. In short, we hope to present an algorithm here that will improve considerably on other existing methods for some, but not all, situations in which novel gapped motifs must be found.

3 Overview of the Algorithm

3.1 Suffix Tree Construction

The preprocessing step uses Ukkonen's algorithm[12] as described in [3], a practical and fairly intuitive $O(n)$ algorithm for suffix tree construction.

3.2 Iteration

At each step in the algorithm, one string is discarded from the existing alignment and must be replaced with the best-scoring string from G. Just as before, the program uses dynamic programming to align the incoming candidate string with the current alignment. However, instead of using the letters of G in order as the input to the dynamic programming table, the new algorithm uses a traversal of the suffix tree to drive the dynamic programming. This allows the search to sometimes examine fewer than all n characters of the input. The method of traversing the tree will be described in section 4. Because the *worst case* performance of this traversal is a factor of $O(l)$ worse than the $O(ln)$ time the whole-sequence search requires—which can be significant, particularly for longer motifs—an arbitrary time limit can be set on the tree search, after which point, if no result has been found, the entire sequence is traversed as in the original algorithm. This ensures a guaranteed bound on the worst-case performance which can be as close to the original time as is necessary.

3.3 Termination

The algorithm ends when it is neither advantageous to change the size of the alignment (either length or number of instances), nor is it advantageous to replace any string with a different string or different alignment of the same string.

4 Details of Search Step

This section fills in the details of the search step from section 3.2, describing how the algorithm searches the suffix tree to find the best match in G. First, section 4.1 describes the basic search algorithm. Next, section 5 introduces the idea of k-difference matching and describes how a suffix tree can be used for a k-difference search. Finally, section 6 describes how to modify this concept for the search component of this algorithm in order to improve the worst case running time.

4.1 Suffix Tree Search

In order to align each string occurring in G to the current alignment, the search step uses a depth-first traversal of the suffix tree edges to directly drive the dynamic programming. As an edge is traversed, a dynamic programming (DP) table column is computed for each letter on the edge. Later, when the search backtracks up that edge (to try a different path out of its parent node), a DP column is deleted from the end of the DP table for each character on the edge. The traversal thus does $O(l)$ work constructing and later removing a column of l entries in the DP table for each character seen in the suffix tree search.

4.2 Improving Time Required

A complete traversal of the tree would see $n^2/2$ characters for an overall time of $O(n^2 l)$, since each suffix of G is in the tree in its entirety. However, several techniques ensure that the traversal takes less time in practice. The most significant of these is a branch-and-bound heuristic. First, note that for each alignment column i and character α (including a gap) there is a fixed score $\sigma_i(\alpha)$ for adding α to column i. The maximum score, $\sigma_{i,\max}$, for any column is thus well-defined and need only be computed once for each search of G. This can be conceptualized as a scoring matrix M_σ fixed by the current alignment, in which the $A+1$ entries in the i^{th} scoring matrix column are the scores of matching each character to the i^{th} alignment column. This means that for any given DP column, it takes $O(l)$ time (not significantly more than to compute the column itself) to check whether there is any set of characters we can see in the future that can cause this path to improve upon the best string found so far. If not, we can backtrack immediately instead of continuing down that branch.

4.3 Limiting Search Depth

The first benefit of this branch-and-bound technique is that, since insertions must be defined to have negative score in any effective motif scoring function, the traversal will seldom go significantly more than l characters down any branch of the tree. To quantify this statement more precisely, we will define the quasi-constant $\epsilon \geq 1$, which depends only on the scoring matrix M_σ:

Definition 1. *Let $d_i > 0$ be the largest absolute-value difference between any two scores in column i of M_σ—in other words, the maximum possible mismatch penalty for alignment column i (including for a gap). Let $I > 0$ be the minimum penalty for an insertion, that is, the absolute value of the maximum score for aligning any character against a blank alignment column. Then define $\epsilon = \epsilon(M_\sigma)$ to be $2 + \sum_i d_i/(lI)$*

If any string S is longer than $\epsilon \times l$, then consider the string S' consisting of the first l characters of S. At most, S' can have a mismatch in every column while S has none, for a maximum scoring penalty $\sum_i d_i$. Also, S must have more than $l\epsilon - l$ insertions, while S' can have at most l, for a scoring gain greater than

$I(l\epsilon - 2l) = \sum_i d_i$. This means that once the edge traversal has gone more than ϵl characters deep into the tree, no future set of characters seen can cause the path to outscore the string consisting of the first l characters on that path. The following observation therefore holds:

Fact 1. *No path through the tree longer than ϵl will be considered when using the branch-and-bound heuristic.*

Although ϵ does depend on the scoring matrix, for any given iteration of the search phase it can be considered a constant. In addition, for scoring functions used in practice to model gapped pattern discovery in nearly any domain, insertions are highly penalized, and so in practice ϵ is very small. Since ϵ is constant during any given search, future equations will include ϵ in O-notation bounds only when it appears in an exponent or when this makes the derivation clearer.

What the above implies is that using branch-and-bound makes the worst-case number of characters seen $O(nl)$ rather than $O(n^2)$, since at worst we examine $l\epsilon$ characters of each of the n suffixes of G for a maximum of $nl\epsilon$ characters seen. Although this leads to an $O(l^2 n\epsilon)$ overall worst-case running time, a factor of $l\epsilon$ more expensive than the original algorithm, it does mean that if we restrict the search phase to take time $O(ln)$ (with constant factor less than $l\epsilon$ and preferably less than 1) before doing a whole-sequence search, the tree search has a significant chance of finishing before the cutoff.

If l is small, of course, the worst-case behavior is better than this—if the search looks at every possible string of length $l\epsilon$, it still only encounters $A^{l\epsilon}$ total characters, creating a size-l DP column for each, for a worst-case time of $O(lA^{l\epsilon})$. For short patterns and a small alphabet, this is the relevant bound.

4.4 Limiting Search Breadth

The second, equally important, benefit of the branch-and-bound operation is that not every string of length $l\epsilon$ will be examined. For example, if the first string seen happens to be the highest-scoring possible string to match the alignment (the *consensus string*, C), then every future branch will be pruned (because any other string will be unable to beat this first string's score), and the search step will take $O(l)$ time. More generally, if a string differing only slightly from the consensus is seen early in the search, then the search will efficiently prune most of the unhelpful branches.

4.5 Dependence on Search Order

Although this latter creates a marked improvement in practice, it does not affect the worst-case bounds. Even if the exact consensus string C is in G, if it is on the last path the search traverses then it will not help eliminate any edge traversals. If the search happened to encounter each string in worst-to-best order, then no pruning would take place and the running time would be maximal. A second heuristic that helps this problem significantly in practice is to search the children

of each suffix tree node in order of how much the first letter on that edge improves the score along the DP table diagonal. For example, if a letter "T" would be most helpful next, the search looks first on the edge beginning with "T". Again, though, this greedy heuristic—however beneficial in practice—does nothing for the worst case bound. The best match string $S_{\max}$ in G might be an exact match to C except for a difference in the *first* character of $S_{\max}$, and then the search will be almost over—without the pruning benefit of knowing about $S_{\max}$'s score—before it encounters this best string.

The tree search performance will inevitably depend on the difference in score between $S_{\max}$ and the consensus string, as well as on the properties of the implicit scoring matrix defined by the scores $\sigma_i(\alpha)$, but need not do so in such an arbitrary way. The following sections describe a method for eliminating the dependence on chance and on the *order* in $S_{\max}$ of the differences from the consensus C.

5 *k*-Difference Matching

The k-difference matching problem is the question of, given a query string C—for our purposes, C will be the consensus string given the alignment scoring matrix—and sequence G, how to find all substrings of G whose edit distance from C is no more than k. In other words, find the strings in G that can be converted into C by applying no more than k insertions, deletions, and substitutions.

This problem is more tractable than the one we need to solve in the search step, it uses an edit distance metric instead of the more complicated relative entropy scores. However, the problems are related, and there are certain techniques for finding k-difference strings that would be helpful in this situation. Fortunately, we will see in section 6, there is a way to relate our problem to a more general form of k-difference.

5.1 *k*-Difference Suffix Tree Search

If we restrict ourselves for the moment to solving the simple k-difference problem, a similar algorithm to that described in section 4.1 can be used. Just as before, use the suffix tree to drive a dynamic programming alignment, but instead of using the best score seen so far as the criterion to prune a branch, use the score that a string must meet in order to be edit distance k from the consensus. If c_1 is the score for a match, and c_2 is the penalty for a mismatch, this threshold is just $(l-k)c_1 - kc_2$. Whenever the DP column scores are low enough that it can never beat the threshold, return back to the parent node as before, unwinding the dynamic programming. This will happen precisely when every current path in the DP table—that is, any path which goes through to the last DP column computed—implies more than k differences from the consensus.

An upper bound on the number of strings the k-difference search can consider is $O((A+1)^k (l\epsilon)^{k+1})$, with ϵ as defined in Definition 1. This is because it must investigate strings with any arrangement of k differences over $l\epsilon$ positions, each

of which can be any of $A+1$ characters (including a gap). For each character a column of l DP table entries is constructed, giving a bound of $O((A+1)^k l^{k+2})$.

If k is large, the $O(nl\epsilon)$ bound on characters visited described in section 4.1 is a more relevant limit. However, for k small, the above guaranteed search time can be much less than $l^2 n$. This is not the most efficient way to do k-difference searching, but faster k-difference methods can not be easily extended to arbitrary score matrices as will be done in section 6.

5.2 Iterative *k*-Difference

This technique can also be used to find the *best* string in G, even though the number of differences between $S_{\max}$ and C is not known in advance. Again, assume for now an edit-distance metric. Then the best string can be found by performing a 1-difference search, then a 2-difference search, and so forth until the search finds a string meeting the current criterion, that is, a string with only k differences from C. When this happens, that string can immediately be returned as the optimal, since it is certain that no other string has a better edit-distance score (or it would have been found on an earlier k-difference search pass).

Since each pass takes little time compared to the next pass, the work duplicated by starting over with each new k is negligible. In other words, performance is comparable to that when the value of k is known in advance. A bound on total time taken will be $O(T(k))$ where

$$T(k) = \min((A+1)^{k^*} l\epsilon^{k^*+2},\ \ (A+1)^{l\epsilon}) \tag{2}$$

and k^* is the number of differences between the best-match string in G and the consensus. The second term will dominate when $k*$ is close to l, and simply enumerates all strings of length $l\epsilon$.

Notice that, unlike in section 4.1 where the time taken depends on the order in which the strings are encountered in the tree, this bound depends only on how good a match to the alignment exists anywhere in G.

6 Applying *k*-Difference Techniques

6.1 Thresholds for an Arbitrary Score Matrix

The technique above can clearly improve the worst case performance if strings are scored using an edit-distance metric, but the question remains of how to extend the technique to an arbitrary scoring matrix. The basic idea here is to retain "k-difference" thresholds that must be met at each pass, and to somehow construct this series of thresholds so that even on an arbitrary scoring matrix there are guarantees on the behavior.

More specifically, because of the irregular scoring metric, a string with higher score than another string might nevertheless have differences in more alignment columns. In other words, there no longer exists a score threshold τ_k that will allow any possible string with at most k differences while preventing any string with

more than k differences from being examined. Since the worst-case performance bound in Equation 2 increases exponentially as k grows, we want to step up the thresholds slowly enough that if there exists a string in G sufficiently close to the consensus, it will be found before each pass becomes too long.

6.2 Calculating the Thresholds

First, remember that for each column i and character (or blank) α we know the score $\sigma_i(\alpha)$ of aligning α with column i. Define $\sigma_{i,0}$ to be the maximum score for column i, $\sigma_{i,1}$ the next highest score, and so on up to $\sigma_{i,A}$. Because which string the algorithm selects depends only on the differences between these scores, not on the absolute score, it is useful to define the set of differences. Let

$$\delta_{i,a} = \sigma_{i,a} - \sigma_{i,a+1}, 0 \leq a < A \tag{3}$$

be the difference between the score of matching the a^{th} best character to column i and that of matching the $(a+1)^{th}$ best.

Now look at the set of values $\delta_{i,0}$, the set of differences between the best and second-best score for each column. Define an array $\Delta_0[0\,..\,l]$ containing each $\delta_{i,0}$ sorted into increasing order.

Now compute a series of "threshold scores" τ_k. Begin with

$$\tau_0 = \sum_i \sigma_{i,0} \tag{4}$$

or, in other words, the maximum possible score. Then for $1 \leq k \leq l$ let

$$\tau_k = \tau_{k-1} - \Delta_0[k]. \tag{5}$$

Each subsequent threshold τ_k is then less than the previous threshold by an increment of the k^{th} smallest difference δ between the maximum and second-best score for some column.

The significance of the threshold score τ_k is that it subtracts the minimum possible penalty for k differences from the maximum score.[3] Of course, there may be strings with fewer than k differences that score below τ_k, since different differences have varying penalties. However, one deduction clearly follows from the definition of τ_k:

Fact 2. *If we look at all strings with scores above τ_k, each will have strictly fewer than k differences from C.*

[3] Strictly speaking, this is not quite accurate with Δ as defined above, since some column may have an expensive mismatch penalty that is worse than the score for an insertion. In reality, any values at the end of $\Delta[..]$ that are higher than the cheapest insertion penalty I for this score matrix will be replaced with I, since subtracting I from the threshold always permits an additional difference, namely an insertion, to take place in strings scoring above that threshold.

As we will see in the following search algorithm, this fact combined with the use of these thresholds will allow some control over the worst-case performance of the search step.

If the scoring matrix scored each mismatch for a given column with an equal penalty, then every possible string would have a score of at least τ_l, the smallest threshold calculated above, which is the score of adding the second-best character to each column. However, since this is not the case, additional thresholds need to be added, continuing down in small decrements so that the last threshold is the lowest possible score. We therefore define a set of arrays $\Delta_a, 1 \leq a < A$, similarly to Δ_0: each contains the values $\delta_{i,a}$ sorted in increasing order. After τ_l, the next set of decrements will reflect the differences between second-worst and third-worst matches to a column (Δ_1), then between the third-worst and fourth-worst, etc.:

$$\tau_{al+k} = \tau_{al+k-1} - \Delta_a[k], 1 \leq k \leq l, 0 \leq a < A \tag{6}$$

6.3 Iterative Search with Arbitrary Scores

The final search algorithm combines the iterative k-difference search of section 5.2 with this method of calculating threshold scores. As in section 5.2, a number of search passes are tried, decreasing the threshold score each time. Whereas before, the threshold for the k^{th} pass was a score of $(l-k)c_1 - kc_2$, the new threshold is τ_k. If, at the end of pass k, the best string seen so far has a score strictly greater than τ_k, then this string is returned as the best match, since the k^{th} pass has considered every string in the tree that can score better than τ_k.

6.4 Worst Case Performance

Fact 2 ensures that no string with more than k differences is seen on the k^{th} pass, giving an upper time bound of $T(k)$ as in Equation 2. However, there is no guarantee that a given string with k differences from C will be found on the k^{th} pass, since it might have a score lower than τ_k. All we can say is

Fact 3. *If the best match $S_{\max}$ has score higher than τ_k, then it will be found in time no greater than $T(k)$.*

This means that if any of the best few strings to match the alignment exist in G, then the search is sure to rapidly find an answer, unlike in the one-pass suffix tree search where the search might not be effectively bounded.

7 Empirical Results

We have implemented the algorithm described in this paper in approximately 10,000 lines of `C++` code, in order to demonstrate its practicability, but more importantly in order to apply the algorithm to biological problems of interest. We plan to use this program in the future to discover and analyze patterns in various real genomic data sets.

7.1 Practicality of Suffix Trees

Suffix trees, despite their usefulness, are often avoided in practice due to fear of their coding complexity or that the $O(n)$ space and time bounds hide large constants. Neither concern turns out to be well founded. This suffix tree class implementation requires under 2,000 lines of code, and the $O(n)$ construction involves only a small constant factor so that preprocessing a genome is practical, under a minute for a 400 kilobase dataset on a laptop computer. The space used for this implementation is in practice around $8 \times n$ integers, certainly not infeasible even on genome-sized datasets with the ever-increasing availability of memory.

7.2 Empirical Time Behavior

Choosing axes along which to measure the performance of the search algorithms is a nontrivial matter, since it depends on a variety of complicated factors—pattern length, information content of G (which affects the shape and properties of the suffix tree), the score of the actual best match to the alignment in G, and most significantly, the granularity of the scoring matrix.

A good estimator for how the algorithms perform in practice, though, is to run the Gibbs sampler on a sample of genomic data—improving the performance of which was, after all, the motivation behind this algorithm—and measure how each of the two suffix tree searches measure up to the original whole-sequence search. Averaging over many iterations in this way will not show instances of extreme behavior, but it will give a good idea of how the methods can be expected to behave in general.

For each of a range of motif lengths, 50 trials were run on the forward and reverse-complement noncoding regions of the *H.influenzae* genome, a dataset of about 400 kilobases. Each trial starts from a random set of 10 sequences of that length and runs the Gibbs sampling algorithm to completion. To isolate the effect of motif length on the running time, the component of the algorithm which tries to change the length and number of alignment strings has been disabled for this experiment, so each trial ends when the alignment has stabilized at the initial length (modulo length changes due to insertions) and number of instances. For each trial, the three methods are made to operate on the same alignment at each iteration during the trial, so that the results are directly comparable.

As mentioned in section 3.2, after the suffix-tree search has traversed a predetermined number d of characters it will switch methods to perform a whole-sequence search. In these trials d was set at $(0.1)n$, chosen because $(1.1)n$ seemed like a reasonable worst-case penalty to pay for the potential savings: low enough that this algorithm is not significantly less practical than whole-sequence search even on very long motifs that are unlikely to see a savings with the suffix tree search.

Figure 2 shows average percent of G examined per iteration over the 50 trials. The original whole-sequence search method, of course, is always fixed at 1.0—the whole dataset is examined precisely once per iteration. Only the two suffix

tree search algorithms are therefore drawn explicitly. The wide bars depict one standard deviation, and the outer lines show the minimum and maximum percent of G seen during any iteration. The averages for length 10 are approximately 0.0009 and 0.0011 respectively.

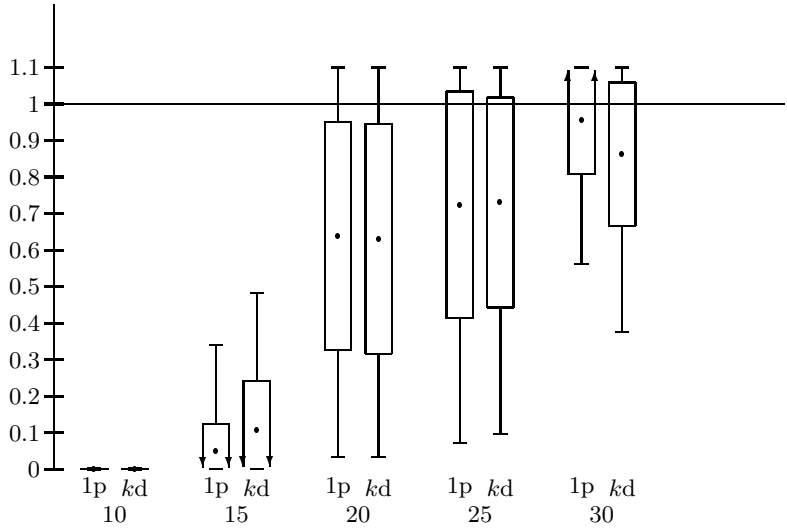


Fig. 2. Average percent of genome seen vs. length of motif during Gibbs sampler runs. Shows relative behavior of whole-sequence search (constant at 1.0), one-pass suffix tree search, and k-difference-style suffix tree search.

7.3 Practicality of Iterative k-Difference

Note that the performance of the iterative k-difference style search was consistently comparable to that of the 1-pass suffix tree search. Although any single search using the 1-pass method might take much longer, the heuristics used appear to be effective enough that this simpler method can consistently perform approximately as well when averaged over many trials.

One might be tempted to conclude that the iterative k-difference algorithm is useful more as an explanatory tool to discuss worst case performance than as a practical algorithm. However, the Gibbs sampling process has enough random factors that it is unlikely to repeatedly produce worst-case situations for the 1-pass algorithm, while other problem domains could do so. Since the k-difference search is little more complicated in code, and appears to perform with no penalty compared to the simpler algorithm, it makes sense to use the k-difference approach in practice to guard against worst-case scenarios.

8 Conclusions

In conclusion, this paper presents a novel use of suffix trees to improve the performance of a Gibbs sampling algorithm for discovering novel gapped motifs. While the worst case is a constant factor worse than using whole-sequence search, the average performance is significantly faster with the new algorithm. In particular, searching for a relatively short pattern is orders of magnitude faster using this method. Since the string lengths for which the suffix tree search shows a marked improvement over the whole-sequence search extends significantly beyond the string lengths at which it is practical to combinatorially enumerate all possible motifs, this algorithm will unquestionably be useful in using suffix trees to find mid-length gapped motifs in long sequences.

Acknowledgments. The author would like to thank Martin Tompa for his guidance and many helpful suggestions in preparing this work.

References

1. Chang, W.I., Lampe, J., Theoretical and Empirical Comparisons of Approximate String Matching Algorithms. *Proc. 3rd Symp. on Combinatorial Pattern Matching*, Springer LNCS 644, 175–84, 1992.
2. Fickett, J.W., Fast Optimal Alignment. *Nucl. Acids Res.*, **12**:175-80, 1984.
3. Gusfield, D., *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, New York, 1997.
4. Landau, G.M., Vishkin, U., Efficient string matching with k mismatches. *Theor. Comp. Sci.*, **43**:239-49, 1986
5. Lawrence, C., Altschul, S., Boguski, M., Liu, J., Neuwald, A., Wootton, J. Detecting Subtle Sequence Signals: A Gibbs Sampling Strategy for Multiple Alignment. *Science*, **262**:208–214, 8 October 1993.
6. Marsan, L., Sagot, M.F., Extracting Structured Motifs Using a Suffix Tree—Algorithms and Application to Promoter Consensus Identification. To appear in *Proceedings of RECOMB 2000.*
7. Meyers, E.W., An O(nd) Difference Algorithm and Its Variations. *Algorithmica*, **1**:251-66, 1986.
8. Rocke, E., Tompa, M., An Algorithm for Finding Novel Gapped Motifs in DNA Sequences. *Proceedings of the Second Annual International Conference on Computational Molecular Biology*, 228–233., New York, NY, March 1998.
9. Sagot, M-F., Spelling Approximate Repeated or Common Motifs Using a Suffix Tree. *Proceedings of LATIN*, 374–390, 1998.
10. Smith, T.F., Waterman, M.S., Identification of Common Molecular Subsequences. *J. Mol. Biol.*, **284**:1-18, 1995.
11. Ukkonen, E., Approximate String-Matching Over Suffix Trees. *Proc. 4th Symp. on Combinatorial Pattern Matching*, Springer LCNS 684, 228–42, 1993.
12. Ukkonen, E., On-line Construction of Suffix-Trees. *Algorithmica*, **14**:249–60, 1995.
13. Ukkonen, E., Algorithms for Approximate String Matching. *Information Control*, **64**:100–18, 1985.
14. Weiner, P., Linear Pattern Matching Algorithms. *Proc. of the 14th IEEE Symp. on Switching and Automata Theory*, 1–11, 1973.

Indexing Text with Approximate q-Grams

Gonzalo Navarro[1⋆], Erkki Sutinen[2], Jani Tanninen[2], and Jorma Tarhio[2]

[1] Dept. of Computer Science, University of Chile
gnavarro@dcc.uchile.cl
[2] Dept. of Computer Science, University of Joensuu, Finland
{sutinen,jtanni,tarhio}@cs.joensuu.fi

Abstract. We present a new index for approximate string matching. The index collects text q-samples, i.e. disjoint text substrings of length q, at fixed intervals and stores their positions. At search time, part of the text is filtered out by noticing that any occurrence of the pattern must be reflected in the presence of some text q-samples that match approximately inside the pattern. We show experimentally that the parameterization mechanism of the related filtration scheme provides a compromise between the space requirement of the index and the error level for which the filtration is still efficient.

1 Introduction

Approximate string matching is a recurrent problem in many branches of computer science, with applications to text searching, computational biology, pattern recognition, signal processing, etc. The problem is: given a long text $T_{1..n}$ of length n, and a (comparatively short) pattern $P_{1..m}$ of length m, both sequences over an alphabet Σ of size σ, retrieve all the text substrings (or "occurrences") whose *edit distance* to the pattern is at most k. The *edit distance* between two strings A and B, $ed(A, B)$, is defined as the minimum number of character insertions, deletions and replacements needed to convert A into B or vice versa. We define the "error level" as $\alpha = k/m$.

In the on-line version of the problem, the pattern can be preprocessed but the text cannot. The classical solution uses dynamic programming and is $O(mn)$ time [23]. It is based in filling a matrix $C_{0..m,0..n}$, where $C_{i,j}$ is the minimum edit distance between $P_{1..i}$ and a suffix of $T_{1..j}$. Therefore all the text positions j such that $C_{m,j} \leq k$ are the endpoints of occurrences of P in T with at most k errors. The matrix is initialized at the borders with $C_{i,0} = i$ and $C_{0,j} = 0$, while its internal cells are filled using

$$C_{i,j} \;=\; \text{if } P_i = T_j \text{ then } C_{i-1,j-1} \text{ else } 1 + \min(C_{i-1,j}, C_{i-1,j-1}, C_{i,j-1}) \quad (1)$$

which extends the previous alignment when the new characters match, and otherwise selects the best choice among the three alternatives of insertion, deletion

⋆ Work developed during postdoctoral stay at the University of Helsinki, partially supported by the Academy of Finland and Fundación Andes. Also supported by Fondecyt grant 1-000929.

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 350–363, 2000.

and replacement. Figure 1 shows an example. In an on-line searching only the previous column $C_{*,j-1}$ is needed to compute the new one $C_{*,j}$, so the space requirement is only $O(m)$.

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	**2**	**2**	**2**

Fig. 1. The dynamic programming matrix to search the pattern `"survey"` inside the text `"surgery"`. Bold entries indicate matching text positions when $k = 2$.

A number of algorithms improved later this result [20]. The lower bound of the on-line problem (proved and reached in [7]) is $O(n(k + \log_\sigma m)/m)$, which is of course $\Omega(n)$ for constant m.

If the text is large even the fastest on-line algorithms are not practical, and preprocessing the text becomes necessary. However, just a few years ago, indexing text for approximate string matching was considered one of the main open problems in this area [27,3]. Despite some progress in the last years, the indexing schemes for this problem are still rather immature.

There are two types of indexing mechanisms for approximate string matching, which we call "word-retrieving" and "sequence-retrieving". Word retrieving indexes [18,5,2] are more oriented to natural language text and information retrieval. They can retrieve every *word* whose edit distance to the pattern *word* is at most k. Hence, they are not able to recover from an error involving a separator, such as recovering the word `"flowers"` from the misspelled text `"flo wers"`, if we allow one error. These indexes are more mature, but their restriction can be unacceptable in some applications, especially where there are no words (as in DNA), where the concept of word is difficult to define (as in oriental languages) or in agglutinating languages (as Finnish).

Our focus in this paper is sequence retrieving indexes, which put no restrictions on the patterns and their occurrences. Among these, we find three types of approaches.

Neighborhood Generation. This approach considers that the set of strings matching a pattern with k errors (called $U_k(P)$, the pattern "k-neighborhood") is finite, and therefore it can be enumerated and each string in $U_k(P)$ can be searched using a data structure designed for *exact* searching. The data structures used have been the suffix tree [16,1] and DAWG [9,6] of the text. These data structures allow a recursive backtracking procedure for finding all the relevant text substrings (or suffix tree / DAWG nodes), instead of a brute-force enumeration and

searching of all the strings in $U_k(P)$. The approaches [12,15,26,8] differ basically in the traversal procedure used on the data structure.

Those indexes take $O(n)$ space and construction time, but their construction is not optimized for secondary memory and is very inefficient in this case (see, however, [10]). Moreover, the structure is very inefficient in space requirements, since it takes 12 to 70 times the text size (see, e.g. [11]). The simpler search approaches [12] can run over a suffix array [17,13], which takes 4 times the text size. With respect to search times, they are asymptotically independent on n, but exponential in m or k. The reason is that $|U_k(P)| = O(\min(3^m, (m\sigma)^k)$ [26]. Therefore, neighborhood generation is a promising alternative for short patterns only.

Reduction to Exact Searching. These indexes are based on adapting on-line filtering algorithms. Filters are fast algorithms that discard large parts of the text checking for a necessary condition (simpler than the matching condition). Most such filters are based on finding substrings of the pattern without errors, and checking for potential occurrences around those matches. The index is used to quickly find those pattern substrings without errors.

The main principle driving these indexes is that, if two strings match with k errors and $k+s$ non-overlapping samples are extracted from one of them, then at least s of these must appear unaltered in the other. Some indexes [24,21] use this principle by splitting the pattern in $k + s$ nonoverlapping pieces and searching these in the text, checking the text surrounding the areas where s pattern pieces appear at reasonable distances. These indexes need to be able to find any text substring that matches a pattern piece, and are based on suffix trees or indexing all the text q-grams (i.e. substrings of length q).

In another approach [25], the index stores the locations of all the text q-grams with a fixed interval h; these q-grams are called "q-samples". The distance h between samples is computed so that there are at least $k + s$ q-samples inside any occurrence. Thus, those text areas are checked where s pattern q-grams appear at reasonable distances among each other. Related indexes [15,14] are based on the intersections of two sets of q-grams: that in the pattern and that in its potential occurrence.

These indexes can also be built in linear time and need $O(n)$ space. Depending on q they achieve different space-time tradeoffs. In general, filtration indexes are much smaller than suffix trees (1 to 10 times the text size), although they work well for low error levels α: their search times are sublinear provided $\alpha = O(1/\log_\sigma n)$. A particularly interesting index with respect to space requirements is [25], because it does not index *all* the text q-grams. Rather, the q-samples selected are disjoint and there can be even some space among them. Using this technique the index can take even less space than the text, although the acceptable error level is reduced.

Intermediate Partitioning. Somewhat between the previous approaches are [19,22], because they do not reduce the search to exact but to approximate search of pattern pieces, and use a neighborhood generating approach to search

the pieces. The general principle is that if two strings match with at most k errors and j disjoint substrings are taken from one of them, then at least one of these appears in the other with $\lfloor k/j \rfloor$ errors. Hence, these indexes split the pattern in j pieces, each piece is searched in the index allowing $\lfloor k/j \rfloor$ errors and the approximate matches of the pieces are extended to complete pattern occurrences. The existing indexes differ in how j is selected (be it by indexing-time constraints [19] or by optimization goals [22]), and in the use of different data structures used to search the pieces with a neighborhood generating approach. They achieve search time complexities of $O(n^\lambda)$, where $\lambda < 1$ for low enough error levels ($\alpha < 1 - e/\sqrt{\sigma}$, a limit shown to be probably impossible to surpass in [4]).

The idea of intermediate partitioning has given excellent results [22] and was shown to be an optimizing point between the extremes of neighborhood generating (that worsens as longer pieces are searched) and reduction to exact searching (that worsens as shorter pieces are searched). However, it has only been exploited in one direction: taking the pieces from the pattern. The other choice is to take text q-samples ensuring that at least j of them lie inside any match of the pattern, and search the pattern q-grams allowing $\lfloor k/j \rfloor$ errors in the index of text q-samples. This idea has been indeed proposed in [25] as an on-line filter, but it has never evolved into an indexing approach.

This is our main purpose. We first improve the filtering condition of [25] and then show how an index can be designed based upon this principle. We finally implement the index and show how it performs. The index has the advantage of taking little space and being an alternative tradeoff between neighborhood generation and reduction to exact searching. By selecting the interval h between the q-samples, the user is able to decide which of the two goals is more relevant: saving space by a higher h or better performance for higher error levels, ensured by a lower h.

In particular, the scheme allows us to handle the problem posed by high error levels in a novel way: by adjusting parameters, we can do part of the dynamic programming already in the filtration phase, thus restricting the text area to be verified. In certain cases, this gives a better overall performance compared to the case where a weaker filtration mechanism results in a larger text area to be checked by dynamic programming.

2 The Filtration Condition

A filtration condition can be based on locating approximate matches of pattern q-grams in the text. In principle, this leads to a filtration tolerating higher error level as compared to the methods applying exact q-grams: an error breaking pattern q-gram u yields one error on it. Thus, the modified q-gram u' in an approximate match is no more an exact q-gram of the pattern, but an approximate q-gram of it. Hence, while u' cannot be used in a filtration scheme based on exact q-grams, it gives essential information for a filtration scheme based on approximate q-grams.

This is the idea we pursue in this section. We start with a lemma that is used to obtain a necessary condition for an approximate match.

Lemma 1. *Let A and B be two strings such that $ed(A, B) \leq k$. Let $A = A_1 x_1 A_2 x_2 ... x_{j-1} A_j$, for strings A_i and x_i and for any $j \geq 1$. Then, at least one string A_i appears in B with at most $\lfloor k/j \rfloor$ errors.*

Proof: since at most k edit operations (errors) are performed on A to convert it into B, at least one of the A_i's get no more than $\lfloor k/j \rfloor$ of them. Or put in another way, if each A_i appears inside B with not less than $\lfloor k/j \rfloor + 1 > k/j$ errors, then the whole A needs strictly more than $j \cdot k/j = k$ errors to be converted into B.

This shows that an approximate match for a pattern implies also the approximate match of some pattern pieces. It is worthwhile to note that it is possible that $j \cdot \lfloor k/j \rfloor < k$, so we are not only "distributing" the errors across pieces but also "removing" some of them. Figure 2 illustrates.

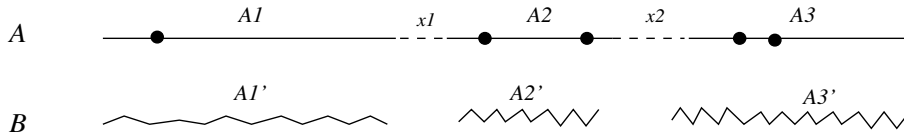


Fig. 2. Illustration of Lemma 1, where $k = 5$ and $j = 3$. At least one of the A_i's has at most one error (in this case A_1).

Lemma 1 is used by considering that the string B is the pattern and the string A is its occurrence in the text. Hence, we need to extract j pieces from each potential pattern occurrence in the text.

Given some q and $h \geq q$, we extract one text q-gram (called a "q-sample") each h text characters. Let us call d_r the q-samples, $d_1, d_2, \ldots, d_{\lfloor \frac{n}{h} \rfloor}$, where $d_r = T_{h(r-1)+1..h(r-1)+q}$.

We need to guarantee that there are at least j text samples inside any occurrence of P. As an occurrence of P has minimal length $m - k$, the resulting condition on h is

$$h \leq \left\lfloor \frac{m - k - q + 1}{j} \right\rfloor \tag{2}$$

(note that h has to be known at indexing time, when m and k are unknown, but in fact one can use a fixed h and adjust j at query time).

Figure 3 illustrates the idea, pointing out another fact not discussed until now. If the pattern P matches in a text area containing a *test sequence* of q-samples $D_r = d_{r-j+1} \ldots d_r$, then d_{r-j+i} must match inside a specific substring Q_i of P. These *pattern blocks* are overlapping substrings of P, namely $Q_i = P_{(i-1)h+1...ih+q-1+k}$.

A *cumulative best match distance* is computed for each D_r, as the sum of the best distances of the involved consecutive text samples d_{r-j+i} inside the Q_i's.

More formally, we compute for D_r

$$\sum_{1 \le i \le j} bed(d_{r-j+i}, Q_i)$$

where

$$bed(u, Q) = \min_{Q' < Q} ed(u, Q')$$

(where $<$ denotes substring of). That is, $bed(u, Q)$ gives the best edit distance between u and a substring of Q. The text area corresponding to D_r is examined only if its cumulative best match distance is at most k.

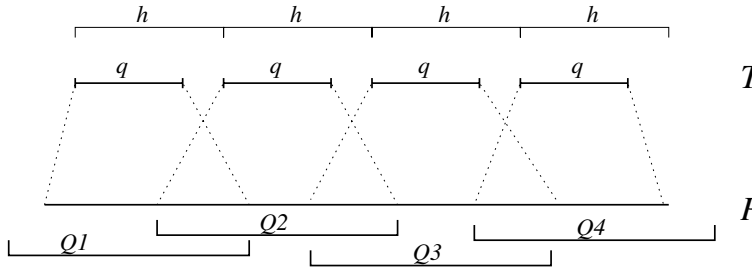


Fig. 3. Searching using q-samples, showing how the four relevant text samples at each position are aligned with the corresponding pattern blocks.

The algorithm works as follows. Each *counter* M_r, corresponding to the sequence $D_r = d_{r-j+1} \dots d_r$, indicates the number of errors produced by D_r. The counters are initialized to $M_r = j(e+1)$, were $q \ge e \ge \lfloor k/j \rfloor$ is unspecified by now. That is, we start by assuming that each text q-sample yields enough errors to disallow a match. Later, we can concentrate only on those that can be found in the pattern with at most e errors.

Now, for each pattern block Q_i, we obtain its "q-gram e-environment", defined as

$$U_e^q(Q_i) = \{u \in \Sigma^q, bed(u, Q_i) \le e\}$$

which is the set of possible q-grams that appear inside Q_i with at most e errors. Now, each $d_r \in U_e^q(Q_i)$ represents a text q-sample that matches inside pattern block Q_i. Therefore, we update all the counters

$$M_{r+j-i} \quad \leftarrow \quad M_{r+j-i} - (e+1) + bed(d_r, Q_i)$$

Finally, all the text areas whose counter $M_r \le k$ are checked with dynamic programming. Of course, it is not necessary to maintain all the counters M_r, since they can implicitly be assumed to be initialized at $j(e+1)$ until a text q-sample participating in D_r is found in some $U_e^q(Q_i)$.

3 Finding Approximate q-Grams

In this section we focus on the problem of finding all the text q-samples that appear *inside* a given pattern block Q_i, that is, find all the r such that $d_r \in U_e^q(Q_i)$. The first observation is that it is not necessary to generate all $U_e^q(Q_i)$, since we are interested only in the text q-samples (more specifically, in their positions). Rather, we generate

$$I_e^q(Q_i) = \{r \in 1..\lfloor n/h \rfloor, bed(d_r, Q_i) \leq e\}$$

The idea is to store all the different text q-samples in a trie data structure, where the leaves store the corresponding r values. A backtracking approach is used to find all the leaves of the trie that are relevant for a given pattern block Q_i, i.e. those that match inside Q_i with at most e errors.

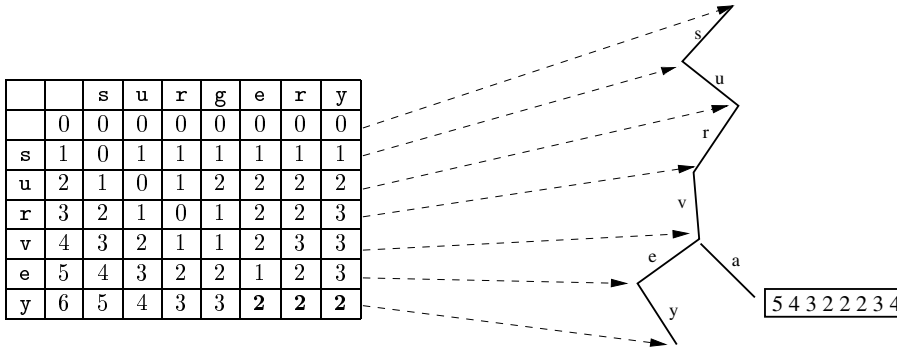
From now on we use $Q = Q_i$ and use i for other purposes. If considering a specific text q-sample $S = s_1 \dots s_q$ (corresponding to some d_r), the problem is solved by the use of the dynamic programming algorithm explained in the Introduction, where the text is the pattern block Q and the pattern is the text q-sample S. That is, we fill a matrix $C_{0..q,0..|Q|}$ such that $C_{i,\ell}$ is the smallest edit distance between $S_{1..i}$ and a suffix of $Q_{1..\ell}$. When this matrix is filled, we have that the text q-sample S is relevant if and only if $C_{q,\ell} \leq e$ for some ℓ (in other words, S matches somewhere inside Q with at most e errors). In a trie traversal of the q-samples, the characters of S are obtained one by one, so this matrix will be filled row-wise instead of the typical on-line column-wise filling.

The algorithm works as follows. We perform an exhaustive search on the trie, starting at the root and entering into all the children of each node. At each moment, if we are in a trie node representing a prefix S' of some text q-samples, we keep $C_{|S'|,\ell}$ for all ℓ, i.e. the current row of the dynamic programming matrix. Upon entering into the children of the current node following an edge labeled with the letter c, a new row of C is computed from the current one using c as the next pattern letter. When we reach the leaf nodes of the trie (at depth q) we check in the last row of C whether there is a cell with value at most e, in which case the corresponding text q-sample is reported. Note that since we only store the rows of the ancestors of the current node at each time, the total space requirement for the backtrack is just $O(|Q|q) = O(mq)$.

As we presented it, it seems that we traverse all the nodes of the trie. However, some pruning can be done. As all the values from a row to the next are nondecreasing, we know that if all the values of a row are larger than e then this will keep true in descendant nodes. Therefore, at that point we can abandon that branch of the trie without actually considering its subtree.

Figure 4 shows an example, using $Q =$ `"surgery"` and $S =$ `"survey"`. If $e = 1$ then the alternative path shown can be abandoned immediately since all its entries are larger than 2.

An alternative way to consider the problem is to model the search with a non-deterministic automaton (NFA). Consider the NFA for $e = 2$ errors shown in Figure 5. It is built for a fixed pattern block Q and is fed with the characters



		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	**2**	**2**	**2**

Fig. 4. The dynamic programming algorithm run over the trie of text q-samples. We show just one path and one additional link.

of a text q-gram S. Every row denotes the number of errors seen (the first row zero, the second row one, etc.). Every column represents matching a prefix of S. Horizontal arrows represent matching a character (i.e. if the characters of S and Q match, we advance in S and in Q). All the others increment the number of errors (move to the next row): vertical arrows insert a character in S (we advance in Q but not in S), solid diagonal arrows replace a character (we advance in Q and S), and dashed diagonal arrows delete a character from S (they are ε-transitions, since we advance in S without advancing in Q). The initial set of ε-transitions allow a match of S to start anywhere inside Q. The q-gram prefix S' of S matches inside Q as long as there is an active state after considering all the characters of S'.

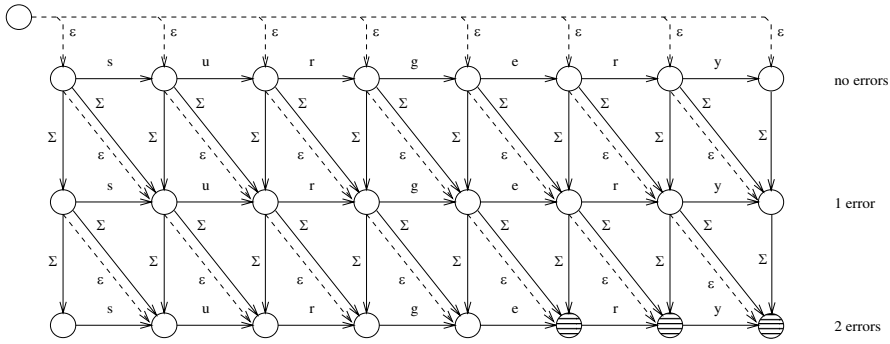


Fig. 5. An NFA for approximate string matching inside the pattern block $Q =$ `"surgery"` with two errors. The shaded states are those active after considering the text q-sample `"survey"`.

In [4,22] it is shown that this NFA can be simulated using bit-parallelism, mapping each state to a bit in a computer word and updating all the states of a single computer word in $O(1)$ operation. The total time needed is $O(|Q|e/w)$ per node of the trie, where w is the number of bits in the computer word (cf. the dynamic programming $O(|Q|)$ time per trie node). The only change necessary to the simulation technique used in [22] is to start with all the states active to account for the initial ε-transitions, absent in [22]. Checking that there is an active state in the automaton is easily done (so the branch of the trie can be abandoned if there are no more active states). Finally, checking the exact number of errors of a match (i.e. finding the smallest row with an active state) is easily done in $O(e)$ time using bit masks. We use this simulation in our implementation.

4 The Parameters of the Problem

The value of e has been left unspecified in the previous development. This is because there is a tradeoff involved. If we use a small e value, then the search of the e-environments will be cheaper, but as we have to assume that the text q-samples not found have only $e+1$ errors (which may underestimate the real number of errors it has), so some unnecessary verifications will be carried out. On the other hand, using larger e values gives more exact estimates of the actual number of errors of each text q-sample and hence reduces unnecessary verifications in exchange for a higher cost to search the e-environments.

As the cost of this search grows exponentially with e, the minimal $e = \lfloor k/j \rfloor$ can be a good choice. With the minimal e the sequences D_{r-j+i} are assumed to have $j(\lfloor k/j \rfloor + 1)$ errors, which can get as low as $k+1$. In that particular case we can avoid the use of counters, since every text q-gram d_{r-j+i} found inside Q_i will trigger a verification in D_r.

It is interesting to consider the interplay between the different remaining parameters h, q and j. Equation (2) relates these parameters, introducing also m and k in the condition. In general m and k are not known at index construction time, while h and q have to be determined at that moment. Therefore, j must be adjusted at search time in order to make Eq. (2) hold. For a given query, j has a maximum acceptable value. As j grows, longer test sequences with less errors per piece are used, so the cost to find the relevant q-samples decreases but the amount of text verification increases.

So j and e permit finding the best compromise between both parts of the search. On the other hand, q and h determine the space usage of the index, which is in the worst case $O(\sigma^q + n/h)$. Having a smaller index puts restrictions in the allowed j values and, indirectly, on e.

5 Experimental Results

In the following experiments, the texts have been generated according to the symmetric Bernoulli model where each character occurs at the same probability, independently of other characters, like its predecessors.

Table 1 shows how the error level increases the number of processed columns, in cases of random text in 4- and 20-character alphabets. The behavior in other alphabets is similar but a bigger alphabet implies a higher tolerated error level. Note that some fluctuations in the number of processed columns are due to the change in the value of e.

			$\sigma = 4$	$\sigma = 20$
k	j	e	Columns	Columns
$0 \ldots 3$	5	0	0.0	0.0
4	5	0	7.5	0.0
5	5	1	0.0	0.0
6	4	1	33.9	0.0
7	4	1	93.7	0.1
8	4	2	97.0	0.0
9	4	2	100.0	0.0
10	4	2	100.0	0.2
11	4	2	100.0	9.0
12	3	4	100.0	99.9
13	3	4	100.0	100.0

Table 1. Processed columns (in per cent) for $m = 40$, $q = h = 6$, and $n = 100,000$.

Altering the number of q-samples in test sequences D_r, i.e., the value of j, is related to changes in the values of h and q. This phenomenon lets us also to achieve more efficient filtration for higher error levels. Compare the results in Table 2 to those in Table 1.

k	h	q	j	e	Cols
6	7	7	4	1	6.0
7	8	8	3	2	44.2
8	8	8	3	2	95.6

Table 2. Processed columns (in per cent) for $\sigma = 4$, $m = 40$, and $n = 100,000$, for different values of h, q and j.

Table 3 shows how our scheme allows to do part of the dynamic programming already in the filtration phase, by traversing the trie structure and evaluating minimum edit distances between q-samples and substrings of pattern blocks. This is based on increasing the value of e. Although the results seem promising at the first sight, one has to remember that a small portion of processed columns does not necessarily imply a shorter processing time. In fact, the optimal setting for

e depends on several factors, like the length of the text and the implementation of the trie.

e	Columns	Traversed nodes
1	33.3	8,061
2	11.6	19,304
3	9.6	21,500
4	7.1	21,544
5	4.9	21,544
6	2.1	21,544

Table 3. Processed columns (in per cent) and the number of traversed nodes of the q-sample trie for $\sigma = 4$, $m = 40$, $k = 6$, $q = h = 6$, $j = 4$, and $n = 100,000$, for different values of e.

The distance h between the q-samples is crucial for the space requirement of the index. Table 4 shows that a lower interval h, and thus, a larger index, yields a more efficient filtration, as indicated, for example, in the number of processed columns.

h	j	Columns
7	11	100.0
6	8	99.8
5	6	90.7
4	5	14.2
3	4	0.1

Table 4. Processed columns (in per cent) with a decreasing h, for $\sigma = 4$, $m = 40$, $k = 5$, $q = 3$, $e = 3$, $n = 100,000$. Note that the parameter j has to be adjusted according to h.

Since the index of the presented approach only stores non-overlapping q-samples, its space requirement is small, and can be kept below the size of the text [25]. This should be kept in mind when the performance is compared to other related approaches. Table 5 shows that the new approach works for a small error level almost as efficiently as its competitor [22] which, however, consumes more space; in fact, four times as much as the text does. It is obvious that an index which stores only a fraction of text portions cannot compete with one with more information on the text.

Let us conclude by briefly discussing how the space consumption of our index depends on the sampling interval h. The standard implementation of a q-gram index stores all the locations of all the q-grams of the text. Since the number

k	Alg. A	Alg. B
4	0.0	1.0
5	0.3	1.0
6	5.3	1.1
7	30.2	1.2
8	81.1	22.9
9	99.5	23.6

Table 5. Processed columns (in per cent) for relatively low error levels. The new approach, denoted as A, collects *non-overlapping* q-samples, and an intermediate partitioning approach [22], denoted by B, stores *all* the text pieces which need to be searched for. The parameters are as follows: $\sigma = 4$, $m = 40$, $q = h = 6$ for algorithm A, $j = 4$, $e = 6$, and $n = 100,000$.

of q-grams in a text of length n is $n - q + 1$ and storing a position takes $\log n$ bits (without compression), the overall space consumption is $n \log n$ (q is small compared to n). Let us define a *space saving factor* v_r as the space requirement ratio between our method and the standard approach, i.e.

$$v_r = \frac{\frac{n}{h} \log \frac{n}{h}}{n \log n} \approx \frac{1}{h} \text{ (for large } n\text{)}.$$

Table 6 shows how the space saving factor decreases with an increasing h.

h	v_r
1	1.000
2	0.470
3	0.302
4	0.220
5	0.172
6	0.141
7	0.119
8	0.102
9	0.090
10	0.080

Table 6. *Space saving factor* v_r *for* $n = 100,000$.

6 Conclusions

We have introduced a static pattern matching scheme which is based on locating approximate matches of the pattern substrings among the q-samples of the text.

The mechanism breaks the fixed division of pattern matching into two phases, filtration and checking, where dynamic programming belongs only to the last phase. In our approach, it is possible to share dynamic programming between these phases by setting appropriate parameters. This is an important feature, since it makes it possible to tune the algorithm according to the particular problem instance. In some cases, saving space is a critical issue, whereas a high error level requires a more dense index. At the moment, the presented approach presumes non-overlapping q-samples ($h \geq q$). However, this is a question of parameterization. In the future, we will evaluate the impact of these parameters in different environments and problem instances, and enhance the scheme to allow also overlapping q-samples.

References

1. A. Apostolico and Z. Galil. *Combinatorial Algorithms on Words*. Springer-Verlag, New York, 1985.
2. M. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In *Proc. WSP'97*, pages 2–20. Carleton University Press, 1997.
3. R. Baeza-Yates. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, volume I, pages 465–476. Elsevier Science, September 1992.
4. R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.
5. R. Baeza-Yates and G. Navarro. Block-addressing indices for approximate text retrieval. *J. of the American Society for Information Science (JASIS)*, 51(1):69–82, January 2000.
6. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. Chen, and J. Seiferas. The samllest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.
7. W. Chang and T. Marr. Approximate string matching and local similarity. In *Proc. CPM'94*, LNCS 807, pages 259–273, 1994.
8. A. Cobbs. Fast approximate matching using suffix trees. In *Proc. CPM'95*, pages 41–54, 1995. LNCS 937.
9. M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.
10. M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *Proc. SODA'98*, pages 174–183, 1998.
11. R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. In *Proc. WAE'99*, LNCS 1668, pages 30–42, 1999.
12. G. Gonnet. A tutorial introduction to Computational Biochemistry using Darwin. Technical report, Informatik E.T.H., Zuerich, Switzerland, 1992.
13. G. Gonnet, R. Baeza-Yates, and T. Snider. *Information Retrieval: Data Structures and Algorithms*, chapter 3: New indices for text: Pat trees and Pat arrays, pages 66–82. Prentice-Hall, 1992.
14. N. Holsti and E. Sutinen. Approximate string matching using q-gram places. In *Proc. 7th Finnish Symposium on Computer Science*, pages 23–32. University of Joensuu, 1994.
15. P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. of MFCS'91*, volume 16, pages 240–248, 1991.

16. D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.
17. U. Manber and E. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, pages 935–948, 1993.
18. U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. USENIX Technical Conference*, pages 23–32, Winter 1994.
19. E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, Oct/Nov 1994.
20. G. Navarro. A guided tour to approximate string matching. Technical Report TR/DCC-99-5, Dept. of Computer Science, Univ. of Chile, 1999. To appear in *ACM Computing Surveys*.
`ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/survasm.ps.gz`.
21. G. Navarro and R. Baeza-Yates. A practical q-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1(2), 1998. `http://www.clei.cl`.
22. G. Navarro and R. Baeza-Yates. A new indexing method for approximate string matching. In *Proc. CPM'99*, LNCS 1645, pages 163–186, 1999.
23. P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.
24. F. Shi. Fast approximate string matching with q-blocks sequences. In *Proc. WSP'96*, pages 257–271. Carleton University Press, 1996.
25. E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proc. CPM'96*, LNCS 1075, pages 50–61, 1996.
26. E. Ukkonen. Approximate string matching over suffix trees. In *Proc. CPM'93*, pages 228–242, 1993.
27. S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, October 1992.

Simple Optimal String Matching Algorithm
(Extended Abstract)

Cyril Allauzen and Mathieu Raffinot

Institut Gaspard-Monge, Université de Marne-la-Vallée, Cité Descartes
Champs-sur-Marne, 77454 Marne-la-Vallée Cedex 2, France
{allauzen,raffinot}@monge.univ-mlv.fr

Abstract We present a new string matching algorithm linear in the worst case (in $O(m+n)$ where n is the size of the text and m the size of the searched word, both taken on an alphabet Σ) and optimal on average (with equiprobability and independence of letters, in $O(m+n\log_{|\Sigma|} m/m)$). Of all the algorithms that verify these two complexities, our is the simplest since it uses only a single structure, a suffix automaton. Moreover, its preprocessing phase is *linearly dynamical*, *i.e.* it is possible to search the words p_1, then $p_1p_2, p_1p_2p_3, \ldots, p_1p_2p_3\ldots p_i$ with $O(\sum |p_i|)$ total preprocessing time. Among the algorithms that verify this property (for instance the Knuth-Morris-Pratt) our algorithm is the only one to be optimal on average.

1 Introduction

The string-matching problem is to find all the occurrences of a given pattern $p = p_1p_2\ldots p_m$ in a large text $T = t_1t_2\ldots t_n$, both sequences of characters from a finite character set Σ. This problem is fundamental in computer science and is the basic part of many others, like text retrieval, symbol manipulation, computational biology, network security.

Several algorithms exist to solve this problem. One of the most famous, and the first having linear worst-case behavior, is Knuth-Morris-Pratt (KMP) [10]. The search is done by scanning the text character by character, and for each text position i remembering the longest prefix of the pattern which is also a suffix of $t_1\ldots t_i$. This approach is $O(n)$ worst-case time but it needs to scan all characters in the text, independently of the pattern. A second classical algorithm of Boyer and Moore (BM)[2] allows to skip characters. The search is done inside a window of length m, ending at position i in the text. It searches backwards a suffix of $t_1\ldots t_i$ which is also a suffix of the pattern. If the suffix is the whole pattern a match is reported. Then the window is shifted to the next occurrence of the suffix in the pattern.

An average lower bound in $O(n\log(m)/m)$ (with equiprobability and independence of letters) has been proven by A. C Yao in [12].

The *Backward Dawg Matching* (*BDM*) [7,11] is the first algorithm that reached this average bound. It uses a suffix automaton in a backward search

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 364–374, 2002.

(of the same kind as the one of *Boyer-Moore*). However, its worst case complexity is $O(nm)$. Several techniques can be used in order to make its worst case linear, but only by combining it with an algorithm like *KMP*. This makes it much more difficult to implement, increases its memory use and also the number of characters of the text that have to be read twice.

In this extended abstract, we propose a new algorithm which is both optimal on average and worst case linear. It only uses a suffix automaton. We obtain a much simpler algorithm, that requires less memory and that avoids a lot of characters to be read twice.

Moreover, the preprocessing phase of our algorithm is *linearly dynamical*, *i.e* it is possible to search the words p_1, then $p_1p_2, p_1p_2p_3, \ldots, p_1p_2p_3 \ldots p_i$ with $O(\sum |p_i|)$ total preprocessing time. Among the algorithms that verify this property (for instance the Knuth-Morris-Pratt) our algorithm is the only one to be optimal on average. This property is useful, for instance, when compressing a file by the LZ77 compression algorithm [13] using string matching. The LZ77 algorithm runs as follow. If we assume that a prefix $t_1 \ldots t_i$ of a text $T = t_1 \ldots t_n$ has already been compressed, the next step is to find the longest prefix of the rest of the text, *i.e.* $t_{i+1} \ldots t_n$ that appears already in $t_1 \ldots t_i$. Once this prefix has been found, it is coded as a reference to its position in $t_1 \ldots t_i$. There exist many different ways to find this longest prefix, and one of them is to search for the first occurrence of the character p_{i+1} in $t_1 \ldots t_i$, and if found in position l, to search for the first occurrence of $t_{i+1}t_{i+2}$ in $t_l \ldots t_i$, and so on. This approach is quite rough (using a suffix tree, it is possible to obtain a total compression of all the text in linear time), but requires very little memory in comparison with the others. The *linearly dynamical* property insures that the total preprocessing time needed to search this longest prefix u is linear in the size of u.

We now define notions and definitions that we need along this paper. A *word* p is a finite sequence $p = p_1p_2 \ldots p_m$ of letters taken in an alphabet Σ. We will keep the notation p along this paper to denote the word on which we are working. A word $x \in \Sigma^*$ is a *factor* of p if and only if p can be written as $p = uxv$ with $u, v \in \Sigma^*$. We denote Fact(p) the set of all the factors of word p. A factor x of p is a *prefix* (resp. a *suffix*) of p if $p = xu$ (resp. $p = ux$) with $u \in \Sigma^*$. The set of all the prefixes of p is denoted by Pref(p) and the one of all the suffixes Suff(p). We say that x is a *proper factor* (resp. *proper prefix*, *proper suffix*) of p if x is a factor (resp. prefix, suffix) of p distinct from p and from the empty word ϵ.

Finally, we define for $u \in \text{Fact}(p)$ the set $\text{endpos}_p(u) = \{i \mid p = wup_{i+1} \ldots p_m\}$.

2 Suffix Automaton

The *suffix automaton* of a word p is the minimal deterministic automaton (not necessarily complete) that recognizes the set of suffixes of p. We denote it $\mathcal{AS}(p)$. The figure 1 represents $\mathcal{AS}(baabbaa)$.

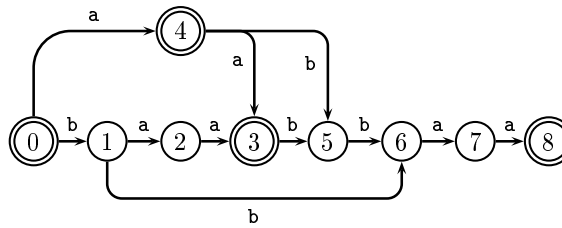


Figure 1. Suffix automaton of the word `baabbaa`

More formally, the suffix automaton is built from the following relation.

Definition 1. *Let u and v be two words of Σ^*. We define the relation $\sim_p$ by:*

$$u \sim_p v \text{ if and only if } endpos_p(u) = endpos_p(v)$$

It is not difficult to verify that this relation is an equivalence relation. Moreover, lemma 1 shows that this relation is compatible with the concatenation operation.

Lemma 1. *Let $u, v \in Fact(p)$ such that $u \sim_p v$. Then $u\sigma \sim_p v\sigma$.*

We denote $Cl_{\sim_p}(u)$ the equivalence class of the word $u \in \sigma^*$ for relation $\sim_p$.

From congruence $\sim_p$, we can build the suffix automaton $\mathcal{AS}(p) = \{S, \delta, I, T\}$ defined by:

- S is the set of states of the automaton. We take for S the set of equivalence classes for relation $\sim_p$, excluding the one that corresponds to the words that are not factors of p.
- $\delta : S \times \Sigma \to S$ is the transition function defined for $q \in S$ and $a \in \Sigma$ by: $\delta(q, a) = p$, with p such that there exists a word $u \in \Sigma^*$ for which $q = Cl_{\sim_p}(u)$ and $p = Cl_{\sim_p}(ua)$.
- $I \in S$ is the initial state, defined by $I = Cl_{\sim_p}(\epsilon)$.
- $T \subset S$ is the set of terminal states, defined by: $q \in S$ belongs to T if and only if there exists a suffix s of p such that $q = Cl_{\sim_p}(s)$.

Lemma 2. *The automaton $\mathcal{AS}(p)$ defined above is the minimal automaton that recognizes the set of suffixes of p.*

Lemma 3. *Let u and v be two distinct words recognized by $\mathcal{AS}(p)$ in the same state q. Then one is suffix of the other.*

Corollary 1. *The minimal length word u recognized in state q of $\mathcal{AS}(p)$ is suffix of all the words recognized by $\mathcal{AS}(p)$ in q.*

We add to the previous definition of the automaton a supply function s_p : $\text{Fact}(p) \setminus \epsilon \to \text{Fact}(p)$ defined for every non-empty factor v of p by:

$$s_p(v) = \text{ the longest } u \in \text{Suff}(v) \text{ such that } u \not\sim_p v$$

We show that the supply function is compatible with the congruence $\equiv_{\text{Suff}(p)}$. It is the object of the following lemma.

Lemma 4. *Assume that* $p \neq \epsilon$, *and let* $u, v \in Fact(p) \setminus \{\epsilon\}$. *If* $u \sim_p v$, *then* $s_p(u) = s_p(v)$.

For every non-initial state q, we call *supply state* of q, denoted by $S_p(q)$, the equivalence class of $s_p(u)$.

Figure 2 represents the suffix automaton $\mathcal{AS}(baabbaa)$ with its supply function (represented by dashed arrows).

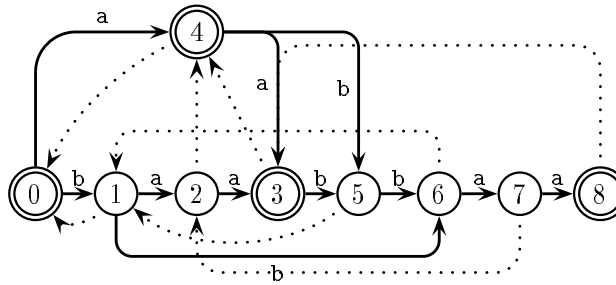


Figure 2. Suffix automaton with its supply function (in dashed arrows) of the word `baabbaa`

Some properties of the supply function of a suffix automaton will be used in the following.

Lemma 5. *Let* q *a non-initial state of the suffix automaton* $\mathcal{AS}(p = p_1 \dots p_m)$ *such that* $s = S_p(q)$. *Then the longest word* v *recognized in* s *is a suffix of every word recognized in* q.

We denote $CS_p(q) = \{q, S_p(q), S_p(S_p(q)), \dots, I\}$ the supply path of state q in $\mathcal{AS}(p)$.

Lemma 6. *Let* q *be a state of* $\mathcal{AS}(p)$ *such that there is no outgoing transition from* q *by a character* $a \in \Sigma$, *and let* o *the first state encountered on* $CS_p(q)$ *(going down from* q *to* I*) that has an outgoing transition by* a. *If such a state exists, then the longest word* v *recognized in* o *is the longest suffix of one of the words recognized in* q *such that there is an occurrence of* va *in* p.

The suffix automaton of a word $p = p_1 p_2 \dots p_m$ can be build online (*i.e.* by reading the letters p_1, then p_2, until p_m, by modifying the build automaton with each addition) in $O(m)$. We do not present this algorithm, but it can be found in [3,6].

For each state q of the suffix automaton, we define Length(q) to be the length of the longest word recognized in q. This value associated to each state is used in the online construction algorithm of the suffix automaton [3,6] and in the string matching algorithms we present in the following.

It is important to notice that, by construction, the label of any path in the suffix automaton of p is a factor of p and, conversely, every factor of p labels a path starting in the initial state in the automaton. The suffix automaton can then be used to recognize the set of factors of p in the same time as the set of the suffixes of p.

3 Forward Search

The suffix automaton is used in [3,6] for a forward search of a word $p = p_1 \dots p_m$ in a text $T = t_1 \dots t_n$, *i.e.* by reading the letters of the text one after the other. The text T and the search word p are both taken on an alphabet Σ. The algorithm is called *Forward Dawg Matching* (FDM). It is based on the following idea. For each position *pos* in the text, we compute the length of the longest factor of p that is a suffix of the prefix $t_1 \dots t_{pos}$ (see figure 3).

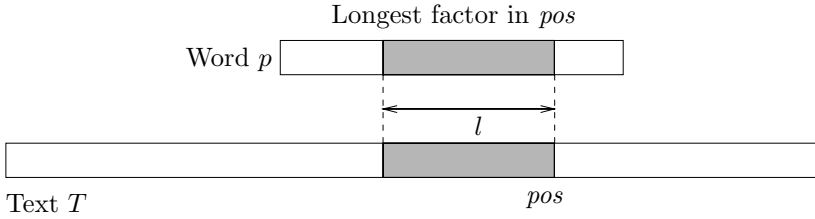


Figure 3. Current situation in the FDM

We first build the suffix automaton $\mathcal{AS}(p)$ and initialize the current length l to 0. We then read the characters of the text one by one with the automaton, updating the length l for each letter. Assume that we read the text until *pos* and that we arrived in a state q with some value l. We read character t_{pos+1} and we update l and q in the following way.

1. If there is a transition from q by t_{pos+1} to a state f, then the current state becomes f and l is incremented by 1.
2. If not, we go down the supply path of q until we found a state which has an existing transition by t_{pos+1}. Two cases may occur:
   (i) If there is such a state d that has a transition to a state r by t_{pos+1}. Then the current state becomes r and l is updated to $\text{Length}(d) + 1$.
   (ii) If not, the current state is set to the initial state of $\mathcal{AS}(p)$ with l set to 0.

Lemma 7. *Let q_{pos} be the state of $\mathcal{AS}(p)$ reached after the reading of $t_1 \dots t_{pos}$ by the preceding algorithm, and l_{pos} be the computed length. One of the paths from the initial state to q_{pos} in $\mathcal{AS}(p)$ is labeled by the longest suffix of $t_1 \dots t_{pos}$ that is a factor of p, whose length is l_{pos}.*

Now that we have in each step the length of the longest factor of p, we need a way of recognizing an occurrence of p in the text T.

Corollary 2. *There is an occurrence of $p = p_1 \dots p_m$ in the text in position* pos *if and only if the value of l computed after the reading of t_{pos} is equal to $m = |p|$.*

The validity proof of the FDM is straightforward from lemma 7 and corollary 2.

Lemma 8. *The complexity of the FDM for the search of a word p in the text T is $O(|p| + |T|)$ on the average and in the worst case.*

A second corollary of lemma 7 will be useful in the following.

Corollary 3. *The length l computed by the preceding algorithm in position* pos *in the text T is a majorant of the length of the longest prefix of p that is also a suffix of $t_1 \dots t_{pos}$.*

4 A Simple and Optimal Algorithm

We now present our new string matching algorithm. In a way, it uses two FDM, we call it *Double Forward Dawg Matching* (DFDM).

To explain it in detail, we first present the basic idea it is built on. We move a window of size m (the length of the searched word) along the text, and we make our search in that window. At a given time, the window is located on the text like in the top of figure 4. Assume that we read forward the text in that window, starting near the right end of the window in position B on figure 4. We start reading the text with the suffix automaton (the hatched part of the window, figure 4), but we fail on a letter σ (before we reach the end of the window). We denote u the hatched factor on figure 4. The factor $u\sigma$ is not a factor of the searched word p, because the suffix automaton recognizes them all. Then, we can move the search window safely (*i.e.* without missing any occurrence of word p) up to position $B + 1$.

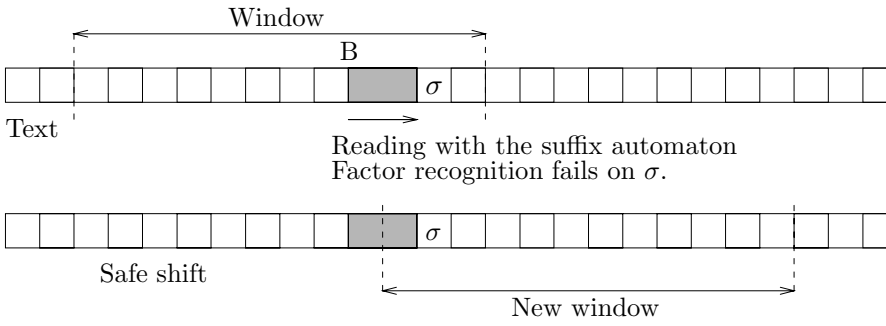


Figure 4. Failure of the recognition of a factor on character σ in the DFDM

If we can repeat this operation on the new window, and so on, we can avoid reading many characters. However, we now have to combine this idea with the FDM and be able to recognize an occurrence of the searched word, while staying linear in the worst case.

We then use a trick combined with our basic approach of figure 4. During the recognition of a factor, we fail on letter σ before reaching the right end of the window. But, this stage corresponds to the beginning of the FDM algorithm. The recognition has failed but we can continue the reading of the text with the FDM (going down the supply links), at least until we reach the right end of the window. We then have to stop this forward search quickly in order to use again the basic idea (figure 4).

More formally, here is our complete algorithm. The current position of our search window is shown figure 5. The word u on the figure is a prefix of the searched window that is also a factor of word p. It is the result of an earlier forward search with a FDM. We know that the length $|u|$ of word u is a bound on the length of the longest prefix of p that can start before position A, and that $|u| < \alpha m$ with $0 < \alpha < 1$, typically $\alpha = 1/2$.

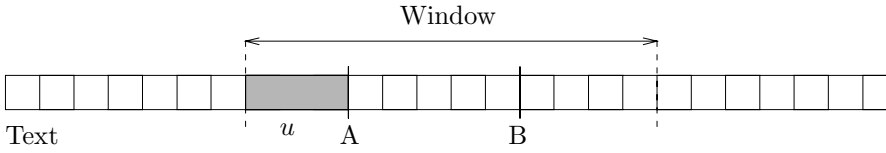


Figure 5. Current position in the DFDM. The word u is such that $|u|$ is a bound on the length of the longest prefix of p

In the current position, we start to read forward the text with a new FDM from position B close (we will precise this later) to the right end of the window. Two cases may occur.

1. We do a supply jump before we have reached the right end of the window. In that case, we simply continue our FDM until we pass the right end of the initial window, and then we stop this FDM once the computed value l (*i.e.*, according to lemma 3, a bound on the length of the longest prefix) is less than αm. Figure 6 illustrates this situation.
2. We reach the right end of the window without doing any supply jump, *i.e.* the part of the window located after B is a factor of p. We resume the reading with the previous FDM that we have stopped in A in order to read again all the characters up to the right end of the window. Once the right end is passed, as in the previous case, we stop this FDM once the computed value l is less than αm. Figure 7 illustrates this situation.

The occurrences are marked during the reading with the FDM.

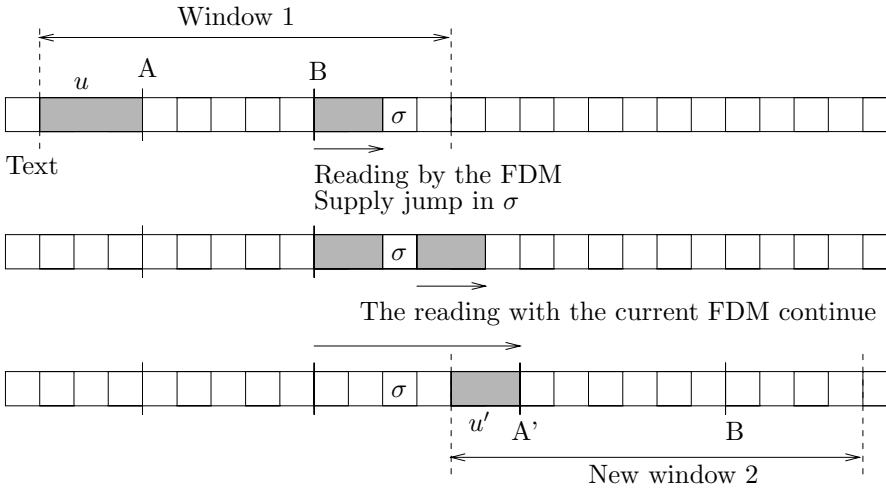


Figure 6. First case in the DFDM. The end of the search window is not reached without supply jump in the FDM started from position B. We continue the reading after the end of window 1 by the FDM until the bound on the longer prefix is smaller than αm.

We always take positions A and B such that $A < B$. Regardless of the value of A and B, we can prove lemma 9 and theorem 1.

Lemma 9. *The DFDM algorithm for searching a word p in a text T marks all the occurrences of p in t and only them.*

Theorem 1. *The complexity in the worst case of the DFDM algorithm to search a word p in a text T is $0(|p| + |T|)$.*

Proof. The pre-processing phase of our algorithm is the construction of the suffix automaton of p, in $O(|p|)$. During the search phase, it is clear that any character in the text can be read at most twice by a FDM. As the complexity of the FDM is $O(|T|)$, the complexity of the search phase is also $O(|T|)$. So the total complexity of the FDM is $O(|p| + |T|)$. □

In order to analyse the average complexity of our algorithm for a word $p = p_1 \dots p_m$ on the text $T = t_1 \dots t_n$, we have to set B and α. We take B such that the distance d_B from B to the end of the search window is:

$$d_B = \max[1, \min(m/5,\ 3\lceil \log_{|\Sigma|}(m) \rceil)\ ],$$

where / is the integer division and $\log_{|\Sigma|}(m)$ the logarithm in base $|\Sigma|$ of m. We now take α such that $d_B < \alpha m < m/2$.

Under these conditions, we can now prove the following results.

Theorem 2. *Under a model of independence and equiprobability of the letters of Σ in the text $T = t_1 \dots t_n$, the average complexity of the DFDM algorithm for the search of a word $p = p_1 \dots p_m$ in T is $O(n \log_{|\Sigma|}(m)/m)$.*

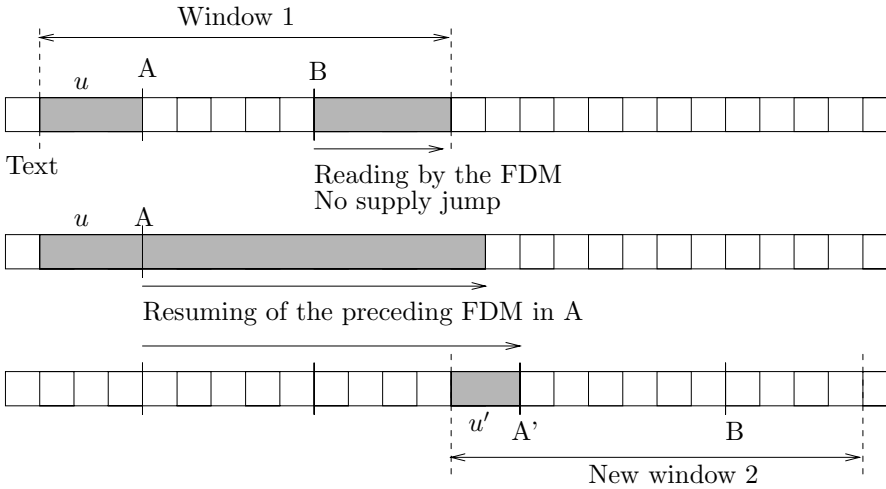


Figure 7. Second case in the DFDM. The end of the search window is reached without supply jump in the FDM started from position B. We resume the reading with the FDM that was previously stopped in A. We continue the reading after the end of window 1 by this FDM until the bound on the longer prefix is smaller than αm.

Proof. We start by bounding the average number of characters that have to be read in order to get a window move of at least $m/2$ characters, leaving and going back to the current situation (of figure 3).

As we are interested in an asymptotic complexity, we consider the case where $d_B = 3\lceil \log_{|\Sigma|}(m) \rceil$.

We consider the two cases of the algorithm:

1. We have a supply jump before the end of the window. In that case, we get a window move of at least $m/2$, and as $d_B < \alpha m$, we are back to the current position. A bound on the number of read character is $d_B = 3\lceil \log_{|\Sigma|}(m) \rceil$.
2. We reach the end of the window without any supply jump and we resume the reading from A. In that case, a bound on the number of characters read to get a window move of at least $m/2$ is $3m/2 + d_B$: d_B characters to read up to the end of the window, then m characters to read from A to the end of the window, then in the worst case $m/2$ characters to be sure of having a window move of at least $m/2$ — $m/2$ is a bound because (a) if the FDM stops before having read these $m/2$ characters, the longest prefix is less than $\alpha m < m/2$ and the window is well more than $m/2$, (b) if the FDM does not stop after $m/2$ characters, the search window which is of length m has been moved of at least $m/2$. We bound $3m/2 + d_B$ by $2m$.
   We now have to consider the number of characters that still have to be read in order to go back to the current position. For a given position i we denote it ν_i. These characters are only read once during the algorithm execution and their total number $\sum_i \nu_i$ is bound by n.

We consider the probabilities of being in each of these cases. The number δ of words δ of length d_B appearing potentially (equiprobably) is $\delta = |\Sigma|^{d_B} = |\Sigma|^{3\lceil \log_{|\Sigma|}(m)\rceil}$ and satisfies $m^3 \leq \delta < |\Sigma| m^3$. The number of factors of p of length d_B is at most m. An upper bound on the probability that a word of length d_B read in the text is a factor of p is $m/m^3 = 1/m^2$. We now go back to our two cases. We underestimate the probability of being in the first case, because it is the one that requires less character readings, and we overestimate the probability of being in the worst case, the second one. A lower bound on the probability of being in the first case then is $(1 - 1/m^2)$. An upper bound on the probability of being in the second case is $1/m^2$.

We get an upper bound on the average number of characters read in both cases equal to

$$\left(1 - \frac{1}{m^2}\right) \times 3\lceil \log_{|\Sigma|}(m)\rceil) + \frac{1}{m^2} \times (2m + \nu_i).$$

We have at most $n/(m/2) = 2n/m$ current position to cover the whole text. Hence an upper bound on the average total number of read characters is

$$\sum_{i=1}^{2n/m} \left[\left(1 - \frac{1}{m^2}\right) \times 3\lceil \log_{|\Sigma|}(m)\rceil) + \frac{1}{m^2} \times (2m + \nu_i)\right],$$

which can be rewritten:

$$\frac{6n}{m} \times \lceil \log_{|\Sigma|}(m)\rceil \left(1 - \frac{1}{m^2}\right) + \frac{4n}{m^2} + \frac{1}{m^2} \sum_{i=1}^{2n/m} \nu_i.$$

As $\sum_{i=1}^{2n/m} \nu_i \leq n$, the total complexity is $O(n \log_{|\Sigma|} m/m)$. □

5 Conclusion

We have presented a new string matching algorithm linear in the worst case and optimal in the average under a model of independence and equiprobability of letters, which uses only a single structure, a suffix automaton.

These last few years, many linear algorithms in constant space have been found [4,9,8,5], but none of them is optimal in the average. The problem of finding an algorithm linear in the worst case, optimal in the average and also in constant space stays open.

References

1. C. Allauzen and M. Raffinot. Simple optimal string matching. Technical Report 99-14, Institut Gaspard-Monge, Université de Marne-la-Vallée, 1999. Accepted for publication in *Journal of Algorithms*.
2. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
3. M. Crochemore. Transducers and repetitions. *Theor. Comput. Sci.*, 45(1):63–86, 1986.
4. M. Crochemore. Constant-space string-matching. In K. V. Nori and S. Kumar, editors, *Proceedings of the 8th Conference on Foundations of Software Technology and Theoretical Computer Science*, number 338 in Lecture Notes in Computer Science, pages 80–87. Springer-Verlag, Berlin, 1988.
5. M. Crochemore, L. Gasieniec, and W. Rytter. Constant-space string matching in sublinear average time. In B. Carpentieri, A. De Santis, U. Vaccaro, and J.A. Storer, editors, *Compression and Complexity of Sequences*, pages 230–239. IEEE Computer Society, 1998.
6. M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.
7. A. Czumaj, M. Crochemore, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12:247–267, 1994.
8. Z. Galil and J. Seiferas. Time-space optimal string matching. *J. Comput. Syst. Sci.*, 26(3):280–294, 1983.
9. L. Gąsieniec, W. Plandowski, and W. Rytter. Constant-space string matching with smaller number of comparisons: sequential sampling. In Z. Galil and E. Ukkonen, editors, *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, number 937 in Lecture Notes in Computer Science, pages 78–89, Espoo, Finland, 1995. Springer-Verlag, Berlin.
10. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.
11. T. Lecroq. *Recherches de mot*. Thèse de doctorat, Université d'Orléans, France, 1992.
12. A. C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.
13. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23:337–343, 1977.

Exact and Efficient Computation of the Expected Number of Missing and Common Words in Random Texts

Sven Rahmann[1] and Eric Rivals[2]

[1] Theoretische Bioinformatik (TBI)
Deutsches Krebsforschungszentrum (DKFZ)
Im Neuenheimer Feld 280, D-69120 Heidelberg, Germany
S.Rahmann@dkfz-heidelberg.de
[2] L.I.R.M.M.
161 rue Ada, F-34392 Montpellier Cedex 5, France
rivals@lirmm.fr

Abstract. The number of missing words (NMW) of length q in a text, and the number of common words (NCW) of two texts are useful text statistics. Knowing the distribution of the NMW in a random text is essential for the construction of so-called monkey tests for pseudorandom number generators. Knowledge of the distribution of the NCW of two independent random texts is useful for the average case analysis of a family of fast pattern matching algorithms, namely those which use a technique called q-gram filtration. Despite these important applications, we are not aware of any exact studies of these text statistics. We propose an efficient method to compute their expected values exactly. The difficulty of the computation lies in the strong dependence of successive words, as they overlap by $(q-1)$ characters. Our method is based on the enumeration of all string autocorrelations of length q, i.e., of the ways a word of length q can overlap itself. For this, we present the first efficient algorithm. Furthermore, by assuming the words are independent, we obtain very simple approximation formulas, which are shown to be surprisingly good when compared to the exact values.

1 Introduction

We consider *random texts*. A *text* of length n is a string of n characters from a given *alphabet* Σ of size σ. *Randomness* in this article refers to the symmetric Bernoulli model, meaning that the probability to encounter any character at any position of the text is $1/\sigma$, independently of the other text positions. As soon as the text length has been fixed to n, every text has the same chance of occurring, namely σ^{-n}. A *word* of length q, also called a *q-gram*, is a substring of length q. If $n \geq q$, we find exactly $(n - q + 1)$ (overlapping) q-grams in a text of length n; they end at positions q, $q + 1$, ..., n. However, not all of these need to be different.

We consider two applications that motivate why it is interesting to determine the distribution, and hence especially the expectation, of the number of missing

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 375–387, 2000.

words (NMW) in a random text, and of the number of common words (NCW) of two independent random texts.

Missing Words and Monkey Tests. Knowing the distribution of the NMW in a random text allows the construction of so-called *monkey tests* for pseudo-random number generators (PRNGs).

Assume we are given a subroutine that supposedly produces instances of a random variable U uniformly distributed in the interval $[0, 1]$, but we are unsure of the quality of this PRNG, i.e., whether the produced sequences of pseudorandom numbers indeed share properties with truly random sequences. In this case one usually runs a series of empirical tests, a list of which can be found in Knuth's comprehensive work [9].

One set of tests called *monkey tests* was proposed by Marsaglia and Zaman [10]. One variant works as follows: Each call to the PRNG is used to create a pseudorandom bit (e.g., the most or least significant bit of U), and a text of length n is created by concatenating the bits from successive calls to the PRNG. (Imagine a monkey typing randomly on a 0-1-keyboard; hence the name monkey test.) One counts how many of the 2^q possible q-grams are missing from the resulting text, and compares this number to the expected NMW in a truly random bit sequence. The PRNG should be rejected if these numbers differ significantly.

The advantage of monkey tests is that, for each q-gram, only one bit needs to be stored to indicate whether the q-gram has occurred in the text or not. Other tests, like the well-known chi-square test on the frequencies of q-grams, require the storage of an integer (the number of occurrences) for every q-gram. Hence monkey tests allow to choose q relatively large, such as $q = 33$, needing 2^{33} bits or 1 GB of memory. In comparison, if one runs the chi-square test with 1 GB of memory, one is restricted to $q \leq 28$ (assuming 4 bytes per integer). Hence the monkey test allows to detect possible deficiencies of the PRNG within a larger context and can capture dependencies within the generated sequence that other tests may miss.

To construct a precise statistical test, we need to know the distribution of the NMW. Here, we present an efficient method to compute the exact expectation. In [10] the authors express a conjecture about the distribution from simulations. In Section 4, we propose a slightly different central limit conjecture, which agrees in principle with the observations in [10], but additionally has theoretical foundations in the classical occupancy problem for urn models (e.g., see [7]).

Common Words and Analysis of Pattern Matching Algorithms Using q-Gram Filtration. The NCW statistic has many applications. It serves as a distance measure between texts, especially for the comparison of biological sequences [11, 17, 13, 6]. It is also important for the analysis of text algorithms and especially for pattern matching algorithms. Here we consider the analysis of filtration algorithms for the *k-difference approximate pattern matching problem* defined as follows: Given a pattern P and a text T, find all ending positions i

in T of an approximate match of P with at most k differences (substitutions, insertions or deletions). Filtration algorithms use a quickly verifiable necessary condition for a match to filter out parts of T where no match can occur; the remaining parts are checked with a slower dynamic programming algorithm. Several filtration strategies are based on the condition that an approximate match and P should share a sufficient number of q-grams (among others, see [8, 18, 19]). Algorithms based on q-gram filtration perform very well in practice when the filtration condition is not fulfilled very often. The average running time depends on the NCW. The ability to compute its expectation should allow to analyze the average time complexity and to determine under which conditions which algorithm performs best. We were motivated to examine word statistics in random texts since we hope to analyze and fine-tune the recently proposed QUASAR algorithm [2], which uses the q-gram filtration approach. It serves to search for similar biological sequences in large databases, and has has been shown to be an order of magnitude faster than standard programs like BLAST [1].

Organization of this Article. As of today, the literature does not report any exact systematic statistical study of the NMW and NCW in random texts (but see [10, 12] for some results). We describe an efficient method to calculate the exact expectations in Section 2. They depend on the probability that a q-gram does not occur in a text of length n, which itself depends only on the *autocorrelation* of the q-gram. The autocorrelation is a binary vector giving the shifts that cause the word to overlap itself. Our method requires the computation of all possible autocorrelations of length q, for which we propose the first efficient algorithm (Section 3).

The difficulty in computing the exact expectations arises from the fact that successive q-grams in a text overlap and are hence dependent. Treating them as if they were independent, one obtains the so-called classical occupancy problem for urn models (see [7] or Section 4). In this much simpler setup, many results are known, and we propose to use them as approximations to the missing words problem. The quality of these approximations is evaluated in Section 5, where we also give some exemplary results on the two applications mentioned above.

2 Computation of Expectations

2.1 Expected Number of Missing q-Grams

We assume that the word length $q \geq 2$ and the alphabet size $\sigma \geq 2$ are fixed. We denote by $X^{(n)}$ the random number of missing q-grams in a text $T^{(n)}$ of length n. Assume we have enumerated the σ^q possible q-grams in some arbitrary order; let us call them $Q_1, Q_2, \ldots, Q_{\sigma^q}$. For the event that a q-gram Q occurs in $T^{(n)}$ we use the shorthand notation $Q \in T^{(n)}$, and $Q \notin T^{(n)}$ for the complementary event. By the method of indicator variables we find that

$$\mathbb{E}[X^{(n)}] = \sum_{i=1}^{\sigma^q} \Pr(Q_i \notin T^{(n)}). \tag{1}$$

An important point is that even in the symmetric Bernoulli model the probabilities of non-occurrence are not the same for every q-gram. For example, among the eight bit-strings of length 3, five do not contain the 2-gram '00', while four do not contain the 2-gram '01'. The probability depends on the autocorrelation of the q-gram, defined as follows (see also [4, 16]).

Definition 1 (Autocorrelation of a q-Gram). Let $Q = Q[0]\dots Q[q-1]$ be a q-gram over some alphabet. Then we define its *autocorrelation* $c(Q) \equiv c := (c_0, \dots, c_{q-1})$ as follows: For $i = 0, \dots, q-1$, set $c_i := 1$ iff the pattern overlaps itself if slided i positions to the right, i.e., iff $Q[i+j] = Q[j]$ for all $j = 0, \dots, (q-i-1)$. Otherwise, set $c_i := 0$. Note that by this definition we always have $c_0 = 1$.

The corresponding *autocorrelation polynomial* $(C(Q))(z) \equiv C(z)$ is obtained by taking the bits as coefficients: $C(z) := c_0 + c_1 z + c_2 z^2 + \dots + c_{q-1} z^{q-1}$.

As an example, consider the 11-gram Q=`ABRACADABRA`. By looking at the positions where this word can overlap itself, we find that $c(Q) = (10000001001)$, and therefore $(C(Q))(z) = 1 + z^7 + z^{10}$.

The following lemma gives the probability of the event $Q \notin T^{(n)}$.

Lemma 1 (Guibas and Odlyzko [5]). Let Q be a q-gram over some alphabet of size σ; let $C(z)$ be its autocorrelation polynomial. Then the generating function $P(z)$ of the sequence $(p_n := \Pr(Q \notin T^{(n)}))_{n \geq 1}$ is given by

$$P(z) = \frac{C\left(\frac{z}{\sigma}\right)}{\left(\frac{z}{\sigma}\right)^q + (1-z) \cdot C\left(\frac{z}{\sigma}\right)}.$$

The lemma states that, if we expand $P(z)$ as a power series $P(z) = \sum_{n=0}^{\infty} p_n z^n$, then the coefficient p_n of z^n is exactly $\Pr(Q \notin T^{(n)})$. We will use the usual "coefficient extraction" notation $[z^n]P(z)$ to refer to p_n.

Efficient Computation. In the symmetric Bernoulli model, the probability of non-occurrence of a q-gram depends only on the alphabet size and on the autocorrelation, but not on the q-gram itself. Thus, we can simplify the calculation of $\mathbb{E}[X^{(n)}]$ by grouping together q-grams with the same autocorrelation. Let us enumerate the distinct autocorrelations that occur among the q-grams in some arbitrary order: $C_1(z)$, $C_2(z)$, ..., $C_\kappa(z)$, with κ being their number. Let $P_j(z)$ be the generating function of Lemma 1 with $C(z)$ replaced by $C_j(z)$, and let N_j be the number of q-grams with autocorrelation polynomial $C_j(z)$. We refer to N_j as the *population size* of the autocorrelation $C_j(z)$. With these definitions and Lemma 1, Equation (1) becomes

$$\mathbb{E}[X^{(n)}] = \sum_{j=1}^{\kappa} N_j \cdot [z^n] P_j(z). \tag{2}$$

To evaluate this sum, we must know the κ different polynomials $C_j(z)$ and their population sizes N_j. This is discussed in Section 3. First, we mention how to evaluate $[z^n]P_j(z)$ efficiently.

2.2 Coefficient Extraction

Note that the $P_j(z)$ $(j = 1, \ldots, \kappa)$ are rational functions of the form $\frac{f(z)}{g(z)}$, where f and g are polynomials in z such that $\deg(f) < \deg(g)$ (here $\deg(f) = q - 1$ and $\deg(g) = q$). Moreover, f and g have no common factors, and $g(0) = 1$. For the remainder of this section, call a function of this form a *standard* rational function.

There is an exact method to compute the coefficients of the power series expansion of standard rational functions in time $O(q^3 \log n)$, e.g., see [9].

However, in our case the generating functions P_j are particularly well behaved when it comes to asymptotic approximation based on the following lemma.

Lemma 2 (Asymptotic Approximation). Let $P(z) = \frac{f(z)}{g(z)}$ be a standard rational function, and let $b_1, \ldots, b_q$ be the (not necessarily distinct) poles of $P(z)$, i.e., the complex values for which $g(z) = 0$, appearing according to their multiplicity and ordered such that $|b_1| \leq |b_2| \leq \ldots \leq |b_q|$. If $|b_1| < |b_2|$, then $\beta := b_1 \in \mathbb{R}$ and

$$p_n \quad \sim \quad -\frac{f(\beta)}{g'(\beta)} \cdot \beta^{-(n+1)},$$

where $\sim$ means that the ratio of the left and right hand sides tends to 1 as $n \to \infty$. The error drops roughly as $\left|\frac{b_1}{b_2}\right|^n$.

Proof. See [16], or any textbook about linear difference equations. □

To apply the lemma, we need to verify that the $P_j(z)$ fulfill the $|b_1| < |b_2|$ condition. The rather technical proof is not included here; a proof along similar lines can be found in [3]. It turns out that the pole of smallest absolute value of $P_j(z)$ is always slightly larger than 1, whereas the absolute values of the other poles are much larger. Hence $|b_1|/|b_2|$ is small, and the asymptotic approximation is already excellent for small values of n, such as $n = 2q$. A good way to determine $\beta = b_1$ is Newton's method with a starting value of 1. We summarize the main result:

Theorem 1 (Expected Number of Missing Words). Except for a negligible asymptotic approximation error,

$$\mathbb{E}[X^{(n)}] = \sum_{j=1}^{\kappa} -N_j \cdot \frac{f_j(\beta_j)}{g_j'(\beta_j)} \cdot \beta_j^{-(n+1)},$$

where $f_j(z) := C_j(z/\sigma)$ and $g_j(z) := (z/\sigma)^q + (1 - z) \cdot C_j(z/\sigma)$, and β_j is the unique root of smallest absolute value of $g_j(z)$. □

2.3 Expected Number of Common q-Grams of Two Texts

The same principles can be used to compute the expected number of q-grams that are common to two independent random texts $T^{(n)}$ and $S^{(m)}$ of lengths n and m, respectively.

We need to define more precisely how to count common q-grams. If a given fixed q-gram Q appears $x > 0$ times in $T^{(n)}$ and $y > 0$ times in $S^{(m)}$, should that count as (a) only 1, (b) x, (c) y, or (d) xy common q-gram(s)? The answer depends on the counting algorithm. Case (a) suggests that we check for each of the σ^q q-grams whether it occurs in both texts and increment a counter if it does. Case (b) implies that we look at each q-gram in $T^{(n)}$ successively, and increment a counter by one if we find the current q-gram also in $S^{(m)}$. Case (c) is symmetric to (b), and case (d) suggests that for each q-gram of $T^{(n)}$, a counter is incremented by the number of times the q-gram appears in $S^{(m)}$.

Let $Z_\bullet^{(n,m)}$ denote the number of q-grams that occur simultaneously in both $T^{(n)}$ and $S^{(m)}$ according to case ($\bullet$) with $\bullet \in \{\mathrm{a,b,c,d}\}$.

Theorem 2 (Expected Number of Common Words). Except for a negligible asymptotic approximation error,

$$\mathbb{E}[Z_a^{(n,m)}] = \sum_{j=1}^{\kappa} N_j \cdot \left(1 + \frac{f_j(\beta_j)}{g_j'(\beta_j)}\beta_j^{-(n+1)}\right) \cdot \left(1 + \frac{f_j(\beta_j)}{g_j'(\beta_j)}\beta_j^{-(m+1)}\right)$$

$$\mathbb{E}[Z_b^{(n,m)}] = \frac{n-q+1}{\sigma^q}\left(\sigma^q + \sum_{j=1}^{\kappa} N_j \frac{f_j(\beta_j)}{g_j'(\beta_j)}\beta_j^{-(m+1)}\right)$$

$$\mathbb{E}[Z_d^{(n,m)}] = \frac{(n-q+1)\cdot(m-q+1)}{\sigma^q} \quad \text{(This is exact.)}$$

The result for $\mathbb{E}[Z_c^{(n,m)}]$ is obtained by exchanging n and m in the result for $\mathbb{E}[Z_b^{(n,m)}]$. We have used κ, N_j, $f_j(z)$, $g_j(z)$ and β_j as in Theorem 1.

Proof. As before, let $Q_1, \ldots, Q_{\sigma^q}$ be an arbitrary enumeration of all q-grams over Σ. For case (a), we have $\mathbb{E}[Z_a^{(n,m)}] = \sum_{i=1}^{\sigma^q} \Pr(Q_i \in T^{(n)} \text{ and } Q_i \in S^{(m)})$. By using the independence of the texts, noting $\Pr(Q_i \in T^{(n)}) = 1 - \Pr(Q_i \notin T^{(n)})$, and grouping q-grams with the same autocorrelation together, we obtain

$$\mathbb{E}[Z_a^{(n,m)}] = \sum_{j=1}^{\kappa} N_j \cdot (1 - [z^n]P_j(z)) \cdot (1 - [z^m]P_j(z)).$$

An application of Lemma 2 proves the theorem for case (a).

For case (b), let $Y^{(m)}$ be the number of missing words in $S^{(m)}$. Then the expected number of different q-grams appearing in $S^{(m)}$ is $\sigma^q - \mathbb{E}[Y^{(m)}]$. Each of these is expected to appear $(n-q+1)/\sigma^q$ times in $T^{(n)}$. Since the texts are independent, we obtain $\mathbb{E}[Z_b^{(n,m)}] = \frac{n-q+1}{\sigma^q}(\sigma^q - \mathbb{E}[Y^{(m)}])$, and an application of Theorem 1 proves case (b).

Case (c) is symmetric to case (b), and case (d) is left to the reader. □

3 Enumerating Autocorrelations and Population Sizes

Enumerating the Autocorrelations. Our enumeration algorithm is based on a recursively defined test procedure $\Xi(v)$ proposed by Guibas and Odlyzko

in [4]. It takes as input a binary vector v of arbitrary length q, and outputs 'true' if v arises as the autocorrelation of some q-gram over any alphabet[1], and 'false' otherwise.

To enumerate $\Gamma(q)$, the set of all autocorrelations of length q, one could in principle use Ξ to test every binary vector $v = (v_0, \ldots, v_{q-1})$ with $v_0 = 1$ whether it is an autocorrelation, but this requires an exponential number of tests. The recursive structure of $\Xi(v)$ allows to build an efficient dynamic programming algorithm for the enumeration of $\Gamma(q)$, which we describe now.

First, a notational remark. While we use n for the text length in the other sections, here we use n as a running variable for the autocorrelation length, $n = 1, \ldots, q$. This should cause no confusion.

For a binary vector v of length n with $v_0 = 1$, define $\pi(v) := \min\{1 \leq i < n : v_i = 1\}$, and let $\pi(v) := n$ if no such i exists. For $n \geq 1$, $p = 1, \ldots, n$, let $\Gamma(n,p)$ be the set of autocorrelations v of length n for which $\pi(v) = p$. Then $\Gamma(n) = \bigcup_{p=1}^{n} \Gamma(n,p)$. We denote the concatenation of two bit vectors s and t by $s \circ t$, and the k-fold concatenation of s with itself by s^k. So $\mathtt{1}\mathtt{0}^k \circ w$ is the binary vector starting with $\mathtt{1}$, followed by k $\mathtt{0}$s, and ending with the binary vector w.

From the definition of autocorrelation we have that $v \in \Gamma(n,p)$ implies $v_{jp} = 1$ for all $j = 1, 2, \ldots$, for which $jp < n$. It follows that $\Gamma(n,1) = \{\mathtt{1}^n\}$. Also, $\Gamma(n,n) = \{\mathtt{1}\mathtt{0}^{n-1}\}$. To obtain $\Gamma(n,p)$ for $n \geq 3$ and $2 \leq p \leq (n-1)$, we assume all $\Gamma(m,p')$ with $m < n$, $1 \leq p' \leq m$, are already known. Then there are two cases:

Case (a) $[\mathbf{2 \leq p \leq \frac{n}{2}}]$: Let $r' := n \bmod p$ and $r := r' + p$. Then $p \leq r < 2p$, and there are two sub-cases. In each of them, $\Gamma(n,p)$ can be constructed from a subset of $\Gamma(r)$. Hence, let $s_{n,p} := (\mathtt{1}\mathtt{0}^{p-1})^{\lfloor n/p \rfloor - 1}$; every string in $\Gamma(n,p)$ has $s_{n,p}$ as a prefix, and is then followed by a string $w \in \Gamma(r)$, as follows.

1. Case $r = p$:

$$\Gamma(n,p) = \{s_{n,p} \circ w \mid w \in \Gamma(r,p');\ r' + \gcd(p,p') < p' < p\} \tag{3}$$

2. Case $p < r < 2p$:

$$\begin{aligned} \Gamma(n,p) = {} & \{s_{n,p} \circ w \mid w \in \Gamma(r,p)\} \\ & \dot{\bigcup} \{s_{n,p} \circ w \mid w \in \Gamma(r,p');\ r' + \gcd(p,p') < p' < p;\ \ w_p = 1\} \end{aligned} \tag{4}$$

We remark that the condition imposed on p' in (3) and (4) ($r' + \gcd(p,p') < p' < p$) implies that p' must not divide p.

Case (b) $[\mathbf{\frac{n}{2} < p \leq (n-1)}]$: $\Gamma(n,p)$ is constructed from $\Gamma(n-p)$.

$$\Gamma(n,p) = \{\mathtt{1}\mathtt{0}^{p-1} \circ w \mid w \in \Gamma(n-p)\} \tag{5}$$

[1] In [4], it is proved that if v is an autocorrelation of some word over an alphabet of size $\sigma \geq 2$, then it is also the correlation of word over a two-letter alphabet.

The equations (3), (4) and (5), along with the known forms of $\Gamma(n,1)$ and $\Gamma(n,n)$ for all n, yield a dynamic programming algorithm for the construction of $\Gamma(q)$. For $n = 3, \ldots, q$, in order to compute all $\Gamma(n,p)$ with $1 \le p \le n$, one needs the sets of autocorrelations of shorter lengths $\Gamma(m,p')$ with $m < \lfloor \frac{2n}{3} \rfloor$, $1 \le p' \le m$. The correctness of the algorithm follows from the correctness of the recursive predicate $\Xi(v)$ in [4], after which it is modeled.

One improvement is possible: In case (a), $\Gamma(n,p)$ is obtained from autocorrelations $w \in \Gamma(r)$ with $r \ge p$. Examining the characterization of autocorrelations given in [4] more closely, it can be shown that such w must have $\pi(w) > (n \bmod p)$, and therefore it is possible to construct $\Gamma(n,p)$ from the sets $\Gamma(s)$ with $s < p$. Hence, to obtain $\Gamma(n,p)$, in both cases (a) and (b), only the sets $\Gamma(m,p')$ with $m \le \lfloor \frac{n}{2} \rfloor$, $1 \le p' \le m$ are needed. For example, to compute $\Gamma(200)$, we only need to know $\Gamma(1), \ldots, \Gamma(100)$, but not $\Gamma(101), \ldots, \Gamma(199)$.

The Number of Autocorrelations. When we know $\Gamma(q)$, we also know its cardinality $\kappa(q)$, which is the number of terms in the sum of (2). In [4] it is proved that asymptotically

$$\kappa(q) \le \exp\left(\frac{(\ln q)^2}{2\ln(3/2)} + o((\ln q)^2)\right);$$

hence computing the expected number of missing words by (2) is considerably more efficient than by (1), which uses σ^q terms.

Population Sizes. To determine how many q-grams over an alphabet of size σ share a given autocorrelation $v \in \Gamma(q)$ (the population size $N(v)$), we refer the reader to [4, Section 7], where a recurrence for $N(v) = N((v_0, \ldots, v_{q-1}))$ in terms of $N((v_{\pi(v)}, \ldots, v_{q-1}))$ is given, or to our technical report [15], where an alternative method is described.

4 Approximations

Since for large q, the exact methods in Section 2 become very time-consuming, we consider a simpler but related problem, whose solution we take as an approximate solution to the missing words problem.

Classical Occupancy Problem: When N balls are independently thrown into M equiprobable urns, what is the distribution of the number of empty urns X after the experiment? For this setup, the moments of X (expectation, variance, and higher moments) are known. For example, we have that

$$\mathbb{E}[X] = M\left(1 - \frac{1}{M}\right)^N, \tag{6}$$

$$\mathrm{Var}[X] = M\left(1 - \frac{1}{M}\right)^N + M(M-1)\left(1 - \frac{2}{M}\right)^N - M^2\left(1 - \frac{1}{M}\right)^{2N} \tag{7}$$

Even the distribution of X can be given explicitly in terms of the so-called Stirling numbers of the second kind. From this knowledge, the following result can be derived.

Lemma 3 (Central Limit Theorem for the Classical Occupancy Problem). Let (N_k) and (M_k) be sequences of natural numbers such that $N_k \to \infty$, $M_k \to \infty$ and $\frac{N_k}{M_k} \to \lambda > 0$, as $k \to \infty$. Let (X_k) be the sequence of random variables denoting the number of empty urns after N_k balls have been thrown independently into M_k urns. Then, as $k \to \infty$, we have

$$\mathbb{E}[X_k/M_k] \to e^{-\lambda}, \tag{8}$$

$$\mathrm{Var}[X_k/\sqrt{M_k}] \to (e^{\lambda} - 1 - \lambda)e^{-2\lambda}. \tag{9}$$

Moreover, we have convergence in distribution

$$\frac{X_k - M_k\, e^{-\lambda}}{\sqrt{M_k\,(e^{\lambda} - 1 - \lambda)\, e^{-2\lambda}}} \xrightarrow{\mathcal{D}} \mathcal{N}, \tag{10}$$

where $\mathcal{N}$ denotes the standard normal distribution.

Proof. See, for example, [7]. □

The missing words problem is a modification of the classical occupancy problem in the following sense. The M urns correspond to the σ^q possible words, and the N balls correspond to the $(n-q+1)$ q-grams of the text. The difference is that successive q-grams in the text are strongly dependent because they overlap by $(q-1)$ characters, while the balls in the occupancy problem are assumed to be independent.

Approximations. We propose to use the results from the occupancy problem by assuming that the q-grams are not taken from a text but generated independently. Then the probability that a fixed q-gram does not occur among $n-q+1$ randomly drawn q-grams is $(1-\frac{1}{\sigma^q})^{n-q+1}$. Hence, the expected number of missing words can be approximated by (6), giving

$$\mathbb{E}[X^{(n)}] \approx \sigma^q \cdot \left(1 - \frac{1}{\sigma^q}\right)^{n-q+1} \tag{11}$$

This can be further approximated by using the asymptotic value from (8), resulting in

$$\mathbb{E}[X^{(n)}] \approx \sigma^q \cdot e^{-\lambda}, \qquad \lambda := \frac{n-q+1}{\sigma^q} \tag{12}$$

We call (11) the *independence approximation* (IA), and (12) the *exponential approximation* (EA).

A Central Limit Conjecture for Missing Words. The numerical results in Section 5 show that the independence approximation is surprisingly good. Also, the asymptotic variance in the occupancy problem ($M_k\,(e^{\lambda} - 1 - \lambda)\,e^{-2\lambda}$) is in accordance with the variance of the number of missing words observed by Marsaglia & Zaman in [10] ($M_k\,(e^{\lambda} - 3)\,e^{-2\lambda}$) when they conducted simulations for slightly varying $\lambda \approx 2$. Although we have not checked higher moments, we formulate the following conjecture.

Conjecture 1. The number of missing q-grams over an alphabet of size σ in a text of length n, and the number of empty urns after $N := n - q + 1$ balls have been independently thrown into $M := \sigma^q$ urns have the same Gaussian limit distribution, as given in Lemma 3.

Further Remarks. Other approximations are possible. For example, one may assume that the waiting time until a q-gram first occurs in the text has a geometric distribution. Due to space constraints, we do not consider this approximation any further here. In our tests, (IA) always turned out to be better by an order of magnitude. Also, (IA) and (EA) can be applied to the common words problem as well, as shown here for case (b) from Section 2.3:

(IA): $\mathbb{E}[Z_b^{(n,m)}] \approx (n-q+1) \cdot \left(1 - \left(1 - \frac{1}{\sigma^q}\right)^{m-q+1}\right)$
(EA): $\mathbb{E}[Z_b^{(n,m)}] \approx (n-q+1) \cdot \left(1 - \exp\left(\frac{m-q+1}{\sigma^q}\right)\right)$

5 Comparison of the Methods

We evaluate the quality of the approximations in two different scenarios of practical relevance.

A Monkey Test of High Order. Coming back to the monkey test described in Section 1, we compute the expected number of missing bit-strings of length 33 in a random text of varying length n exactly by Theorem 1 (XT), and approximately according to the independence approximation and the exponential approximation from Section 4. The excellent quality of the approximations can be seen in Table 1. The relative errors are around 10^{-8} or lower. Since for the exact method, $\kappa = \kappa(33) = 538$ different correlations have to be evaluated, the approximations save time without sacrificing quality. This behavior is quite typical when the alphabet size, the word length and the text length are sufficiently large. For some further numerical results, see [14]. For all computations, we used 100-digit-precision arithmetic. High precision is necessary for the exact computation and for (IA), as the standard floating point introduces large roundoff errors. (EA) is more robust in this respect.

QUASAR Analysis. The QUASAR algorithm [2] can be used to search DNA databases (long texts with $\sigma = 4$) for approximate matches to a pattern of

Table 1. Expected number of missing 33-grams for alphabet size $\sigma = 2$ and varying text length n. Line 1 (XT) shows the exact values according to Theorem 1. Line 2 (IA) gives the independence approximation (IA), and Line 3 the order of magnitude of its relative error. Line 4 shows the exponential approximation (EA), and Line 5 the order of magnitude of its relative error.

Expected Fraction of Missing q-grams, $\mathbb{E}[X^{(n)}]/\sigma^q$, for $\sigma = 2$ and q=33					
	Method	$n = 0.5\,\sigma^q$	$n = \sigma^q$	$n = 2\,\sigma^q$	$n = 4\,\sigma^q$
1	(XT)	0.606530661913	0.36787944248	0.135335283715	0.018315638966
2	(IA)	0.606530661954	0.36787944252	0.135335283725	0.018315638952
3	$\log_{10}$(Rel.Error)	-10.169	-9.972	-10.147	-9.124
4	(EA)	0.606530661972	0.36787944254	0.135335283741	0.018315638957
5	$\log_{10}$(Rel.Error)	-10.014	-9.783	-9.726	-9.285

Table 2. Expected number of common q-grams of two texts of length 50 and 256 according to case (b) of Section 2.3, for varying q. The alphabet size is $\sigma = 4$. Line 1 shows the exact value (XT). Line 2 gives the independence approximation (IA), and Line 3 shows the magnitude of its relative error. Line 4 gives the exponential approximation (EA), and Line 5 the order of its relative error. Line 6 shows the Jokinen-Ukkonen threshold $t(q)$ for q-gram filtration to find matches with $k = 7$ differences. Line 7 gives an upper bound on the probability of at least $t(q)$ common q-grams, based on Markov's inequality.

	Quantity	$q = 3$	$q = 4$	$q = 5$	$q = 6$
1	$\mathbb{E}[Z]$ (XT)	47.10188	29.55391	10.04229	2.67534
2	$\mathbb{E}[Z]$ (IA)	47.12092	29.53968	10.03927	2.67509
3	$\log_{10}$(Rel.Error(IA))	-3.393	-3.317	-3.523	-4.041
4	$\mathbb{E}[Z]$ (EA)	47.09294	29.50585	10.03495	2.67478
5	$\log_{10}$(Rel.Error(EA))	-3.722	-2.789	-3.136	-3.679
6	$t(q)$	27	19	11	3
7	$\Pr(Z \geq t(q))$ (Markov)	trivial ≤ 1	trivial ≤ 1	≤ 0.913	≤ 0.892

length $n = 50$ with at most $k = 7$ differences. Assume that the database has been partitioned into blocks of size $m = 256$. Assume further that an index is available to quickly determine how many q-grams $Z \equiv Z_b^{(n,m)}$ each block and the pattern have in common according to Section 2.3, Case (b), for $q \in \{3, 4, 5, 6\}$. By a lemma of Jokinen & Ukkonen [8], no k-approximate match can occur in a block if the number of common q-grams is below $t(q) := n - q + 1 - kq$. Hence in the event $Z \geq t(q)$ a block must be kept for further inspection. It is natural to ask for which value of q the filtration performance is optimal, i.e., $\Pr(Z \geq t(q))$ is minimal. When only $\mathbb{E}[Z]$ is known, $\Pr(Z \geq t(q))$ can be bounded with Markov's inequality, $P(|Z| \geq t) \leq \mathbb{E}[|Z|]/t$. Table 2 lists some relevant values. The approximation error is of the order 10^{-3}, i.e., approximately 0.1%.

If one is willing to accept the simple random input model[2], $q = 6$ offers the best bound, but a more precise statement can only be made if more is known about the distribution of Z.

6 Conclusion and Future Work

We have presented exact and approximate methods to compute the expected number of missing q-grams in a random text and the expected number of common words in two independent random texts. The exact computations require the knowledge of all autocorrelations of length q, for which we exhibit an efficient dynamic programming algorithm.

We observe that in general the independence approximation (IA) gives excellent results, although clearly the q-grams of a text are not independent. An explanation is that, although for each i, $\Pr(Q_i \notin T^{(n)})$ may be very different from $(1 - \frac{1}{\sigma^q})^n$, there is an averaging effect in the sum over all q-grams, such that (IA) continues to hold approximately. We have been unable to prove this, though. It also remains an open problem to prove or disprove Conjecture 1, or to make it more precise. We wanted to point out the relatedness of the NCW problem and the classical occupancy problem, which seems to have been overlooked in related work.

For short texts, small alphabets and small word lengths, the errors due to the use of either approximation are quite high. The exact method is then advised, since it is inexpensive in these cases. However, high precision arithmetic should be used to avoid roundoff errors. In the other cases, (EA) is the most reasonable choice, because its evaluation poses the least numerical problems, and the approximation error can be neglected.

Acknowledgments: We thank M. Vingron, P. Nicodème, and E. Coward for helpful discussions. We wish to thank especially the groups of Ph. Flajolet and S. Schbath for the opportunity to discuss this work at an early stage. The referees' comments have led to substantial improvements in the presentation of this work. E. R. was supported by a grant of the Deutsches Humangenomprojekt and is now supported by the CNRS.

References

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic Local Alignment Search Tool (BLAST). *Journal of Molecular Biology*, 215:403–410, 1990.

[2] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. q-gram based Database Searching Using a Suffix Array (QUASAR). In S. Istrail, P. Pevzner, and M. Waterman, editors, *Proceedings of The Third International Conference on Computational Molecular Biology*, pages 77–83. ACM-Press, 1999.

[2] DNA is surely not a "random text" since it contains biological information. Still, the analysis of the algorithm under a simple random input model can give valuable hints to its performance on real data, as long as no stronger formal assumptions about the data can be made.

[3] L. J. Guibas and A. M. Odlyzko. Maximal Prefix-Synchronized Codes. *SIAM Journal of Applied Mathematics*, 35(2):401–418, 1981.
[4] L. J. Guibas and A. M. Odlyzko. Periods in Strings. *Journal of Combinatorial Theory, Series A*, 30:19–42, 1981.
[5] L. J. Guibas and A. M. Odlyzko. String Overlaps, Pattern Matching, and Nontransitive Games. *Journal of Combinatorial Theory, Series A*, 30:183–208, 1981.
[6] W. Hide, J. Burke, and D. Davison. Biological evaluation of d2, an algorithm for high-performance sequence comparison. *J. Comp. Biol.*, 1:199–215, 1994.
[7] N. L. Johnson and S. Kotz. *Urn Models and Their Applications*. Wiley, New York, 1977.
[8] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In A. Tarlecki, editor, *Proceedings of the 16th symposium on Mathematical Foundations of Computer Science*, number 520 in Lecture Notes in Computer Science, pages 240–248, Berlin, 1991. Springer-Verlag.
[9] D. E. Knuth. *The Art of Computer Programming*, volume 2 / Seminumerical Algorithms. Addison-Wesley, Reading, MA, third edition, 1998.
[10] G. Marsaglia and A. Zaman. Monkey Tests for Random Number Generators. *Computers and Mathematics with Applications*, 26(9):1–10, 1993.
[11] A. A. Mironov and N. N. Alexandrov. Statistical method for rapid homology search. *Nucleic Acids Res*, 16(11):5169–73, Jun 1988.
[12] O. E. Percus and P. A. Whitlock. Theory and Application of Marsaglia's Monkey Test for Pseudorandom Number Generators. *ACM Transactions on Modeling and Computer Simulation*, 5(2):87–100, April 1995.
[13] P. A. Pevzner. Statistical distance between texts and filtration methods in sequence comparison. *Comp. Appl. BioSci.*, 8(2):121–127, 1992.
[14] S. Rahmann and E. Rivals. The Expected Number of Missing Words in a Random Text. Technical Report 99-229, LIRMM, Montpellier, France, 1999.
[15] E. Rivals and S. Rahmann. Enumerating String Autocorrelations and Computing their Population Sizes. Technical Report 99-297, LIRMM, Montpellier, France, 1999.
[16] R. Sedgewick and P. Flajolet. *Analysis of Algorithms*. Addison-Wesley, Reading, MA, 1996.
[17] D. C. Torney, C. Burks, D. Davison, and K. M. Sirotkin. Computation of d^2: A measure of sequence dissimilarity. In G. Bell and R. Marr, editors, *Computers and DNA*, pages 109–125, New York, 1990. Sante Fe Institute studies in the sciences of complexity, vol. VII, Addison–Wesley.
[18] E. Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science*, 92(1):191–211, Jan. 1992.
[19] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the Association for Computing Machinery*, 35(10):83–91, Oct. 1992.

Periods and Quasiperiods Characterization

Mireille Régnier*[1] and Laurent Mouchard**[2,3]

[1] INRIA Rocquencourt, Domaine de Voluceau
B.P. 105, 78153 Le Chesnay Cedex, France
[2] ESA 6037 - ABISS, Université de Rouen, 76821 Mont-Saint-Aignan Cedex, France
[3] Dept. of Comp. Sci., King's College London, Strand WC2R 2LS, London, UK

Abstract. We provide a new characterization of periods and quasiperiods that is constructive. It allows for a canonical partition of the set of borders of a given word w. Each subset of the partition contains a superprimitive border q and possibly quasiperiodic borders that admit q as a cover. Notably, we characterize superprimitive borders. A few enumeration results are given.

1 Introduction

In recent study of repetitive structures of strings, generalized notions of periods have been introduced. A typical regularity, the period u of a given string x, grasps the repetitiveness of x since x is a prefix of a string constructed by concatenations of u. For example, $u = ab$ is a period of $x = ababa$ ($v = abab$ is also a period of x). One can define a dual notion of period, a border which is both a proper prefix and suffix of x, note that in our example, $u' = aba$ is a border of x and $x = uu'$, $v' = a$ is a border of x and $vv' = x$.

Apostolico et *al.* in [AFI91] introduced the notion of covers and gave a linear-time algorithm for finding the shortest cover: a proper factor w of x is a cover of x if x can be constructed by concatenations and superpositions of w. A string which has at least one cover is quasiperiodic and superprimitive otherwise. For example, *abaaabaaabaabaa* is quasiperiodic and its (only) cover is *abaa* whereas *abaabaaababa* is superprimitive. Note that a cover of x is necessarily at least a border of x. The notion "cover" is a generalization of period in the sense that superpositions as well as concatenations are considered to define them, whereas only concatenations are considered for period.

We provide a new characterization for the periods of a word that refines previous known results. It relies on the recent concept of quasiperiodicity [AE93,Bre92], [IM99a]. We show that exists a finite partition of the periods or borders of a word. Each element B_k of the partition contains a superprimitive border q_k and quasiperiodic borders that admit q_k as a quasiperiod. Our results rely upon a new characterization of periods that is constructive, and we show that recent characterization by Mignosi and Restivo[MR00] may be derived from it.

* This research was supported by ESPRIT LTR Project No. 20244 (ALCOM IT)
** This research was partially supported by ABISS and C.N.R.S. Program "Génomes"

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 388–396, 2000.

In the following, we denote $\mathcal{S}$ the set of unbordered words, that is words having no border and therefore no period, by $\mathcal{P}$ the set of primitive words, we recall that a word x is primitive if assuming $x = w^n$ leads to $w = x$ and $n = 1$, by $\mathcal{SP}$ the set of superprimitive words and $\mathcal{LQ}$ the set of quasiperiodic words. Given a word w that is not in $\mathcal{S}$, we denote by $B(w)$ its largest border, and $qp(w)$ its cover.

2 Period Characterization

Characterization of superprimitive words from their borders is a hard problem, to be addressed in Section 3. We need to characterize the set of periods of a given word, and notably, the fundamental properties their lengths satisfy.

Definition 1. *A word b is a* degenerated border *of a word w iff w rewrites $w = bxb$ where x is a non-empty word.*

Theorem 1. *Let $B_0 = \epsilon < B_1 < \cdots < B_k = B(w)$ be the increasing sequence of borders of a word w, and define $b_i = |B_i|$. Define:*

$$\delta_i = b_i - b_{i-1}, 1 \leq i \leq k \ .$$

Any δ_i satisfies one of the following conditions:

(a) $\delta_i = \delta_{i-1}$;
(b) $\delta_i = \sum_{j=1}^{i-1} \delta_j + \epsilon_i, \epsilon_i \geq 0$; *when* $\epsilon_i = 0$, *then* $i \leq 2$ *or exists* $j_0 \leq i - 2, \sum_{j=1}^{j_0} \delta_j < \delta_{i-1} < \sum_{j=1}^{j_0+1} \delta_j$;
(c) $\delta_i = \sum_{j=j_0+1}^{i-1} \delta_j, 1 \leq j_0 < i - 2, \delta_{j_0+1} < \delta_{i-1}$.

Remark 1. Condition (c) implies that condition (b) applied for some $j, j_0 < j \leq i - 2$, the smallest index such that $\delta_{j_0+1} \neq \delta_j$. Similarly, if a border rewrites $B_i = B_j \cdot X_j^k$, with $k \geq 1$, then $|X_j| > b_j$ and $B_{i-1} = B_j$ if $k > 1$.

Example 1. Let $W = s(as)^2s(as)^3s(as)^2vs(as)^2s(as)^3s(as)^2$ where $s \in \mathcal{SP}, as \in \mathcal{P}$ and $vs(as)^2s(as)^3s(as)^2 \in \mathcal{P}$. This can be achieved for instance if s, a and v are three different letters. Then, the set of borders is :

$$\{s; sas; s(as)^2; s(as)^2s(as)^2; s(as)^2s(as)^3s(as)^2\} \ .$$

Border construction falls in turn into the three cases:

$\delta_1 = |s| = \epsilon_1$ through condition (b);
$\delta_2 = \delta_1 + \epsilon_2$ with $|\epsilon_2 = |a|$ (condition (b));
$\delta_3 = \delta_2$ (condition (a));
$\delta_4 = \sum_{j=1}^{3} \delta_j + \epsilon_4$ with $\epsilon_4 = 0$. This is case (b) with additional constraint: $\delta_1 < \delta_3 < \delta_1 + \delta_2$.
$\delta_5 = \delta_3 + \delta_4$. This is case (c), with $j_0 = 2$ and $\delta_3 = \delta_4$.

Proof. We show how border B_i can be built from B_{i-1}. We rely on the famous equation: $xu = vx$ that implies $x = \alpha(\beta\alpha)^*$ and $u = (\beta\alpha)^*, v = (\alpha\beta)^*$ and apply it for: $B_i = B_{i-1}u = vB_{i-1}$.

(a) B_{i-1} is a degenerated border of B_i: in that case, B_i rewrites $B_{i-1}xB_{i-1}$, where xB_{i-1} is a primitive word. It follows that:

$$\delta_i = |x| + b_{i-1} = \epsilon_i + \sum_{j=1}^{i-1} \delta_j \ .$$

If $\epsilon_i \neq 0$, one can always choose x in order to have xB_{i-1} primitive. If $\epsilon_i = 0$, then $B_i = B_{i-1} \cdot B_{i-1}$ and B_{i-1} must be the largest border of B_i. This additional constraint is equivalent to $B_{i-1} \neq z^m$. To prove this, one notices that a larger border would satisfy: $B_{i-1}t = xB_{i-1}$. Moreover, equation $B_i = B_{i-1} \cdot B_{i-1}$ implies that x is a proper suffix and t is a proper prefix of B_{i-1}. Hence, $x = t$ and $xB_{i-1} = B_{i-1}x, |x| < b_{i-1}$; this implies that $B_{i-1} = z^m, m \geq 2$. In turn, this is equivalent to:

$$\begin{aligned} \delta_{i-1} &= \ldots = \delta_{i-(m-1)} \\ \delta_{i-(m-1)} &= \sum_{j=1}^{i-m} \delta_j \ . \end{aligned}$$

Contradicting this condition yields additional condition in (b).

(b) B_{i-1} is a self-overlapping border of B_i: in that case, exists a unique couple (α, β) with $\beta\alpha \in \mathcal{P}$, such that:

$$\begin{aligned} B_{i-1} &= \alpha(\beta\alpha)^m, m \geq 1 \\ B_i &= B_{i-1}(\beta\alpha)^k \end{aligned}$$

Condition $m \geq 1$ means that B_{i-1} is not a degenerated border. Condition $B_{i-1} = B(B_i)$ implies that $k = 1$. Let us consider in turn two subcases:

$m > 1$ Here, $B(B_{i-1}) = \alpha(\beta\alpha)^{m-1}$. Hence, $\delta_{i-1} = |\beta\alpha| = b_i - b_{i-1} = \delta_{i-1}$. This is case (a).

$m = 1$ Here, $B_{i-1} = \alpha \cdot \beta\alpha, \beta \neq \epsilon$ and $B_i = B_{i-1} \cdot \beta\alpha$. This implies that α is a border B_{j_0} of B_{i-1}. Hence, $|\beta\alpha| = b_{i-1} - b_{j_0} = \sum_{j=j_0}^{i-1} \delta_j$. Case $j_0 + 1 = i - 1$ implies $\delta_{i-1} = \delta_i$ which is taken into account in case (a). Now, $\beta = \epsilon$ implies that $B_{i-1} = \alpha^2, \alpha \in \mathcal{P}$. Then, $B(B_{i-1}) = \alpha = B_{i-2}$ which implies that $j_0 = i - 2$, and $\delta_{i-1} = \delta_i$, which is (a). We get condition (c).

Let us show that our theorem implies conditions recently stated by Mignosi and Restivo[MR00].

Theorem 2. *Increasing sequence δ_i satisfying the two following conditions, when $b_i \geq \delta_{i+1}$:*

(A) *exists j such that $b_j = b_i - \delta_{i+1}$;*
(B) *δ_i does not divide δ_{i+1}, unless $\delta_i = \delta_{i+1}$.*

can be associated to a set of borders of a word w. These two conditions are equivalent to:

(A') *exists* j *such that* $b_j = b_i - k\delta_{i+1}$ *for* $1 \leq k \leq \lfloor \frac{b_i}{\delta_{i+1}} \rfloor$;
(B') $\delta_i + \delta_{i+1} > b_i + \gcd(\delta_i, \delta_{i+1})$ *or* $\delta_i = \delta_{i+1}$.

Intuitively, condition (A) means that B_i admits a border of size $b_i - \delta_{i+1}$, and that its suffix of size δ_{i+1} is concatenated on the right to make B_{i+1}.

Proof. Let us prove simultaneously (A) and (B). δ_{i+1} and δ_i satisfy either one of the three conditions of Theorem 1. If condition (a) is satisfied, then $\delta_i = \delta_{i+1}$ and (B) is trivial; moreover, $j = i - 1$ satisfies condition (A) as $b_i - \delta_{i+1} = b_i - \delta_i = b_{i-1}$. Now, case (b) is not consistent with condition $b_i \geq \delta_{i+1}$, unless $\epsilon_{i+1} = 0$. In that case, $B_{i+1} = B_i \cdot B_i$ and $\delta_{i+1} = b_i = \delta_i$. (B) is trivial again and (A) is satisfied for $j = 0$ as, by definition, empty sequence ϵ is a border. Moreover, $\delta_i = \delta_{i+1}$. Finally, if condition (c) is satisfied for some j_0, then $j = j_0$. From our previous proof, B_{i+1} and B_i rewrite $\alpha(\beta\alpha)^2$ and $\alpha\beta\alpha$ with $B_{j_0} = \alpha$. The largest border of B_i, B_{i-1}, rewrites $\alpha'(\beta'\alpha')^*$ with $|\beta'\alpha'| = \delta_i$. Assume δ_i/δ_{i+1}. As $\beta\alpha$ is a suffix of $B_i = \alpha'(\beta'\alpha')^*$, one has: $\beta\alpha = (\beta'\alpha')^*$ which contradicts primitivity and (B) follows.

One can prove (A') as (A). To prove (B') assume that $\delta_i < \delta_{i+1}$. If δ_{i+1} satisfies condition (b) in Theorem 1, one has $\delta_{i+1} > b_i$ (otherwise $\delta_{i+1} = b_i = \delta_i$) and (B') is trivially satisfied. Otherwise, it satisfies condition (c). Then, rewrite:

$$\delta_i + \delta_{i+1} = \delta_i + (b_i - b_{j_0}) \ .$$

and one has:

$$\gcd(\delta_i, \delta_{i+1}) = \gcd(\delta_i, \sum_{j_0+1}^{i-1} \delta_j + \delta_i) = \gcd(\delta_i, \sum_{j_0+1}^{i-1} \delta_j) = \gcd(\delta_i, b_i - b_{j_0}) \ .$$

Rewrite $B_i = B_j.X_j^k$, with $\delta_i = |X_j|$. As B_{j_0} is an overlapping border of B_i, it is an overlapping border of B_{i-1}, hence a border of $B_j \cdot X_j$, and we get:

$$\gcd(\delta_i, \delta_{i+1}) = \gcd(|X_j|, b_i - b_j + b_j - b_{j_0}) = \gcd(|X_j|, b_j - b_{j_0})$$

If $b_{j_0} < b_j$, one gets from Remark 1 that: $|X_j| - b_{j_0} \geq b_j - b_{j_0} \geq \gcd(\delta_i, \delta_{i+1})$ and property is established. Otherwise, B_j is a border of B_{j_0} and condition (b) applied for a border construction at some step between B_{j_0} and B_jX_j. Hence, B_{j_0} is a proper border of B_jX_j and one gets:

$$\delta_i = |X_j| - b_{j_0} \geq b_{j_0} - b_j \geq \gcd(\delta_i, \delta_{i+1}) \ .$$

3 Quasiperiod Characterization

3.1 Structure of Quasiperiodic and Superprimitive Words

We now consider the derivation of superprimitive words from their borders.

Definition 2. *Let ϕ be the application from $\mathcal{SP} - \mathcal{S}$ into $\mathcal{SP}$ defined by:*

$$\phi(w) = qp(B(w)) \ .$$

Definition 3. *We define sets $\mathcal{S}_k$ and $\mathcal{Q}_k$ in the following way:*

$$\mathcal{S}_1 = \mathcal{S}$$
$$\mathcal{S}_k = \{w \in \mathcal{SP}; \phi(w) \in \mathcal{S}_{k-1}\}, \ k \geq 2$$

and:

$$\forall k \geq 1 : \mathcal{Q}_k = \{w; qp(w) \in \mathcal{S}_k\} \ .$$

Example 2. Let $q = ACATACA$ and $q' = A(CA)^2TA(CA)^2$. Both are superprimitive. Here, $B(q) = ACA$ and $B(q') = A(CA)^2$. Their common cover is ACA, which has A as the largest border. Hence, $\phi(ACA) = A \in \mathcal{SP}$; then ACA is in $\mathcal{SP}_2$ and $B(q') = A(CA)^2$ is in $\mathcal{Q}_2$. Finally, q and q' are in $\mathcal{SP}_3$.

Theorem 3. *All sets $\mathcal{Q}_k$ and $\mathcal{S}_k$ are disjoint and*

$$\mathcal{LQ} = \cup_{k\geq 1}\mathcal{Q}_k \ ,$$
$$\mathcal{SP} = \cup_{k\geq 1}\mathcal{S}_k \ .$$

Proof. It follows from the definition, that, for any (i, j), one has: $\mathcal{S}_i \cap \mathcal{Q}_j = \emptyset$. Also, due to the unicity of the cover, if sets $\mathcal{S}_k$ are disjoint, sets $\mathcal{Q}_k$ also are. To prove that sets $\mathcal{S}_k$ are disjoint, we first observe that, for any $k \geq 2$, words in $\mathcal{S}_k$ have a border, hence are not in $\mathcal{S}_1$. Assume now that $\mathcal{S}_i$ are disjoint for $1 \leq i \leq k-1$. From our previous remark, this assertion is true for $k = 3$. Assume now that $k \geq 3$ and that exists $w \in \mathcal{S}_k \cap (\cup_{i=1}^{k-1}\mathcal{S}_i) = \mathcal{S}_k \cap (\cup_{i=2}^{k-1}\mathcal{S}_i)$. Then, $B(w)$ or $qp(B(w))$ is in $\mathcal{S}_{k-1} \cap (\cup_{i=1}^{k-2}\mathcal{S}_i)$, which is a contradiction.

We now prove the inclusions. As a word is either in $\mathcal{LQ}$ or in $\mathcal{SP}$, it is enough to prove that $\mathcal{SP} = \cup_{k\geq 1}\mathcal{S}_k$. It follows from the definition that any word in $\mathcal{S}_k$ is superprimitive, hence that $\cup_{k\geq 1}\mathcal{S}_k \subseteq \mathcal{SP}$. To prove that $\mathcal{SP} \subseteq \cup_{k\geq 1}\mathcal{S}_k$, one remarks that, for any $w \in \mathcal{SP} - \mathcal{S}$, $|\phi(w)| < |w|$. Hence, sequence $\phi^i(w))_{i\geq 1}$ is finite, and its smallest element $\phi^k(w)$ is in $\mathcal{S}$. Hence, w is in $\mathcal{S}_k$.

Definition 4. *The* degree *of a word w is the index of the subset $\mathcal{Q}_k$ or $\mathcal{S}_k$ it belongs to.*

We are now ready to characterize the set of borders of a word.

Theorem 4. (k-factorization Theorem) *Any superprimitive word s_{k+1} of degree $k + 1$ factorizes as $s_{k+1} = b_k a_k b_k$ where $b_k = B(s_{k+1})$ is the largest border of s_k and $a_k b_k$ is primitive.*

Proof. First, the largest border of a superprimitive word that is not in $\mathcal{S}_1 = \mathcal{S}$ is a proper border [IM99a]. Hence, $s_{k+1} = b_k a_k b_k$. Assume that $a_k b_k$ is not primitive, e.g. that $a_k b_k = z^m, m \geq 2$. If $|z| \geq |b_k|$, z factorizes as $c_k b_k$ and $b_k c_k b_k$ is a strict cover of s_{k+1}, a contradiction. Assume now that $|z| < |b_k|$. Then the right suffix of $b_k a_k b_k$ of size $|b_k| + |z|$ factorizes as $tb_k = b_k z$. It follows that $b_k = \alpha(\beta\alpha)^*$, with $z = \beta\alpha$ and $\alpha\beta\alpha$ is a strict cover of s_{k+1}, a contradiction.

Remark 2. Primitivity condition for $a_k b_k$ is not sufficient to ensure superprimitivity of $b_k a_k b_k$. For example, let $b_3 = s(as)^2 us(as)^2$. Then, $asu.s(as)^2 us(as)^2 = a_3 b_3$ is primitive while $qp(b_3 a_3 b_3) = qp(b_3) = b_3$.

Theorem 5. *Let w be a word of index $k, k > 1$. Define, for $1 \leq i \leq k-1$:*

$$s_i(w) = inf\{v \in Border(w); degree(v) = i\}$$
$$q_i(w) = max\{v \in Border(w); degree(v) = i\} \ ,$$

and:

$$s_k(w) = qp(w), \ q_k(w) = w \ .$$

Then $s_i(w)$ is superprimitive of degree i and it is the cover of $q_i(w)$, and of all borders of degree i. Moreover, $B(s_i(w)) = q_{i-1}(w)$.

Remark 3. These definitions are meaningful, as the set of borders of index $i, 1 \leq i \leq k-1$ is non-empty.

Proof. It is enough to remark that $degree(\phi(w)) = degree(w) - 1$ and that all borders of w greater than $\phi(w)$ admit $\phi(w)$ as a cover.

3.2 Border Characterization

Observe first that main Theorem 1 is not enough to characterize superprimitivity or the degree of a word.

Example 3. Let $q = ACATACA$. then $B = \{0, 1, 3\}$. Let

$$T = ACATACA.CATACA.TACA.ACATACA.CATACA.TACA$$
$$T' = ACATACA.GGGACATACA.ACATACA.GGGACA.TACA$$

Words T and T' have the same border sizes. Nevertheless, the quasi-period of T is q, that has degree 3, while T' has quasiperiod $qGGGq$, that has degree 4.

Definition 5. *Given a superprimitive word q, one denotes its set of borders as defines its $Bord(q)$ and defines its* extension alphabet *as:*

$$\mathcal{A}(q) = \cup_{b_l \in Bord(q)} \{e_l\}$$

where $e_l = |q| - |b_l|$.

Intuitively, concatenation to q of its suffixof size e_l creates an overlapping occurrence of q. In other words, $\mathcal{A}(q)$ is the set of sizes of q-suffixes that are in q-autocorrelation set [GO81,Rég99].

Proposition 1. *Given a superprimitive word w, quasiperiodic words with quasiperiod q are uniquely associated to an m-uple on the extension alphabet, with $m \geq 2$.*

Example 4. Let $q = ACATACA$. Here, $Bord(q) = \{0, 1, 3\}$ and $\mathcal{A}(q) = \{7, 6, 4\}$. 6-uple $(7, 6, 4, 7, 6, 4)$ defines exactly text T.

Example 5. Consider border B_5 of W; its quasiperiod is B_2 and $Bord(B_2) = \{0, 1\}$ and $\mathcal{A}(B_2) = \{3, 2\}$. 7-uple $(3, 2, 3, 2, 2, 3, 2)$ characterizes B_5.

We are now ready to fully characterize the sequence of borders.

Definition 6. *Given a periodic word, one defines its canonical decomposition as the increasing sequence of its border lengths and the sequence of indexes of its superprimitive borders.*

Example 6. Index sequence of T is $(1, 2)$, index sequence of T' is $(1, 2, 3)$ and index sequence of W is $(1, 2)$.

Theorem 6. *An index sequence $(l_1, \cdots, l_k)$ is admissible iff*

(i) $l_1 = 1$;
(ii) for $k > 1$, one has:

$$\delta_{l_k} = \sum_{j=1}^{l_k - 1} \delta_j + \epsilon_k, \epsilon_k > 0$$

(iii) $\forall r : l_{k-1} < r < l_k$: *$\delta_r$ satisfies either one of conditions (a) and (c) of Theorem 1 or exists an m-uple $(w_1, \cdots, w_m)$ on $\mathcal{A}(q_{k-1})$ such that:*
  * $w_1 \cdots w_m$ *is a primitive word*
  * $\epsilon_r = w_1 + \cdots + w_m - b_{r-1}$.

Remark 4. Primitivity condition ensures that B_{r-1} is the largest border of B_r. When $r \neq l_{k-1} + 1$, B_{r-1} is quasiperiodic with period q_{k-1} and $(w_1, \cdots, w_m)$ is constrained through its suffix.

Example 7. Border B_4 of W admits B_2 as its quasiperiod. Word $(|s|, |as|, |as|)$ satisfies condition stated in (iii).

Border B_4 of T admits $B_3 = ACATACA$ as a quasiperiod. Word $(6, 4)$, associated to suffixes $CATACA$ and $TACA$ satisfies the condition.

4 Enumeration

We prove a few basic results.

Theorem 7. *Let $\mathcal{S}, \mathcal{B}_1, \mathcal{P}$ and $\mathcal{M}$ be respectively the set of unbordered words, words with one border at least, primitive words and periodic words. Define $S(z)$, $B_1(z)$, $P(z)$ and $M(z)$ as their generating functions. They satisfy the following functional equations:*

$$S(z) = (2 - S(z^2))W(z) \tag{1}$$

$$B_1(z) = (S(z^2) - 1)W(z) \tag{2}$$

$$W(z) = 1 + \sum_{m \geq 1} P(z^m) \tag{3}$$

$$M(z) = \frac{z}{1-z} P(z^2) \tag{4}$$

Proof. Proof of (1) and (2) can be found in [Rég92]. It relies on the simple observation that the smallest border of a bordered word is an unbordered word. To get equation (3), one observes that any non-empty word w rewrites in a unique manner as v^m, where v is a primitive word. Finally, a periodic word of size n is associated in a unique manner to a primitive word z of size $m \leq \frac{n}{2}$: one concatenates this word $\frac{n}{m}$ times and add a suffix of size $n \bmod m$. Equivalently:

$$M(z) = \sum_{v \in P} \sum_{u < v} \sum_{k \geq 2} z^{k|v|} z^{|u|} = \sum_{v \in P} \sum_{k \geq 2} z^{k|v|} \frac{1 - z^{|v|}}{1-z} = \frac{1}{1-z} \sum_{v \in P} z^{2|v|} \quad .$$

Corollary 1. *The number of periodic words and the number of quasiperiodic words is approximately $q^{\frac{n}{2}}$.*

Proof. Generating function of primitive words has a pole at $z = \frac{1}{q}$, which yields an asymptotic expansion q^n . It follows that generating function of periodic words has a pole at $z^2 = \frac{1}{q}$ which yields an asymptotic expansion $q^{\frac{n}{2}}$. Intuitively, dominating term counts less constrained periodic words of length n. Such words rewrite $p.p$ where p is primitive and $|p| = n/2$. And the number of such words is approximately $q^{\frac{n}{2}}$.

Less constrained quasiperiodic words are associated to two overlapping occurrences of a word s^2as^2, where as^2 is primitive. Best choice is done for $|s| = 1$, and one can chose almost all a. As a must appear twice, one only choses a word of size $n/2$, and we have $q^{\frac{n}{2}}$ such choices...

5 Conclusion

We provided a new characterization of periods. We first proved a new theorem on the lengths of periods. Then we defined a canonical partition determined by the superprimitive borders. This opens the way to new on-line algorithms to detect periods and quasiperiods.

References

AE93. A. Apostolico and A. Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theoretical Computer Science*, 119(2):247–265, 1993.

AFI91. A. Apostolico, M. Farach and C. S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39(1):17–20, 1991.

Bre92. D. Breslauer. An on-line string superprimitivity test. *Information Processing Letters*, 44(6):345–347, 1992.

GO81. L. Guibas and A.M. Odlyzko. String Overlaps, Pattern Matching and Nontransitive Games. *Journal of Combinatorial Theory,* Series A, 30:183–208, 1981.

IM99a. C.S. Iliopoulos and L. Mouchard. Quasiperiodicity: from detection to normal forms. *Journal of Automata, Languages and Combinatorics*, 4(3):213-228, 1999.

IM99b. C. S. Iliopoulos and L. Mouchard. An $o(n \log n)$ algorithm for computing all maximal quasiperiodicities in strings. In C. S. Calude and M. J. Dinneen, editors, *Combinatorics, Computation and Logic. Proceedings of DMTCS'99 and CATS'99*, Lecture Notes in Computer Science, pages 262–272, Auckland, New-Zealand, 1999.

IM99c. C.S. Iliopoulos and L. Mouchard. Quasiperiodicity and string covering. *Theoretical Computer Science*, 218(1):205–216, 1999.

Lot83. Lothaire. *Combinatorics on Words.* Addison-Wesley, Reading, Mass., 1983.

MR00. F. Mignosi and A. Restivo. Periodicity (Chapter 9). *Algebraic Combinatorics on Words.* to appear; preliminary version at `http://www-igm.univ-mlv.fr/~{}berstel/Lothaire/`, 2000.

Rég92. M. Régnier. Enumeration of bordered words. *RAIRO Theoretical Informatics and Applications*, 26,4:303–317, 1992.

Rég99. M. Régnier. A Unified Approach to Word Occurrences Probabilities. *Discrete Applied Mathematics*, 1999. to appear; preliminary version at RECOMB'98.

Finding Maximal Quasiperiodicities in Strings

Gerth Stølting Brodal and Christian N.S. Pedersen

Basic Research in Computer Science (BRICS)
Centre of the Danish National Research Foundation
Department of Computer Science, University of Aarhus
Ny Munkegade, 8000 Århus C, Denmark
{gerth,cstorm}@brics.dk

Abstract. Apostolico and Ehrenfeucht defined the notion of a maximal quasiperiodic substring and gave an algorithm that finds all maximal quasiperiodic substrings in a string of length n in time $O(n \log^2 n)$. In this paper we give an algorithm that finds all maximal quasiperiodic substrings in a string of length n in time $O(n \log n)$ and space $O(n)$. Our algorithm uses the suffix tree as the fundamental data structure combined with efficient methods for merging and performing multiple searches in search trees. Besides finding all maximal quasiperiodic substrings, our algorithm also marks the nodes in the suffix tree that have a superprimitive path-label.

1 Introduction

Characterizing and finding regularities in strings are important problems in many areas of science. In molecular biology repetitive elements in chromosomes determine the likelihood of certain diseases. In probability theory regularities are important in the analysis of stochastic processes. In computer science repetitive elements in strings are important in e.g. data compression, speech recognition, coding, automata and formal language theory.

A widely studied regularity in strings are consecutive occurrences of the same substring. Two consecutive occurrences of the same substring is called an occurrence of a *square* or a *tandem repeat*. In the beginning of the last century, Thue [25,26] showed how to construct arbitrary long strings over any alphabet of more than two characters that contain no squares. Since then a lot of work have focused on developing efficient methods to count or detect squares in strings. Several methods [18,23,12] can determine if a string of length n contains a square in time $O(n)$, and methods [11,6,17,24] can find occurrences of squares in a string of length n in time $O(n \log n)$ plus the time it takes to output the detected squares. Recently two methods [16,14] have been presented that find a compact representation of all squares in a string of length n in time $O(n)$.

A way to describe the regularity of an entire string in terms of repetitive substrings is the notion of a *periodic* string. Gusfield [13, page 40] defines string S as periodic if it can be constructed by concatenations of a shorter string α. The shortest string from which S can be generated by concatenations is the *period* of S. A string that is not periodic is *primitive*. Some regularities in strings

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 397–411, 2002.

cannot be characterized efficiently using periods or squares. To remedy this, Ehrenfeucht, as referred in [3], suggested the notation of a *quasiperiodic* string. A string S is quasiperiodic if it can be constructed by concatenations and superpositions of a shorter string α. We say that α covers S. Several strings might cover S. The shortest string that covers S is the *quasiperiod* of S. A covering of S implies that S contains a square, so by the result of Thue not all strings are quasiperiodic. A string that is not quasiperiodic is *superprimitive*. Apostolico, Farach and Iliopoulos [5] presented an algorithm that finds the quasiperiod of a given string of length n in time $O(n)$. This algorithm was simplified and made on-line by Breslauer [7]. Moore and Smyth [22] presented an algorithm that finds all substrings that covers a given string of length n in time $O(n)$.

Similar to the period of a string, the quasiperiod of a string describes a global property of the string, but quasiperiods can also be used to characterize substrings. Apostolico and Ehrenfeucht [4] introduced the notion of maximal quasiperiodic substrings of a string. Informally, a quasiperiodic substring γ of S with quasiperiod α is maximal if no extension of γ can be covered by α or αa, where a is the character following γ in S. Apostolico and Ehrenfeucht showed that the maximal quasiperiodic substrings of S correspond to path-labels of certain nodes in the suffix tree of S, and gave an algorithm that finds all maximal quasiperiodic substrings of a string of length n in time $O(n \log^2 n)$ and space $O(n \log n)$. The algorithm is based on a bottom-up traversal of the suffix tree in which maximal quasiperiodic substrings are detected at the nodes in the suffix tree by maintaining various data structures during the traversal. The general structure of the algorithm resembles the structure of the algorithm by Apostolico and Preparata [6] for finding tandem repeats.

In this paper we present an algorithm that finds all maximal quasiperiodic substrings in a string of length n in time $O(n \log n)$ and space $O(n)$. Similar to the algorithm by Apostolico and Ehrenfeucht, our algorithm finds the maximal quasiperiodic substrings in a bottom-up traversal of the suffix tree. The improved time and space bound is a result of using efficient methods for merging and performing multiple searches in search trees, combined with observing that some of the work done, and data stored, by the Apostolico and Ehrenfeucht algorithm is avoidable. The analysis of our algorithm is based on a stronger version of the well known "smaller-half trick" used in the algorithms in [11,6,24] for finding tandem repeats. The stronger version of the "smaller-half trick" is hinted at in [20, Exercise 35] and stated in Lemma 6. In [21, Chapter 5] it is used in the analysis of finger searches, and in [8] it is used in the analysis and formulation of an algorithm to find all maximal pairs with bounded gap in a string.

Recently, and independent of our work, Iliopoulos and Mouchard in [15] report an algorithm with running time $O(n \log n)$ for finding all maximal quasiperiodic substrings in a string of length n. Their algorithm differs from our algorithm as it does not use the suffix tree as the fundamental data structure, but uses the partitioning technique used by Crochemore [11] combined with several other data structures. Finding maximal quasiperiodic substrings can thus be

done in two different ways similar to the difference between the algorithms by Crochemore [11] and Apostolico and Preparata [6] for finding tandem repeats.

The rest of this paper is organized as follows. In Sect. 2 we define the preliminaries used in the rest of the paper. In Sect. 3 we state and prove properties of quasiperiodic substrings and suffix trees. In Sect. 4 we state and prove results about efficient merging of and searching in height-balanced trees. In Sect. 5 we stated our algorithm to find all maximal quasiperiodic substrings in a string. In Sect. 6 we analyze the running time of our algorithm and in Sect. 7 we show how the algorithm can be implemented to use linear space.

2 Definitions

In the following we let $S, \alpha, \beta, \gamma \in \Sigma^*$ denote strings over some finite alphabet Σ. We let $|s|$ denote the length of S, $S[i]$ the ith character in S for $1 \leq i \leq |S|$, and $S[i\,..\,j] = S[i]S[i+1]\cdots S[j]$ a substring of S. A string α *occurs* in a string γ at position i if $\alpha = \gamma[i\,..\,i+|\alpha|-1]$. We say that $\gamma[j]$, for all $i \leq j \leq i+|\alpha|-1$, is covered by the occurrence of α at position i.

A string α *covers* a string γ if every position in γ is covered by an occurrence of α. Figure 1 shows that $\gamma = abaabaabaabaab$ is covered by $\alpha = abaab$. Note that if α covers γ then α is both a prefix and a suffix of γ. A string is *quasiperiodic* if it can be covered by a shorter string. A string is *superprimitive* if it is not quasiperiodic, that is, if it cannot be covered by a shorter string. A superprimitive string α is a quasiperiod of a string γ if α covers γ. In Lemma 1 we show that if α is unique, and α is therefore denoted the *quasiperiod* of γ.

The *suffix tree* $T(S)$ of the string S is the compressed trie of all suffixes of the string $S\$$, where $\$ \notin \Sigma$. Each leaf in $T(S)$ represents a suffix $S[i\,..\,n]$ of S and is annotated with the index i. We refer to the set of indices stored at the leaves in the subtree rooted at node v as the *leaf-list* of v and denote it $LL(v)$. Each edge in $T(S)$ is labelled with a nonempty substring of S such that the path from the root to the leaf annotated with index i spells the suffix $S[i\,..\,n]$. We refer to the substring of S spelled by the path from the root to node v as the *path-label* of v and denote it $L(v)$. Figure 2 shows a suffix tree.

For a node v in $T(S)$ we partition $LL(v) = (i_1, i_2, \ldots, i_k)$, where $i_j < i_{j+1}$ for $1 \leq j < k$, into a sequence of disjoint subsequences $R_1, R_2, \ldots, R_r$, such that each R_ℓ is a maximal subsequence $i_a, i_{a+1}, \ldots, i_b$, where $i_{j+1} - i_j \leq |L(v)|$ for $a \leq j < b$. Each R_ℓ is denoted a *run* at v and represents a maximal substring of S that can be covered by $L(v)$, i.e. $L(v)$ covers $S[\min R_\ell\,..\,|L(v)|-1+\max R_\ell]$, and we say that R_ℓ is a run from $\min R_\ell$ to $|L(v)|-1+\max R_\ell$. A run R_ℓ at v is said to *coalesce* at v if R_ℓ contains indices from at least two children of v, i.e. if for no child w of v we have $R_\ell \subseteq LL(w)$. We use $C(v)$ to denote the set of coalescing runs at v.

3 Maximal Quasiperiodic Substrings

If S is a string and $\gamma = S[i\,..\,j]$ a substring covered by a shorter string $\alpha = S[i\,..\,i+|\alpha|-1]$, then γ is quasiperiodic and we describe it by the triple $(i, j, |\alpha|)$. A

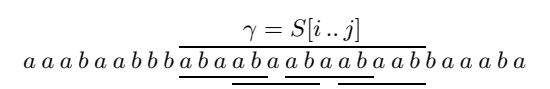


Fig. 1. The substring $\gamma = abaabaabaabaab$ is a maximal quasiperiodic substring with quasiperiod $\alpha = abaab$. Note that the quasiperiod α covers the substring γ.

triple $(i, j, |\alpha|)$ describes a *maximal quasiperiodic substring* of S, in the following abbreviated *MQS*, if the following requirements are satisfied.

1. $\gamma = S[i \,..\, j]$ is quasiperiodic with quasiperiod α.
2. If α covers $S[i' \,..\, j']$, where $i' \leq i \leq j \leq j'$, then $i' = i$ and $j' = j$.
3. $\alpha S[j+1]$ does not cover $S[i \,..\, j+1]$.

Figure 1 shows a maximal quasiperiodic substring. The problem we consider in this paper is for a string S to generate all triples $(i, j, |\alpha|)$ that describe MQSs. This problem was first studied by Apostolico and Ehrenfeucht in [4]. In the following we state important properties of quasiperiodic substrings which are essential to the algorithm to be presented.

Lemma 1. *Every quasiperiodic string γ has a unique quasiperiod α.*

Proof. Assume that γ is covered by two distinct superprimitive strings α and β. Since α and β are prefixes of γ we can without loss of generality assume that α is a proper prefix of β. Since α and β are suffixes of γ, then α is also a proper suffix of β. Since α and β cover γ, and α is a prefix and suffix of β it follows that α covers β, implying the contradiction that β is not superprimitive. □

Lemma 2. *If γ occurs at position i and j in S, and $1 \leq j - i \leq |\gamma|/2$, then γ is quasiperiodic.*

Proof. Let α be the prefix of γ of length $|\gamma| - (j - i)$, i.e. $\alpha = S[i \,..\, i + |\gamma| - (j - i) - 1] = S[j \,..\, i + |\gamma| - 1]$. Since $j - i \leq |\gamma|/2$ implies that $i - 1 + |\gamma| - (j - i) \geq j - 1$, we conclude that α covers γ. □

Lemma 3. *If the triple $(i, j, |\alpha|)$ describes a MQS in S, then there exists a non-leaf node in the suffix tree $T(S)$ with path-label α.*

Proof. Assume that α covers the quasiperiodic substring $S[i \,..\, j]$ and that no node in $T(S)$ has path-label α. Since all occurrences of α in S are followed by the same character $a = S[i + |\alpha|]$, αa must cover $S[i \,..\, j + 1]$, contradicting the maximality requirement 3. □

Lemma 4. *If γ is a quasiperiodic substring in S with quasiperiod α and u is a non-leaf node in the suffix tree $T(S)$ with path-label γ, then there exists an ancestor node v of u in $T(S)$ with path-label α.*

Proof. Since u is a non-leaf node in $T(S)$ of degree at least two, there exist characters a and b such that both γa and γb occur in S. Since α is a suffix of γ we then have that both αa and αb occur in S, i.e. there exist two suffixes of S having respectively prefix αa and αb, implying that there exists a node v in $T(S)$ with $L(v) = \alpha$. Since α is also a prefix of γ, v is an ancestor of u in $T(S)$. □

Lemma 5. *If v is a node in the suffix tree $T(S)$ with a superprimitive path-label α, then the triple $(i, j, |\alpha|)$ describes a MQS in S if and only if there is a run R from i to j that coalesces at v.*

Proof. Let $(i, j, |\alpha|)$ describe a MQS in S and assume that the run $R \in C(v)$ from i and j does not coalesce at v. Then there exists a child v' of v in $T(S)$ such that $R \subseteq LL(v')$. The first symbol along the edge from v to v' is $a = S[i + |\alpha|]$. Every occurrence of α in R is thus followed by a, i.e. αa covers $S[i \,..\, j + 1]$. This contradicts the maximality requirement 3 and shows the "if" part of the theorem.

Let R be a coalescing run from i to j at node v, i.e. $L(v) = \alpha$ covers $S[i \,..\, j]$, and let $a = S[j + 1]$. To show that $(i, j, |\alpha|)$ describes a MQS in S it is sufficient to show that αa does not cover $S[i \,..\, j + 1]$. Since R coalesces at v, there exists a minimal $i'' \in R$ such that αa does not occur in S at position i''. If $i'' = i = \min R$ then αa cannot cover S at position i'' since it by the definition of R cannot occur any position ℓ in S satisfying $i - |\alpha| \leq \ell \leq i$. If $i'' \neq i = \min R$ then αa occurs at $\min R$ and $\max R$, i.e. there exists $i', i''' \in R$, such that $i' < i'' < i'''$, αa occurs at i' and i''' in S, and αa does not occour at any position ℓ in S satisfying $i' < \ell < i'''$. To conclude that $(i, j, |\alpha|)$ describes a MQS we only have to show that $S[i''' - 1]$ is not covered by the occourence of αa at position i', i.e. $i''' - i' > |\alpha| + 1$. By Lemma 2 follows that $i'' - i' > |\alpha|/2$ and $i''' - i'' > |\alpha|/2$, so $i''' - i' \geq |\alpha| + 1$. Now assume that $i''' - i' = |\alpha| + 1$. This implies that $|\alpha|$ is odd and that $i'' - i' = i''' - i'' = (|\alpha| + 1)/2$. Using this we get

$$a = S[i' + |\alpha|] = S[i'' + (|\alpha| - 1)/2] = S[i''' + (|\alpha| - 1)/2] = S[i'' + |\alpha|] \neq a \,.$$

This contradiction shows that $(i, j, |\alpha|)$ describes a MQS and shows the "only if" part of the theorem. □

Theorem 1. *Let v be a non-leaf node in $T(S)$ with path-label α. Since v is a non-leaf node in $T(S)$ there exists $i_1, i_2 \in LL(v)$ such that $S[i_1 + |\alpha|] \neq S[i_2 + |\alpha|]$. The path-label α is quasiperiodic if and only if there exists an ancestor node $u \neq v$ of v in $T(S)$ with path-label β that for $\ell = 1$ or $\ell = 2$ satisfies the following two conditions.*

1. *Both i_ℓ and $i_\ell + |\alpha| - |\beta|$ belong to a coalescing run R at u, and*
2. *for all $i', i'' \in LL(u)$, $|i' - i''| > |\beta|/2$.*

Proof. If α is superprimitive, then no string β covers α, i.e. there exists no node u in $T(S)$ where $C(u)$ includes a run containing both i_ℓ and $i_\ell + |\alpha| - |\beta|$ for $\ell = 1$ or $\ell = 2$. If α is quasiperiodic, then we argue that the quasiperiod β of α satisfies conditions 1 and 2. Since β is superprimitive, condition 2 is satisfied

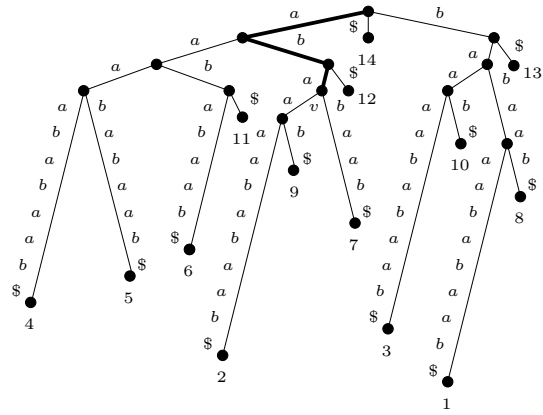


Fig. 2. The suffix tree of the string *babaaaababaab*. Node v has a superprimitive path-label *aba*. There is a coalescing run at v from 7 to 11. Hence the substring *ababa* occurring at position 7 in *babaaaababaab* is a maximal quasiperiodic substring.

by Lemma 2. Since β is the quasiperiod of α, we by Lemma 4 have that β is the path-label of a node u in $T(S)$. Since $\beta = S[i_1 .. i_1+|\beta|-1] = S[i_2 .. i_2+|\beta|-1] = S[i_1+|\alpha|-|\beta| .. i_1+|\alpha|-1] = S[i_2+|\alpha|-|\beta| .. i_2+|\alpha|-1]$ and $S[i_1+|\alpha|] \neq S[i_2+|\alpha|]$ then either $S[i_1+|\alpha|] \neq S[i_1+|\beta|]$ or $S[i_2+|\alpha|] \neq S[i_2+|\beta|]$, which implies that either i_1 and $i_1+|\alpha|-|\beta|$ are in a coalescing run at u, or i_2 and $i_2+|\alpha|-|\beta|$ are in a coalescing run at u. Hence, condition 1 is satisfied. □

Theorem 2. *A triple $(i, j, |\alpha|)$ describes a MQS in S if and only if the following three requirements are satisfied*

1. *There exists a non-leaf node v in $T(S)$ with path-label α.*
2. *The path-label α is superprimitive.*
3. *There exists a coalescing run R from i to j at v.*

Proof. The theorem follows directly from the definition of MQS, Lemma 3 and Lemma 5. □

Figure 2 illustrates the properties described by Theorem 2.

4 Searching and Merging Height-Balanced Trees

In this section we consider various operations on height-balanced binary trees [2], e.g. AVL-trees [1], and present an extension of the well-known "smaller-half trick" which implies a non-trivial bound on the time it takes to perform a sequence of operations on height-balanced binary trees. This bound is essential to the running time of our algorithm for finding maximal quasiperiodic substrings to be presented in the next section.

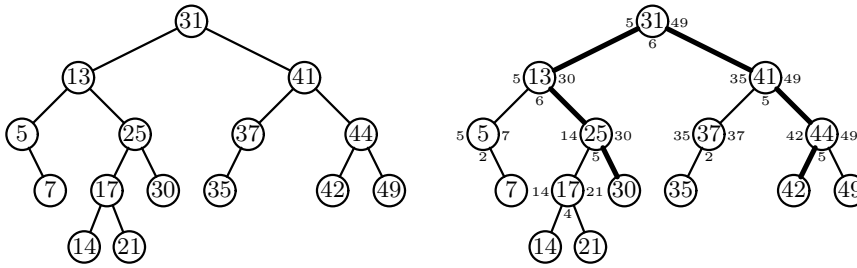


Fig. 3. A height-balanced tree with 15 elements, and the corresponding extended height-balanced tree. Each node in the extended height-balanced tree with at least one child is annotated with min (left), max (right) and max-gap (bottom). The emphasized path is the search path for Δ-Pred$(T, 4, 42)$

For a sorted list $L = (x_1, \ldots, x_n)$ of n distinct elements, and an element x and a value δ, we define the following functions which capture the notation of *predecessors* and *successors* of an element, and the notation of *Δ-predecessors* and *Δ-successors* which in Sect. 5 will be used to compute the head and the tail of a coalescing run.

$$\begin{aligned}
\text{pred}(L, x) &= \max\{y \in L \mid y \leq x\}\ , \\
\text{succ}(L, x) &= \min\{y \in L \mid y \geq x\}\ , \\
\text{max-gap}(L) &= \max\{0, x_2 - x_1, x_3 - x_2, \ldots, x_n - x_{n-1}\}\ , \\
\Delta\text{-pred}(L, \delta, x) &= \min\{y \in L \mid y \leq x \ \wedge\ \text{max-gap}(L \cap [y, x]) \leq \delta\}\ , \\
\Delta\text{-succ}(L, \delta, x) &= \max\{y \in L \mid y \geq x \ \wedge\ \text{max-gap}(L \cap [x, y]) \leq \delta\}\ .
\end{aligned}$$

If $L = (5, 7, 13, 14, 17, 21, 25, 30, 31)$, then $\text{pred}(L, 20) = 17$, $\text{succ}(L, 20) = 21$, $\text{max-gap}(L) = 13 - 7 = 6$, $\Delta\text{-pred}(L, 4, 20) = 13$, and $\Delta\text{-succ}(L, 4, 20) = 25$. Note that $\text{pred}(L, x) = \Delta\text{-pred}(L, 0, x)$ and $\text{succ}(L, x) = \Delta\text{-succ}(L, 0, x)$.

We consider an extension of hight-balanced trees where each node v in addition to $\mathsf{key}(v)$, $\mathsf{height}(v)$, $\mathsf{left}(v)$, $\mathsf{right}(v)$, and $\mathsf{parent}(v)$, which respectively stores the element at v, the height of the subtree T_v rooted at v, pointers to the left and right children of v and a pointer to the parent node of v, also stores the following information: $\mathsf{previous}(v)$ and $\mathsf{next}(v)$ are pointers to the nodes which store the immediate predecessor and successor elements of $\mathsf{key}(v)$ in the sorted list, $\mathsf{min}(v)$ and $\mathsf{max}(v)$ are pointers to the nodes storing the smallest and largest elements in the subtree rooted at v, and $\mathsf{max\text{-}gap}(v)$ is the value of max-gap applied to the list of all elements in the subtree T_v rooted at v. Figure 3 shows a height-balanced tree and the corresponding extended height-balanced tree (previous and next pointers are omitted in the figure).

If v has a left child v_1, $\mathsf{min}(v)$ points to $\mathsf{min}(v_1)$. Otherwise $\mathsf{min}(v)$ points to v. Symmetrically, if v has a right child v_2, $\mathsf{max}(v)$ points to $\mathsf{max}(v_2)$. Otherwise $\mathsf{max}(v)$ points to v. If v stores element e and has a left child v_1 and a right

child v_2, then $\mathsf{max\text{-}gap}(v)$ can be computed as

$$\mathsf{max\text{-}gap}(v) = \max\{0, \mathsf{max\text{-}gap}(v_1), \mathsf{max\text{-}gap}(v_2), \mathsf{key}(v) - \mathsf{key}(\mathsf{max}(v_1)), \mathsf{key}(\mathsf{min}(v_2)) - \mathsf{key}(v)\}\,. \quad (1)$$

If v_1 and/or v_2 do not exist, then the expression is reduced by removing the parts of the expression involving the missing nodes/node. The equation can be used to recompute the information at nodes being rotated when rebalancing a height-balanced search tree. Similar to the function max-gap(L) and the operation $\mathsf{max\text{-}gap}(v)$, we can define and support the function min-gap(L) and the operation $\mathsf{min\text{-}gap}(v)$. The operations we consider supported for an extended height-balanced tree T are the following, where $e_1, \ldots, e_k$ denotes a sorted list of k distinct elements. The output of the four last operations is a list of k pointers to nodes in T containing the answer to each search key e_i.

- $\mathsf{MultiInsert}(T, e_1, \ldots, e_k)$ inserts (or merges) the k elements into T.
- $\mathsf{MultiPred}(T, e_1, \ldots, e_k)$ for each e_i finds $\text{pred}(T, e_i)$.
- $\mathsf{MultiSucc}(T, e_1, \ldots, e_k)$ for each e_i finds $\text{succ}(T, e_i)$.
- $\mathsf{Multi\text{-}}\Delta\mathsf{\text{-}Pred}(T, \delta, e_1, \ldots, e_k)$ for each e_i finds $\Delta\text{-pred}(T, \delta, e_i)$.
- $\mathsf{Multi\text{-}}\Delta\mathsf{\text{-}Succ}(T, \delta, e_1, \ldots, e_k)$ for each e_i finds $\Delta\text{-succ}(T, \delta, e_i)$.

We merge two height-balanced trees T and T', $|T| \geq |T'|$, by inserting the elements in T' into T, i.e. $\mathsf{MultiInsert}(T, e_1, e_2, \ldots, e_k)$ where $e_1, e_2, \ldots, e_k$ are the elements in T' in sorted order. The following theorem states the running time of the operations.

Theorem 3. *Each of the operations* MultiInsert*,* MultiPred*,* MultiSucc*,* Multi-Δ-Pred*, and* Multi-Δ-Succ *can be performed in time* $O(k \cdot \max\{1, \log(n/k)\})$*, where* n *is the size of the tree and* k *is the number elements to be inserted or searched for.*

Proof. If $k \geq n$, the theorem follows immediately. In the following we therefore assume $k \leq n$. Brown and Tarjan in [10] show how to merge two height-balanced trees in time $O(k \cdot \max\{1, \log(n/k)\})$, especially their algorithm performs k top-down searches in time $O(k \cdot \max\{1, \log(n/k)\})$. Since a search for an element e either finds the element e or the predecessor/successor of e it follows that MultiPred and MultiSucc can be computed in time $O(k \cdot \max\{1, \log(n/k)\})$ using the previous and next pointers. The implementation of MultiInsert follows from the algorithm of [10] by observing that only the $O(k \cdot \max\{1, \log(n/k)\})$ nodes visited by the merging need to have their associated min, max and max-gap information recomputed due to the inserted elements, and the recomputing can be done in a traversal of these nodes in time $O(k \cdot \max\{1, \log(n/k)\})$ using Equation 1. The implementation of the Multi-Δ-Pred and Multi-Δ-Succ operations is more technical. For the details see [9, Sect. 4]. □

If each node in a binary tree supplies a term $O(k)$, where k is the number of leaves in the smallest subtree rooted at a child of the node, then the sum over

all terms is $O(N \log N)$. In the literature, this bound is often referred to as the "smaller-half trick". It is essential to the running time of several methods for finding tandem repeats [11,6,24]. Our method for finding maximal quasiperiodic substrings uses a stronger version of the "smaller-half trick" hinted at in [20, Exercise 35] and stated in Lemma 6. It implies that we at every node in a binary tree with N leaves can perform a fixed number of the operations stated in Theorem 3, with n and k as stated in the lemma, in total time $O(N \log N)$.

Lemma 6. *If each internal node v in a binary tree with N leaves supplies a term $O(k \log(n/k))$, where n is the number of leaves in the subtree rooted at v and $k \leq n/2$ is the number of leaves in the smallest subtree rooted at a child of v, then the sum over all terms is $O(N \log N)$.*

5 Algorithm

The algorithm to find all maximal quasiperiodic substrings in a string S of length n first constructs the suffix tree $T(S)$ of S in time $O(n)$ using any existing suffix tree construction algorithm, e.g. [28,19,27], and then processes $T(S)$ in two phases. Each phase involves one or more traversals of $T(S)$. In the first phase the algorithm identifies all nodes of $T(S)$ with a superprimitive path-label. In the second phase the algorithm reports the maximal quasiperiodic substrings in S. This is done by reporting the coalescing runs at the nodes which in the first phase were identified to have superprimitive path-labels.

To identify nodes with superprimitive path-labels we apply the concepts of *questions*, *characteristic occurrences* of a path-label, and *sentinels* of a node. Let v be a non-leaf node in $T(S)$ and $u \neq v$ an ancestor node of v in $T(S)$. Let v_1 and v_2 be the two leftmost children of v, and $i_1 = \min(LL(v_1))$ and $i_2 = \min(LL(v_2))$. A *question posed to u* is a triple (i, j, v) where $i \in LL(v) \subset LL(u)$ and $j = i + |L(v)| - |L(u)| \in LL(u)$, and the answer to the question is true if and only if i and j are in the same coalescing run at u.

We define the two occurrences of $L(v)$ at positions i_1 and i_2 to be the *characteristic occurrences* of $L(v)$, and define the *sentinels* $\hat{v}_1$ and $\hat{v}_2$ of v as the positions immediately after the two characteristic occurrences of $L(v)$, i.e. $\hat{v}_1 = i_1 + |L(v)|$ and $\hat{v}_2 = i_2 + |L(v)|$. Since i_1 and i_2 are indices in leaf-lists of two distinct children of v, we have $S[\hat{v}_1] \neq S[\hat{v}_2]$. In the following we let $SL(v)$ be the list of the sentinels of the nodes in the subtree rooted at v in $T(S)$. Since there are two sentinels for each non-leaf node $|SL(v)| \leq 2|LL(v)| - 2$.

Theorem 1 implies the following technical lemma which forms the basis for detecting nodes with superprimitive path-labels in $T(S)$.

Lemma 7. *The path-label $L(v)$ is quasiperiodic if and only if there exists a sentinel $\hat{v}$ of v, and an ancestor w of v (possibly $w = v$) for which there exists $j \in LL(w) \cap]\hat{v} - 2 \cdot \text{min-gap}(LL(w)) \,;\, \hat{v}[$ such that $(\hat{v} - |L(v)|, j, v)$ is a question that can be posed and answered successfully at an ancestor node $u \neq v$ of w (possibly $u = w$) with $|L(u)| = \hat{v} - j$ and $\text{min-gap}(LL(u)) > |L(u)|/2$.*

Proof. If there exists a question $(\hat{v} - |L(v)|, \hat{v} - |L(u)|, v)$ that can be answered successfully at u, then $\hat{v} - |L(v)|$ and $\hat{v} - |L(u)|$ are in the same run at u, i.e. $L(u)$ covers $L(v)$ and $L(v)$ is quasiperiodic.

If $L(v)$ is quasiperiodic, we have from Theorem 1 that there for $i_\ell = \hat{v}_\ell - |L(v)|$, where $\ell = 1$ or $\ell = 2$, exists an ancestor node $u \neq v$ of v where both i_ℓ and $i_\ell + |L(v)| - |L(u)|$ belong to a coalescing run at u and min-gap$(LL(u)) > |L(u)|/2$. The lemma follows by letting $w = u$ and $j = \hat{v}_\ell - |L(u)|$. □

Since j and $\hat{v}$ uniquely determine the question $(\hat{v} - |L(v)|, j, v)$, it follows that to decide the superprimitivity of all nodes it is sufficient for each node w to find all pairs $(\hat{v}, j)$ where $\hat{v} \in SL(w)$ and $j \in LL(w) \cap]\hat{v} - 2 \cdot \text{min-gap}(LL(w))\,;\, \hat{v}[$, or equivalently $j \in LL(w)$ and $\hat{v} \in SL(w) \cap]j\,;\, j + 2 \cdot \text{min-gap}(LL(w))[$. Furthermore, if $\hat{v}$ and j result in a question at w, but $j \in LL(w')$ and $\hat{v} \in SL(w')$ for some child w' of w, then $\hat{v}$ and j result in the same question at w' since min-gap$(LL(w')) \geq$ min-gap$(LL(w))$, i.e. we only need to find all pairs $(\hat{v}, j)$ at w where $\hat{v}$ and j come from two distinct children of w. We can now state the details of the algorithm.

Phase I – Marking Nodes with Quasiperiodic Path-Labels. In Phase I we mark all nodes in $T(S)$ that have a quasiperiodic path-label by performing three traversals of $T(S)$. We first make a depth-first traversal of $T(S)$ where we for each node v compute min-gap$(LL(v))$. We do this by constructing for each node v a search tree $T_{LL}(v)$ that stores $LL(v)$ and supports the operations in Sect. 4. In particular the root of $T_{LL}(v)$ should store the value $\mathsf{min\text{-}gap}(T_{LL}(v))$ to be assigned to v. If v is a leaf, $T_{LL}(v)$ only contains the index annotated to v. If v is an internal node, we construct $T_{LL}(v)$ by merging the T_{LL} trees of the children of v from left-to-right when these have been computed. If the children of v are $v_1, \ldots, v_k$ we merge $T_{LL}(v_1), \ldots, T_{LL}(v_{i+1})$ by performing a binary merge of $T_{LL}(v_{i+1})$ with the result of merging $T_{LL}(v_1), \ldots, T_{LL}(v_i)$. As a side effect of computing $T_{LL}(v)$ the T_{LL} trees of the children of v are destroyed.

We pose and answer questions in two traversals of $T(S)$ explained below as Step 1 and Step 2. For each node v we let $Q(v)$ contain the list of questions posed at v. Inititially $Q(v)$ is empty.

Step 1 (Generating Questions). In this step we perform a depth-first traversal of $T(S)$. At each node v we construct search trees $T_{LL}(v)$ and $T_{SL}(v)$ which store respectively $LL(v)$ and $SL(v)$ and support the operations mentioned in Sect. 4. For a non-leaf node v with leftmost children v_1 and v_2, we compute the sentinels of v as $\hat{v}_1 = \mathsf{min}(T_{LL}(v_1)) + |LL(v_1)|$ and $\hat{v}_2 = \mathsf{min}(T_{LL}(v_2)) + |LL(v_1)|$. The T_{LL} trees need to be recomputed since these are destroyed in the first traversal of $T(S)$. The computation of $T_{SL}(v)$ is done similarly to the computation of $T_{LL}(v)$ by merging the T_{SL} lists of the children of v from left-to-right, except that after the merging the T_{SL} trees of the children we also need to insert the two sentinels $\hat{v}_1$ and $\hat{v}_2$ in $T_{SL}(v)$.

We visit node v, and call it the *current node*, when the T_{LL} and T_{SL} trees at the children of v are available. During the traversal we maintain an array depth

such that $\mathsf{depth}(k)$ refers to the node u on the path from the current node to the root with $|L(u)| = k$ if such a node exists. Otherwise $\mathsf{depth}(k)$ is undef. We maintain depth by setting $\mathsf{depth}(|L(u)|)$ to u when we arrive at u from its parent, and setting $\mathsf{depth}(|L(u)|)$ to undef when we return from u to its parent.

When v is the current node we have from Lemma 7 that it is sufficient to generate questions for pairs $(\hat{w}, j)$ where $\hat{w}$ and j come from two different children of v. We do this while merging the T_{LL} and T_{SL} trees of the children. Let the children of v be $v_1, \ldots, v_k$. Assume $LL_i = LL(v_1) \cup \cdots \cup LL(v_i)$ and $SL_i = SL(v_1) \cup \cdots \cup SL(v_i)$ has been computed as T_{LL_i} and T_{SL_i} and we are in the process of computing LL_{i+1} and SL_{i+1}. The questions we need to generate while computing LL_{i+1} and SL_{i+1} are those where $j \in LL_i$ and $\hat{w} \in SL(v_{i+1})$ or $j \in LL(v_{i+1})$ and $\hat{w} \in SL_i$. Assume $j \in T_{LL}$ and $\hat{w} \in T_{SL}$, where either $T_{LL} = T_{LL_i}$ and $T_{SL} = T_{SL}(v_{i+1})$ or $T_{LL} = T_{LL}(v_{i+1})$ and $T_{SL} = T_{SL_i}$. There are two cases. If $|T_{LL}| \leq |T_{SL}|$ we locate each $j \in T_{LL}$ in T_{SL} by performing a $\mathsf{MultiSucc}$ operation. Using the next pointers we can then for each j report those $\hat{w} \in T_{SL}$ where $\hat{w} \in \,]j\,;\, j + 2 \cdot \text{min-gap}(LL(v))[$. If $|T_{LL}| > |T_{SL}|$ we locate each $\hat{w} \in T_{SL}$ in T_{LL} by performing a $\mathsf{MultiPred}$ operation. Using the $\mathsf{previous}$ pointers we can then for each $\hat{w}$ report those $j \in T_{SL}$ where $j \in \,]\hat{w} - 2 \cdot \text{min-gap}(LL(v))\,;\, \hat{w}[$. The two sentinels $\hat{v}_1$ and $\hat{v}_2$ of v are handled similarly to the later case by performing two searches in $T_{LL}(v)$ and using the $\mathsf{previous}$ pointers to generate the required pairs involving the sentinels $\hat{v}_1$ and $\hat{v}_2$ of v.

For a pair $(\hat{w}, j)$ that is generated at the current node v, we generate a question $(\hat{w} - |L(w)|, j, w)$ about descendent w of v with sentinel $\hat{w}$, and pose the question at ancestor $u = \mathsf{depth}(\hat{w} - j)$ by inserting $(\hat{w} - |L(w)|, j, w)$ into $Q(u)$. If such an ancestor u does not exists, i.e. $\mathsf{depth}(\hat{w} - j)$ is undef, or $\mathsf{min\text{-}gap}(u) \leq |L(u)|/2$ then no question is posed.

Step 2 (Answering Questions). Let $Q(v)$ be the set of questions posed at node v in Step 1. If there is a coalescing run R in $C(v)$ and a question (i, j, w) in $Q(v)$ such that $\min R \leq i < j \leq \max R$, then i and j are in the same coalescing run at v and we mark node w as having a quasiperiodic path-label.

We identify each coalescing run R in $C(v)$ by the tuple $(\min R, \max R)$. We *answer* question (i, j, w) in $Q(v)$ by deciding if there is a run $(\min R, \max R)$ in $C(v)$ such that $\min R \leq i < j \leq \max R$. If the questions (i, j, w) in $Q(v)$ and runs $(\min R, \max R)$ in $C(v)$ are sorted lexicographically, we can answer all questions by a linear scan through $Q(v)$ and $C(v)$. In the following we describe how to generate $C(v)$ in sorted order and how to sort $Q(v)$.

Constructing Coalescing Runs. The coalescing runs are generated in a traversal of $T(S)$. At each node v we construct $T_{LL}(v)$ storing $LL(v)$. We construct $T_{LL}(v)$ by merging the T_{LL} trees of the children of v from left-to-right. A coalescing run R in $LL(v)$ contains an index from at least two distinct children of v, i.e. there are indices $i' \in LL(v_1)$ and $i'' \in LL(v_2)$ in R for two distinct children v_1 and v_2 of v such that $i' < i''$ are neighbors in $LL(v)$ and $i'' - i' \leq |L(v)|$. We say that i' is a *seed* of R. We identify R by the tuple $(\min R, \max R)$. We have $\min R = \Delta\text{-pred}(LL(v), |L(v)|, i')$ and $\max R = \Delta\text{-succ}(LL(v), |L(v)|, i')$.

To construct $C(v)$ we collect seeds $i_{r_1}, i_{r_2}, \ldots, i_{r_k}$ of every coalescing run in $LL(v)$ in sorted order. This done by checking while merging the T_{LL} trees of the children of v if an index gets a new neighbor in which case the index can be identified as a seed. Since each insertion at most generates two seeds we can collect all seeds into a sorted list while performing the merging. From the seeds we can compute the first and last index of the coalesing runs by doing Multi-Δ-Pred$(T_{LL}(v), |L(v)|, i_{r_1}, i_{r_2}, \ldots, i_{r_k})$ and Multi-Δ-Succ$(T_{LL}(v), |L(v)|, i_{r_1}, i_{r_2}, \ldots, i_{r_k})$. Since we might have collected several seeds of the same run, the list of coalescing runs $R_1, R_2, \ldots, R_k$ might contain duplets which can be removed by reading through the list once. Since the seeds are collected in sorted order, the resulting list of runs is also sorted.

Sorting the Questions. We collect the elements in $Q(v)$ for every node v in $T(S)$ into a single list Q that contains all question (i, j, w) posed at nodes in $T(S)$. We annotate every element in Q with the node v it was collected from. By construction every question (i, j, w) posed at a node in $T(S)$ satisfies that $0 \leq i < j < n$. We can thus sort the elements in Q lexicographically with respect to i and j using radix sort. After sorting the elements in Q we distribute the questions back to the proper nodes in sorted order by a linear scan through Q.

Phase II – Reporting Maximal Quasiperiodic Substrings. After Phase I all nodes that have a quasiperiodic path-label are marked, i.e. all unmarked nodes are nodes that have a superprimitive path-label. By Theorem 2 we report all maximal quasiperiodic substrings by reporting the coalescing runs at every node that has a superprimitive path-label. In a traversal of the marked suffix tree we as in Phase I construct $C(v)$ at every unmarked node and report for every R in $C(v)$ the triple $(\min R, \max R, |L(v)|)$ that identifies the corresponding maximal quasiperiodic substring.

6 Running Time

In every phase of the algorithm we traverse the suffix tree and construct at each node v search trees that stores $LL(v)$ and/or $SL(v)$. At every node v we construct various lists by considering the children of v from left-to-right and perform a constant number of the operations in Theorem 3. Since the overall merging of information in $T(S)$ is done by binary merging we by Lemma 6 have that this amounts to time $O(n \log n)$ in total. To generate and answer questions we use time proportional to the total number of questions generated. Lemma 8 state that the number of questions is bounded by $O(n \log n)$. We conclude that the running time of the algorithm is $O(n \log n)$.

Lemma 8. *At most $O(n \log n)$ questions are generated.*

Proof. We prove that each of the $2n$ sentinels can at most result in the generation of $O(\log n)$ questions. Consider a sentinel $\hat{w}$ of node w and assume that it generates a question $(\hat{w} - |L(w)|, j, w)$ at node v. Since $\hat{w} - j < 2 \cdot \text{min-gap}(LL(v))$,

j is either $\text{pred}(LL(v), \hat{w} - 1)$ (a question of Type A) or the left neighbor of $\text{pred}(LL(v), \hat{w} - 1)$ in $LL(v)$ (a question of Type B). For $\hat{w}$ we first consider all indices resulting in questions of Type A along the path from w to the root. Note that this is an increasing sequence of indices. We now show that the distance of $\hat{w}$ to the indices is geometrically decreasing, i.e. there are at most $O(\log n)$ questions generated of Type A. Let j and j' be two consecutive indices resulting in questions of Type A at node v and at an ancestor node u of v. Since $j < j' < \hat{w}$ and $j' - j \geq \text{min-gap}(LL(u))$ and $\hat{w} - j' < 2 \cdot \text{min-gap}(LL(u))$, we have that $\hat{w} - j' < \frac{2}{3}(\hat{w} - j)$. Similarly we can bound the number of questions generated of Type B for sentinel $\hat{w}$ by $O(\log n)$. □

7 Achieving Linear Space

Storing the suffix tree $T(S)$ uses space $O(n)$. During a traversal of the suffix tree we construct search trees as explained. Since no element, index or sentinel, at any time is stored in more than a constant number of search trees, storing the search trees uses space $O(n)$. Unfortunately, storing the sets $C(v)$ and $Q(v)$ of coalescing runs and questions at every node v in the suffix tree uses space $O(n \log n)$. To reduce the space consumption we must thus avoid to store $C(v)$ and $Q(v)$ at all nodes simultaneously. The trick is to modify Phase I to alternate between generating and answering questions.

We observe that generating questions and coalescing runs (Step 1 and the first part of Step 2) can be done in a single traversal of the suffix tree. This traversal is Part 1 of Phase I. Answering questions (the last part of Step 1) is Part 2 of Phase I. To reduce the space used by the algorithm to $O(n)$ we modify Phase I to alternate in rounds between Part 1 (generating questions and coalescing runs) and Part 2 (answering questions).

We say that node v is *ready* if $C(v)$ is available and all questions from it has been generated, i.e. Part 1 has been performed on it. If node v is ready then all nodes in its subtree are ready. Since all questions to node v are generated at nodes in its subtree, this implies that $Q(v)$ is also available. By definition no coalescing runs are stored at non-ready nodes and Lemma 9 states that only $O(n)$ questions are stored at non-ready nodes. In a round we produce ready nodes (perform Part 1) until the number of questions plus coalescing runs stored at nodes readied in the round exceed n, we then answer the questions (perform Part 2) at nodes readied in the round. After a round we dispose questions and coalescing runs stored at nodes readied in the round. We continue until all nodes in the suffix tree have been visited.

Lemma 9. *There are at most $O(n)$ questions stored at non-ready nodes.*

Proof. Let v be a node in $T(S)$ such that all nodes on the path from v to the root are non-ready. Consider a sentinel $\hat{w}$ corresponding to a node in the subtree rooted at v. Assume that this sentinel has induced three questions $(\hat{w} - |L(w)|, j', w)$, $(\hat{w} - |L(w)|, j'', w)$ and $(\hat{w} - |L(w)|, j''', w)$, where $j' < j'' < j'''$, that are posed at ancestors of v. By choice of v, these ancestors are non-ready nodes. One of the ancestors is node $u = \mathsf{depth}(\hat{w} - j')$. Since question $(\hat{w} -$

$|L(w)|, j', w)$ is posed at u, min-gap$(LL(u)) > |L(u)|/2$. Since $j', j'', j''' \in LL(u)$ and $j''' - j' \leq \hat{w} - j' = |L(u)|$, it follows that min-gap$(LL(u)) \leq \min\{j'' - j', j''' - j''\} \leq |L(u)|/2$. This contradicts that min-gap$(LL(u)) > |L(u)|/2$ and shows that each sentinel has generated at most two questions to non-ready nodes. The lemma follows because there are at most $2n$ sentinels in total. □

Alternating between Part 1 and Part 2 clearly results in generating and answering the same questions as if Part 1 and Part 2 were performed without alternation. The correctness of the algorithm is thus unaffected by the modification of Phase I. Now consider the running time. The running time of a round can be divided into time spent on readying nodes (Part 1) and time spent on answering questions (Part 2). The total time spent on readying nodes is clearly unaffected by the alternation. To conclude the same for the total time spent on answering questions, we must argue that the time spent on sorting the posed questions in each round is proportional to the time otherwise spent in the round.

The crucial observation is that each round takes time $\Omega(n)$ for posing questions and identifying coalescing runs, implying that the $O(n)$ term in each radix sorting is neglectable. We conclude that the running time is unaffected by the modification of Phase I. Finally consider the space used by the modified algorithm. Besides storing the suffix tree and the search trees which uses space $O(n)$, it only stores $O(n)$ questions and coalescing runs at nodes readied in the current round (by construction of a round) and $O(n)$ questions at non-ready nodes (by Lemma 9). In summary we have the following theorem.

Theorem 4. *All maximal quasiperiodic substrings of a string of length n can be found in time $O(n \log n)$ and space $O(n)$.*

References

1. G. M. Adel'son-Vel'skii and Y. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146:263–266, 1962. English translation in *Soviet Math. Dokl.*, 3:1259–1262.
2. A. V. Aho, J. E. Hopcraft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison–Wesley, Reading, Massachusetts, 1974.
3. A. Apostolico and D. Breslauer. Of periods, quasiperiods, repetitions and covers. In *A selection of essays in honor of A. Ehrenfeucht*, volume 1261 of *Lecture Notes in Computer Science*. Springer, 1997.
4. A. Apostolico and A. Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theoretical Computer Science*, 119:247–265, 1993.
5. A. Apostolico, M. Farach, and C. S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39:17–20, 1991.
6. A. Apostolico and F. P. Preparata. Optimal off-line detection of repetitions in a string. *Theoretical Computer Science*, 22:297–315, 1983.
7. D. Breslauer. An on-line string superprimitivity test. *Information Processing Letters*, 44:345–347, 1992.
8. G. S. Brodal, R. B. Lyngsø, C. N. S. Pedersen, and J. Stoye. Finding maximal pairs with bounded gap. In *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1645 of *Lecture Notes in Computer Science*, pages 134–149, 1999.

9. G. S. Brodal and C. N. S. Pedersen. Finding maximal quasiperiodicities in strings. Technical Report RS-99-25, BRICS, September 1999.
10. M. R. Brown and R. E. Tarjan. A fast merging algorithm. *Journal of the ACM*, 26(2):211–226, 1979.
11. M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981.
12. M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.
13. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
14. D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. Technical Report CSE-98-4, Department of Computer Science, UC Davis, 1998.
15. C. S. Iliopoulos and L. Mouchard. Quasiperiodicity: From detection to normal forms. *Journal of Automata, Languages and Combinatorics*, 4(3):213–228, 1999.
16. R. Kolpakov and G. Kucherov. Maximal repetitions in words or how to find all squares in linear time. Technical Report 98-R-227, LORIA, 1998.
17. M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5:422–432, 1984.
18. M. G. Main and R. J. Lorentz. Linear time recognition of squarefree strings. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*, pages 271–278. Springer, Berlin, 1985.
19. E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
20. K. Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, 1994.
21. K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
22. D. Moore and W. F. Smyth. Computing the covers of a string in linear time. In *Proceedings of the 5th Annual Symposium on Discrete Algorithms (SODA)*, pages 511–515, 1994.
23. M. Rabin. Discovering repetitions in strings. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*, pages 279–288. Springer, Berlin, 1985.
24. J. Stoye and D. Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. In *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1448 of *Lecture Notes in Computer Science*, pages 140–152, 1998.
25. A. Thue. Über unendliche Zeichenreihen. *Skrifter udgivet af Videnskabsselskabet i Christiania, Mathematisk- og Naturvidenskabeligklasse*, 7:1–22, 1906.
26. A. Thue. Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen. *Skrifter udgivet af Videnskabsselskabet i Christiania, Mathematisk- og Naturvidenskabeligklasse*, 1:1–67, 1912.
27. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.
28. P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Symposium on Switching and Automata Theory*, pages 1–11, 1973.

On the Complexity of Determining the Period of a String

Artur Czumaj[1]* and Leszek Gąsieniec[2]**

[1] Department of Computer and Information Science
New Jersey Institute of Technology, Newark, NJ 07102-1982, USA
czumaj@cis.njit.edu
[2] Department of Computer Science, University of Liverpool
Peach Str., L69 7ZF Liverpool, United Kingdom
leszek@csc.liv.ac.uk

Abstract. We study the complexity of a classical combinatorial problem of computing the period of a string. We investigate both the average- and the worst-case complexity of the problem.
We deliver almost tight bounds for the average-case. We show that every algorithm computing the period must examine $\Omega(\sqrt{m})$ symbols of an input string of length m. On the other hand we present an algorithm that computes the period by examining on average $O\left(\sqrt{m \cdot \log_{|\Sigma|} m}\right)$ symbols, where $|\Sigma| \geq 2$ stands for the input alphabet.
We also present a deterministic algorithm that computes the period of a string using $m + O(m^{3/4})$ comparisons. This is the first algorithm that have the worst-case complexity $m + o(m)$.

1 Introduction

The studies on *string periodicity* remain a central topic in combinatorial pattern matching due to important applications of periodicity in string searching algorithms, algebra, and in formal language theory (see, e.g., [1,26,36,38,39]).

Let $S =< S[1], S[2], \ldots, S[m] >$ be a string over an alphabet Σ with $|\Sigma| \geq 2$. We shall frequently denote S by $S[1 \cdots m]$. An initial segment $S[1, \ldots, i]$, for $i = 1, \ldots, m$, is called a prefix of string S, and final segment $S[j, \ldots, m]$ is called a suffix of S. A *period* of S is a positive integer p such that for any i, $1 \leq i \leq m - p$, $S[i] = S[i + p]$. The shortest period of string S is called *the period* of S and we will denote it by $per(S)$. Since any $p \geq m$ is a period of S, we always have $1 \leq per(S) \leq m$. String S is called *primitive* iff $per(S) = m$ and S is called *periodic* if $per(S) \leq \frac{m}{2}$.

Example 1.1. String $abracadabra$ has two periods: 7 and 10, and $per(abracadabra) = 7$. On the other hand, strings $artur$ and $leszek$ are primitive and their periods are 5 and 6, respectively. String $blablablablablablabl$ is "highly periodic" and has non-trivial periods 3, 6, 9, 12, 15, and 18. Clearly, $per(blablablablablablabl) = 3$ and $blablablablablablabl$ can be represented as $(bla)^6bl$.

* Work partly done while the author was with the Heinz Nixdorf Institute and Department of Mathematics & Computer Science at the University of Paderborn, D-33095 Paderborn, Germany.
** Supported in part by Nuffield Foundation Award for newly appointed lecturers NUF-NAL.

R. Giancarlo and D. Sankoff (Eds.): CPM 2000, LNCS 1848, pp. 412–422, 2000.

The following lemma describing basic property of periods of a string is known as *periodicity lemma*, see [40].

Lemma 1.1. *If string S has two periods p, q, s.t. $p + q < |S|$ then string S has also a period $gcd(p, q)$.*

The notion of a period plays a crucial role in the design of efficient string matching algorithms. The first linear-time string matching algorithms are due to Knuth, Morris, and Pratt (called later KMP algorithm) [37], and Boyer and Moore (called later BM algorithm) [8] explored the periodicity of strings. The periodicity played also a key role in later developments in string matching including: constant-space linear-time string matching [21,25,33,35] and optimal parallel string matching [11,15,23,27,30,43,44]. A careful study of two-dimensional periodicity led to the optimal alphabet-independent algorithms for two-dimensional pattern matching [2,3,4,32,24]. Many other applications of periodicities can be found in [38,39].

String Matching Problem. Given two strings: pattern $P \in \Sigma^m$, and text $T \in \Sigma^n$, where Σ is an alphabet of size ≥ 2. We say that pattern P occurs at position i in text T iff $P[1 \ldots m] = T[i \ldots i+m-1]$. A *string matching* problem is to find all occurrences of pattern P in text T.

Almost every string matching algorithm works in two separate stages: *pattern preprocessing* and *text searching* stage. During the first stage the algorithm performs an analysis of a pattern and computes certain functions that can be used to speed up the text searching stage. The emphasis in design of efficient string matching algorithms is on the complexity of the searching stage. We shall briefly sketch the existing results in this area.

We start with the average-case complexity. In this case, the lower bound was established by Yao, see [45]. He used a model in which time complexity is measured in terms of a number of examined text symbols, where pattern P is arbitrary and text T is taken uniformly at random from the set of all strings of length n over alphabet $|\Sigma| \geq 2$. Yao proved that in this model every string matching algorithm must examine $\Omega\left(\frac{n}{m} \cdot \log_{|\Sigma|} m\right)$ text symbols. The upper bound is due to Knuth *et al.* [37]. They gave a simple string matching algorithm having the average-case complexity $O\left(\frac{n}{m} \cdot \log_{|\Sigma|} m\right)$ that matches the lower bound in the model explored by Yao. An alternative optimal on average and linear in the worst-case algorithm was proposed later by Crochemore *et al.* in [22].

The complexity of the searching stage is also well studied in the worst-case scenario. In this model the complexity is expressed in a number of symbol comparisons used by an algorithm. There are known several linear-time (i.e., $O(n + m)$-time) string matching algorithms (see, e.g., [7,8,22,28,29,33,37,42], and survey expositions [1,6,20,36]). Recently, an exact complexity of string matching searching stage has been a subject for intensive studies. Initially Rivest [41] showed that any searching algorithm makes at least $n - m + 1$ comparisons in the worst case. This bound was later improved by Galil and Giancarlo [31], and then by Cole *et al.* to $n + \frac{2}{m+3}(n - m)$ comparisons, see [17].

On the other hand it is known that searching stage in KMP algorithm performs at most $2\,n-m+1$ comparisons [37] and the searching stage in BM algorithm uses at most

$3\,n-\Omega(n/m)$ comparisons [14]; and these bounds are tight. The $2\,n-o(n)$ barrier was first beaten by Colussi *et al.*, see [18]. They designed a simple variant of KMP algorithm that performs at most $\frac{3}{2}\,n$ comparisons. A similar result was obtained independently by Apostolico and Crochemore [5]. Colussi, Galil and Giancarlo [19] gave an algorithm that makes at most $n+\frac{1}{3}\,(n-m)$ comparisons in the search phase. This bound was improved later by Breslauer and Galil [13], who obtained a searching stage (followed by a linear-time preprocessing phase) that uses $n+\frac{4\log m+2}{m}\,(n-m)$ comparisons in the worst-case scenario. Finally, this bound was improved by Cole and Hariharan in [16], where they present the searching stage requiring at most $n+\frac{8}{3(m+1)}\,(n-m)$ comparisons, but followed by a quadratic-time preprocessing stage.

Computing the Period. A problem of computing the period of a string is a crucial task in the pattern preprocessing stage of most of string matching algorithms. It is computed in different forms like a *failure table* in KMP and BM algorithms, *witnesses* in parallel string matching [43,11,12], and *2d-witnesses* in 2d-pattern matching [2,4,32]. However a problem of determining the exact complexity of computing the period has been almost neglected so far. Breslauer *et al.* proved that one needs $2m-O(m^{1/2})$ comparisons to compute the failure table in KMP algorithm, for details see [9,10]. But the first attempt to efficient period computation was presented by Gąsieniec *et al.* in the context of constant-space string matching, see [34]. They designed a procedure that computes the period of a string of size m using only $(1+\varepsilon)m$ comparisons, for any constant $\varepsilon>0$. In this paper we investigate both the average-case as well as the worst-case complexity of determining the period of a string.

2 Average-Case Analysis

We start our considerations focusing on the average-case complexity of determining the period. In what follows we assume that input string is taken uniformly at random from the set of all strings of length m over alphabet $|\Sigma|\geq 2$. One can easily show that most of the strings of length m have the period of size $m-O(1)$. Therefore there exists a straightforward algorithm that computes the period of almost all strings in constant time. However, we are interested in computing the period of any string rather than the period for most of strings. We show that this requires significantly more work. We prove that each algorithm that determines the period of a string of length m requires examination (even in the *best case*!) of $\Omega(\sqrt{m})$ symbols. This result shows that as in parallel string matching, compare [12] and [23], searching stage in sequential string matching is significantly easier than pattern preprocessing (computing the period). The results of Yao [45] and Knuth [37] mentioned above imply that for $m\approx n$ and binary alphabet, the average-case complexity of string matching is $\Theta(\log m)$, while for the problem of determining the period is as hard as $\Omega(\sqrt{m})$.

We also present an algorithm that almost matches our lower bound for the average-case complexity. We design a simple algorithm that determines the period of a string of length m by examining on average $O\left(\sqrt{m\cdot\log_{|\Sigma|}m}\right)$ symbols. We also conjecture that the average complexity of the problem of determining the period is

$\Theta\left(\sqrt{m \cdot \log_{|\Sigma|} m}\right)$. Our upper and lower bounds meet for large alphabets ($|\Sigma|$ being a polynomial in m), but for binary alphabets the gap is of order $\sqrt{\log m}$.

Our average-case bounds can be also interpreted in terms of a *communication complexity* of computing the period of a string. In this model one party, called Alice, has a string S of length m and the other party, called Bob, must determine the period of S by asking for the symbols of S. Our lower bound implies that for every string Bob must ask on at least $\Omega(\sqrt{m})$ symbols and it is easy to see that there are strings for which Bob must know all m symbols in order to compute the period (for example, this is the case for every string with period $\lceil \frac{1}{3}m \rceil$). On the other hand, our upper bound tells that for most of the input strings Bob can compute the period by asking Alice on $O\left(\sqrt{m \log_{|\Sigma|} m}\right)$ symbols. In this context, an interesting feature of our algorithm is that to achieve such low complexity it is enough (for Bob) to ask Alice only in *two rounds*. In the first round Bob asks Alice to reveal him $\Theta\left(\sqrt{m \log_{|\Sigma|} m}\right)$ symbols at carefully chosen positions of the input string. After receiving these symbols Bob can determine the period for most of the strings. In the rare situation when the revealed symbols do not determine the period, Bob asks Alice to show the remaining symbols.

Theorem 2.1. *The average-case complexity of finding the period of string of length m is at least $\Omega(\sqrt{m})$ and at most $O\left(\sqrt{m \cdot \log_{|\Sigma|} m}\right)$.*

2.1 Lower Bound

Consider an algorithm A that determines periods of strings of length m. Let $S \in \Sigma^m$ be a string for which A returns the value $per(S)$ after inspecting positions $I_s \subseteq \{1, \ldots, m\}$ of S. Observe first that the set $\{per(S)+1, \ldots, m\}$ must be always a subset of I_S. Indeed, if $i \notin I_S$ for some $per(S) < i \leq m$, then the adversary always could set $s_i \neq s_{i-per(S)}$, contrary to the definition of $per(S)$. Therefore, if $per(S) \leq m - \sqrt{m}$, then $\{per(S)+1, \ldots, m\} \subseteq I_S$, and hence $|I_S| \geq \sqrt{m}$. Now we consider the case when $per(S) > m - \sqrt{m}$. In this case it must be possible to deduce from examined positions of S that for all integer $k < per(S)$, k is not a period of S. In particular, this implies that for every value k with $\frac{m}{2} < k \leq m - \sqrt{m}$, there must exist some $i \in I_S$ such that $i + k \in I_S$. (The requirement that $k > \frac{m}{2}$ implies that this condition is necessary. For small k it would be already sufficient that for some $i \in I_S$ and $j \in \mathbb{N}$ we had $i + j\ k \in I_S$.) Since $k > \frac{m}{2}$, we have $1 \leq i \leq \frac{m}{2}$. Let $A_S = I_S \cap \{1, \ldots, \lfloor \frac{m}{2} \rfloor\}$ and $B_S = I_S \setminus A_S$. If we denote $A_S + B_S = \{i+j \ : \ i \in A_S, j \in B_S\}$, then the arguments above yield $\{\lfloor \frac{m}{2} \rfloor + 1, \ldots, m - \sqrt{m}\} \subseteq A_S + B_S$. Therefore, the pigeonhole principle implies that

$$|I_S| = |A_S| + |B_S| \geq 2\sqrt{\tfrac{m}{2} - \sqrt{m}} = \Omega(\sqrt{m}) \ .$$

2.2 Upper Bound

To prove the upper bound we will use the optimal on the average string matching algorithm due to Knuth *et al.* [37]. Let $M = \left\lceil \sqrt{m \cdot \log_{|\Sigma|} m} \right\rceil$. In what follows we will call

prefix $P =< S_1, \ldots, S_M >$ of S as a pattern, and suffix $T =< S_2, \ldots, S_m >$ of S as a text.

Initially we use the algorithm from [37] to find all occurrences of pattern P in text T. One can easily show that with overwhelming probability pattern P never occurs in text T. Then, for every k, $1 \leq k \leq m - M$ there is a j_k, $1 \leq j_k \leq M$, such that $p_{j_k} \neq t_{k+j_k-1}$. This implies that $S_{j_k} \neq S_{k+j_k}$, and hence any $k = 1, \ldots, m - M$ cannot be a period of S. Therefore, if P never occurs in T, then $per(S) > m - M$. In this case, in order to determine the period of S it is enough to find the longest prefix of P which is a suffix S of length M. For this purpose we use deterministic KMP algorithm [37]. Otherwise, if P occurs in T, our algorithm will examine all symbols of S. Since this event is very unlikely, the case when P occurs in T does not affect the complexity of the algorithm.

The cost of the search for pattern P in text T using the algorithm of Knuth *et al.* [37] is:

$$O\left(\frac{|T_S|}{|P_S|} \cdot \log_{|\Sigma|} m\right) = O\left(\sqrt{m \cdot \log_{|\Sigma|} m}\right)$$

and the cost of comparing pattern P with a suffix S of length M by KMP algorithm is $O(M) = O\left(\sqrt{m \cdot \log_{|\Sigma|} m}\right)$. Therefore, the average-case complexity of the problem is $O\left(\sqrt{m \cdot \log_{|\Sigma|} m}\right)$.

2.3 "Oblivious" Algorithm

In this subsection we describe another algorithm which can be modeled in a very simple, "oblivious" way. Let $q = \lceil \log_{|\Sigma|}(m^2) \rceil = \Theta(\log_{|\Sigma|} m)$ and $M = \lceil \sqrt{m \cdot q} \rceil = \Theta\left(\sqrt{m \cdot \log_{|\Sigma|} m}\right)$. Let us define $\mathcal{A} = \{1, \ldots, M\}$, $\mathcal{B} = \{m-q+1, \ldots, m\}$, and for every integer i, $1 \leq i \leq \left\lceil \frac{\lceil \frac{1}{2}(m-q) \rceil}{M-q+1} \right\rceil$, let $\mathcal{C}_i = \left\{ \frac{m-q+1}{2} + i(N-q+1), \ldots, \frac{m-q+1}{2} + i(M-q+1) + q - 1 \right\}$. Then, if we set $I_E = \mathcal{A} \cup \mathcal{B} \cup \bigcup_i \mathcal{C}_i$, then we will prove that the following algorithm determines the period and examines $|I_E| = \Theta\left(\sqrt{m \cdot \log_{|\Sigma|} m}\right)$ symbols with probability at least $1 - \frac{1}{m}$:

(1) Examine the symbols s_j with $j \in I_E$.
(2) If $per(S)$ cannot be determined after Step 1, then examine all symbols of S.

Theorem 2.2. *The average-case complexity of the algorithm above is* $\Theta\left(\sqrt{m \cdot \log_{|\Sigma|} m}\right)$.

To prove the theorem we need the following lemma.

Lemma 2.1. *If the input string is taken uniformly at random from Σ^m, then its period is determined after Step 1 of the algorithm above with probability at least $1 - \frac{1}{m}$.*

Proof. We first show that with probability at least $1 - \frac{1}{m}$, for every $k = \lceil \frac{m-q+1}{2} \rceil$, $\ldots, m-q$, there is j such that the following three conditions hold:

(1) $j \in \mathcal{C}_i$ for certain i,
(2) $j - k \in \mathcal{A}$, and
(3) $s_j \neq s_{j-k}$.

Having that, we know that after Step 1 either $1 \leq per(S) < \lceil \frac{m-q+1}{2} \rceil$ or $m-q+1 \leq per(S) \leq m$. Once we know that $per(S) \geq m-q+1$, we can easily determine the period of S by examining symbols S_j with $j \in \mathcal{A} \cup \mathcal{B}$. And indeed, one can easily verify that if $per(S) \geq m-q+1$, then $per(S) = \min\{m, \min\{p \in \mathcal{B} : \forall\, j \in \mathcal{A}\ ((j \leq m-p) \Rightarrow (S_j = S_{j+p}))\}\}$.

Thus it remains to show that with probability at least $1-\frac{1}{m}$, for every $k \in \{\lceil \frac{m-q+1}{2} \rceil, \ldots, m-q\}$, there is $j \in \mathcal{C}_i$ for certain i, such that $j-k \in \mathcal{A}$, and $S_j \neq S_{j-k}$. Choose any k from set $\{\lceil \frac{m-q+1}{2} \rceil, \ldots, m-q\}$ and find i such that $\lceil \frac{m-q+1}{2} \rceil + (i-1)(M-q+1) \leq k \leq \lceil \frac{m-q+1}{2} \rceil + i(M-q+1) - 1$. Now observe that for every $j \in \mathcal{C}_i$ it holds $j-k \in \mathcal{A}$. (Indeed: $j-k \geq (\lceil \frac{m-q+1}{2} \rceil + i(M-q+1)) - (\lceil \frac{m-q+1}{2} \rceil + i(M-q+1) - 1) = 1$ and $j-k \leq (\lceil \frac{m-q+1}{2} \rceil + i(M-q+1) + q - 1) - (\lceil \frac{m-q+1}{2} \rceil + (i-1)(M-q+1)) = M$.)

Let X_k be the event that $S_j = S_{j-k}$ for each $j \in \mathcal{C}_i$. Since for each $j \in \mathcal{C}_i$ we also have $j-k \in \mathcal{A}$, it is remains to show that

$$\mathbf{Pr}\left[\exists k : \tfrac{m-q+1}{2} \leq k \leq m-q \ \text{ and } \ X_k = 1\right] \leq \frac{1}{m} \ .$$

Since the symbols s_l for $l \in \mathcal{C}_i$ and $l+k \in \mathcal{C}_i$ are chosen independently and uniformly at random, we have

$$\mathbf{Pr}[X_k = 1] = \prod_{j \in \mathcal{C}_i} \mathbf{Pr}[s_j = s_{j-k}] = \left(\frac{1}{|\Sigma|}\right)^q \ .$$

Now one can apply the Boole-Benferoni inequality to obtain

$$\mathbf{Pr}\left[\exists k : \lceil \tfrac{m-q+1}{2} \rceil \leq k \leq m-q \text{ and } X_k = 1\right] \leq \left\lceil \frac{m-q}{2} \right\rceil \cdot \left(\frac{1}{|\Sigma|}\right)^q \ .$$

Since $q \geq \log_{|\Sigma|}(m^2/2)$, the RHS is at most $\frac{1}{m}$ and the lemma follows.

3 Worst-Case Analysis

In this section we show that there exists deterministic algorithm computing the period of a string S of length m, in time $m + o(m)$. A similar algorithm was used by Gąsieniec *et al.* in [34] in the context of *constant space string matching*. Their algorithm uses $O((1+\varepsilon)m)$ comparisons, for any constant $\varepsilon > 0$. The improvement is possible since

here we use more space to store information that can be useful while processing an input string.

The algorithm uses a paradigm of a failure table F, introduced by Knuth, Morris, and Pratt in [37]. For each position i in string S, value $F(i)$ stands for a length of the longest prefix of $S[1,\ldots,i]$ which is also a suffix of $S[1,\ldots,i]$. Our algorithm computes entries of table F only for a certain number of positions. Computation of values for all entries is not feasible due to the lower bound $2m - O(\sqrt{m})$ proved for the string prefix-matching problem, for details see [9]. The algorithm uses also, as a *black-box* subroutine, on-line comparison based string matching algorithm (called later BG) due to Breslauer and Galil, for details see [13]. Algorithm BG searches for pattern occurrences using no more than $n + \frac{4\log m+2}{m}(n-m)$ comparisons, and it is followed by a linear preprocessing stage.

Our algorithm has two stages: preprocessing and actual search for the period of the input string S. While executing preprocessing stage the algorithm computes complete failure table $F[1,\ldots,m^{3/4}]$ for prefix of string S of size $m^{3/4}$ and performs complete linear time preprocessing of BG algorithm on prefix π of size $m^{1/2}$. The actual search for the period of string S is performed as follows:

SEARCHING STAGE

(1) Traverse the consecutive symbols of S from left to right and apply on-line string matching algorithm BG to find the first occurrence of prefix π. If the first occurrence is found at position i then go to step 2; otherwise, if no occurrence is found, then process naively the prefix and the suffix of S of size $m^{1/2}$ each to compute the period of S (here we use the fact that S has the period of size at least $m - m^{1/2}$).
(2) Find the longest prefix of S starting at position i by testing consecutive symbols in S and update failure table F whenever it is possible;
   (a) if the longest prefix ends at the end of S, the period of size i has been found, STOP.
   (b) if mismatch x occurs before the end of S (prefix of size $x-i$ occurs at position i) and value $F(x-i)$ is defined perform shift of size $x - i - F(x-i)$, and if $F(x-i) > m^{1/4}$ continue step 2 with new $i = x - F(x-i)$; otherwise go back to step 1, with starting position $i = x - F(x-i)$.
   (c) if a mismatch x occurs before the end of S and value $F(x-i)$ is not defined (this may happen when $x - i > m^{3/4}$) perform shift of size $x - i - m^{1/2}$, and go back to step 1 with starting position $i = x - m^{1/2}$.

We will prove the following theorem:

Theorem 3.1. *The algorithm computes the period of a string S of length m using $m + O(m^{3/4})$ comparisons.*

Proof. Assume first that a prefix of string S of size $m^{1/4}$ is non-periodic (all shifts in step 2 have size $\geq \frac{m^{1/4}}{2}$, due to periodicity Lemma 1.1). A periodic case is presented at the end of the proof.

The preprocessing phase is completed using $2m^{3/4} + O(m^{1/2})$ comparisons: computing a failure table for prefix of size $m^{3/4}$ (use standard KMP algorithm) and preprocessing in BG algorithm on prefix π. We need the following lemma:

Lemma 3.1. *The algorithm computes values in failure table F for all positions $j = m^{1/2}, \ldots, m$ in string S, such that $F(j) \geq m^{1/2}$.*

Proof. The proof is done by induction on the length of processed prefix $S[1, \ldots, j]$. Assume that after scanning initial j positions in string S the algorithm computed values in failure table F for all positions $i \leq j$, s.t. $F(i) \geq m^{1/2}$. Notice that in the beginning we compute values for all positions $i \leq m^{3/4}$. Later, when the algorithm is executed two cases hold:

(1) while executing step 1 the algorithm searches for the first occurrence of prefix π, and until the new occurrence is found the values of failure function F at visited positions are smaller than $m^{1/2}$ (they won't be used by the algorithm),
(2) while executing step 2 the algorithm is updating the content of failure table F (whenever value is $\geq m^{1/2}$) and it is capable of doing it due to the inductive assumption.

The execution of the algorithm is a sequence of independent stages $u_1, \ldots, u_k$. A new stage begins when the control of the algorithm goes (back) to step 1. Each stage u_i, for $i = 1, \ldots, k-1$, is associated with a segment (of examined positions) s_i of string S of size at least $m^{1/2}$. Last segment can be shorter, but it does not matter, since due to its size the last segment can be processed with any (e.g. KMP) linear time algorithm.

Each stage u_i, for $i = 1, \ldots, k-1$ starts with an execution of BG procedure detecting first occurrence of prefix π. Let $s_i = S[l, \ldots, l + |s_i| - 1]$ and y be the position in S aligned with the last symbol of first occurrence of π in s_i. The cost of detecting π is bounded by $y - l + O(\frac{(y-l)\log m}{m^{1/2}})$. In the remaining part of the stage KMP algorithm is applied where either symbols are recognized or there is a shift of length $\geq m^{1/4}$ (non-periodicity assumption) in case of a mismatch. Thus the number of comparisons associated with recognized symbols in the KMP part of stage u_i is $|s_i| - (y - l)$ and the number of comparisons associated with mismatches is $\frac{|s_i|-(y-l)}{m^{1/4}}$. Thus in total the number of all comparisons in stage u_i is not greater than $|s_i| + \frac{|s_i|}{m^{1/4}}$.

There are two types of overlaps between two neighboring segments s_i and s_{i+1}:

- if $|s_i| \geq m^{3/4}$ an overlap can be of size $m^{1/2}$ (due to missing entry in failure table). There are $O(m^{1/4})$ segments (overlaps) of this type.
- if $m^{1/2} \leq |s_i| < m^{3/4}$ an overlap is of size $< m^{1/4}$ (precise value in failure table is used). There are $O(m^{1/2})$ segments of this type.

The sum of the lengths of all segments $\Sigma_{i=1}^{k} s_i$ is bounded by $m + O(m^{1/4}) \times m^{1/2} + O(m^{1/2}) \times m^{1/4} = m + O(m^{3/4})$, which is the sum of the length of string S and sums of overlaps of type 1 and 2 respectively.

It follows that the total number of comparisons is bounded by

$$\sum_{i=1}^{k} |s_i|(1 + \frac{1}{m^{1/4}}) = (m + O(m^{3/4})) \times (1 + \frac{1}{m^{1/4}}) = m + O(m^{3/4}).$$

Periodic Case.

(1) Prefix of S of size $m^{1/4}$ and prefix π have different periods p, q respectively. Let prefix π' be the longest prefix of S having period p. Steps 1, 2(a), and 2(b) remain the same. In step 2(b) the shift performed by the algorithm may lead to prefix π'', s.t. $m^{1/4} \leq |\pi''| \leq |\pi'|$. In this case further shifts could be of size $p < m^{1/4}$ and the number of mismatches wouldn't be amortized. When this happens the algorithm continues trying to extend periodicity p as far as it is possible (updating failure table F). Assume that string u is found, s.t. $per(u[1, \ldots, |u| - 1]) = p$ and $u[|u|] \neq u[|u| - p]$. If $S[1, \ldots, |\pi'| + 1]$ is a suffix of u (i.e. $|u| - |S[1, \ldots, |\pi'| + 1]|$ is positive and divisible by p and $u[|u|] = S[|\pi'| + 1]$) then continue step 2 with non-periodic prefix u, otherwise do shift of size $|u| - p$ and go to step 1 with empty prefix.

(2) Prefix of S of size $m^{1/4}$ and prefix π have the same period p, s.t. $p < m^{1/4}$ and let π' be the longest prefix of S with period p. In this case the algorithm enters periodic case directly after step 1 and it performs exactly the same as in case 1 with $\pi'' = \pi$. Please note that if $\pi' > m^{3/4}$ preprocessing stage (determining π') can use more than $O(m^{3/4})$ comparisons. However processing of π' is done with no more than $|\pi'| + O(m^{1/4})$ comparisons and since π' is a prefix of S the amortization argument works.

4 Further Comments

Our deterministic algorithm can be modified to compute all periods of a string of length m with the same worst-case complexity $m + O(m^{3/4})$. Moreover, if we use our algorithm to process a pattern of size m in string matching problem, information obtained will allow to search any text of size n in time $n + O(\frac{n}{m^{1/4}})$. The only known lower bound for computing the period of a string of length m in the worst-case scenario is $m - 1$, and it holds when all characters in the strings are identical. Here we have delivered upper bound of size $m + O(m^{3/4})$, however we do believe that the worst-case complexity of the problem is $m + \Theta(m^{1/2})$. We also claim that the average-case complexity is $\Theta\left(\sqrt{m \cdot \log_{|\Sigma|} m}\right)$, and we leave better estimation of the lower bound as an open problem.

References

1. A. V. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity*, chapter 5, pages 257–300. Elsevier Science, Amsterdam, 1990.
2. A. Amir and G. Benson. Two-dimensional periodicity in rectangular arrays. *SIAM Journal on Computing*, 27(1):90–106, February 1998.
3. A. Amir, G. Benson, and M. Farach. Alphabet independent two dimensional matching. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 59–68, Victoria, British Columbia, Canada, May 4–6, 1992. ACM Press, New York, NY.
4. A. Amir, G. Benson, and M. Farach. An alphabet independent approach to two-dimensional pattern matching. *SIAM Journal on Computing*, 23(2):313–323, April 1994.

5. A. Apostolico and M. Crochemore. Optimal canonization of all substrings of a string. *Information and Computation*, 95(1):76–95, November 1991
6. A. Apostolico and Z. Galil, editors. *Pattern Matching Algorithms*. Oxford University Press, New York, NY, 1997.
7. A. Apostolico and R. Giancarlo. The Boyer-Moore-Galil string searching strategies revisited. *SIAM Journal on Computing*, 15(1):98–105, February 1986.
8. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1977.
9. D. Breslauer, L. Colussi, and L. Toniolo. Tight comparison bounds for the string prefix-matching problem. *Information Processing Letters*, 47(1):51–57, August 1993.
10. D. Breslauer, L. Colussi, and L. Toniolo. On the comparison complexity of the string prefix-matching problem. *Journal of Algorithms*, 29(1):18–67, October 1998.
11. D. Breslauer and Z. Galil. An optimal $O(\log \log n)$ time parallel string matching algorithm. *SIAM Journal on Computing*, 19(6):1051–1058, December 1990.
12. D. Breslauer and Z. Galil. A lower bound for parallel string matching. *SIAM Journal on Computing*, 21(5):856–862, October 1992. A preliminary version appears in *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 439–443, New Orleans, Louisiana, May 6–8, 1991. ACM Press, New York, NY.
13. D. Breslauer and Z. Galil. Efficient comparison based string matching. *Journal of Complexity*, 9(3):339–365, September 1993.
14. R. Cole. Tight bounds on the complexity of the Boyer-Moore string matching algorithm. *SIAM Journal on Computing*, 23(5):1075–1091, October 1994.
15. R. Cole, M. Crochemore, Z. Galil, L. Gąsieniec, R. Hariharan, S. Muthukrishnan, K. Park, and W. Rytter. Optimally fast parallel algorithms for preprocessing and pattern matching in one and two dimensions. In *Proceedings of the 34th IEEE Symposium on Foundations of Computer Science*, pages 248–258, Palo Alto, CA, November 3–5, 1993. IEEE Computer Society Press, Los Alamitos, CA.
16. R. Cole and R. Hariharan. Tighter upper bounds on the exact complexity of string matching. *SIAM Journal on Computing*, 26(3):803–856, June 1997.
17. R. Cole, R. Hariharan, M. Paterson, and U. Zwick. Tighter lower bounds on the exact complexity of string matching. *SIAM Journal on Computing*, 24(1):30–45, February 1995.
18. L. Colussi. Correctness and efficiency of pattern matching algorithms. *Information and Computation*, 95(2):225–251, December 1991.
19. L. Colussi, Z. Galil, and R. Giancarlo. On the exact complexity of string matching. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 135–144, Baltimore, MD, May 14–16, 1990. ACM Press, New York, NY.
20. M. Crochemore. Off-line serial exact string searching. In A. Apostolico and Z. Galil, editors, *Pattern Matching Algorithms*, chapter I, pages 1–53. Oxford University Press, New York, NY, 1997.
21. M. Crochemore. String-matching on ordered alphabets. *Theoretical Computer Science*, 92(1):33–47, January 1992
22. M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12(4/5):247–267, October 1994.
23. M. Crochemore, Z. Galil, L. Gąsieniec, K. Park, and W. Rytter. Constant-time randomized parallel string matching. *SIAM Journal on Computing*, 26(4):950–960, August 1997.
24. M. Crochemore, L. Gąsieniec, W. Plandowski, and W. Rytter. Two-dimensional pattern matching in linear time and small space. In *Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science*, volume 900 of *Lecture Notes in Computer Science*, pages 181–192, Munich, Germany, March 2–4, 1995. Springer-Verlag, Berlin.

25. M. Crochemore and D. Perrin. Two-way string-matching. *Journal of the ACM*, 38(3):651–675, July 1991.
26. M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, New York, NY, 1994.
27. A. Czumaj, Z. Galil, L. Gąsieniec, K. Park, and W. Plandowski. Work-time-optimal parallel algorithms for string problems. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 713–722, Las Vegas, NV, May 29 – June 1, 1995. ACM Press, New York, NY.
28. Z. Galil. Real-time algorithms for string-matching and palindrome recognition. In *Proceedings of the 8th Annual ACM Symposium on Theory of Computing*, pages 161–173, Hershey, PA, May 3–5, 1976. ACM Press, New York, NY.
29. Z. Galil. On improving the worst case running time of the Boyer-Moore string matching algorithm. *Communications of the ACM*, 22(9):505–508, September 1979.
30. Z. Galil. Optimal parallel algorithms for string matching. *Information and Computation*, 67(1–3):144–157, 1985.
31. Z. Galil and R. Giancarlo. On the exact complexity of string matching: Lower bounds. *SIAM Journal on Computing*, 20(6):1008–1020, December 1991.
32. Z. Galil and K. Park. Alphabet-independent two-dimensional witness computation. *SIAM Journal on Computing*, 25(5):907–935, October 1996.
33. Z. Galil and J. Seiferas. Time-space-optimal string matching. *Journal of Computer and System Sciences*, 26(3):280–294, June 1983.
34. L. Gąsieniec, W. Plandowski, and W. Rytter. Constant-space string matching with smaller number of comparisons: Sequential sampling. In Z. Galil and E. Ukkonen, editors, *Proceedings of the 6th Symposium on Combinatorial Pattern Matching*, volume 937 of *Lecture Notes in Computer Science*, pages 78–89, Espoo, Finland, July 5–7, 1995. Springer-Verlag, Berlin.
35. L. Gąsieniec, W. Plandowski, and W. Rytter. The zooming method: a recursive approach to time-space efficient string-matching. *Theoretical Computer Science*, 147(1-2):19–30, August 1995
36. D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, New York, NY, 1997.
37. D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, June 1977.
38. M. Lothaire. *Combinatorics on Words*. Addison-Wesley, Reading, MA, 1983.
39. M. Lothaire. *Algebraic Combinatorics on Words*. Addison-Wesley, Reading, MA, 1999.
40. R.C Lyndon and M.P. Schutzenberger. The equation $a^m = b^n c^p$ in a free group, *Michigan Math. J.*, $9 : 289 - -298$, 1962.
41. R. L. Rivest. On the worst case behavior of string searching algorithms. *SIAM Journal on Computing*, 6(4):669–674, December 1977.
42. W. Rytter. A correct preprocessing algorithm for Boyer-Moore string-searching. *SIAM Journal on Computing*, 9(3):509–512, August 1980.
43. U. Vishkin. Optimal parallel pattern matching in strings. *Information and Computation*, 67(1–3):91–113, October 1985.
44. U. Vishkin. Deterministic sampling - a new technique for fast pattern matching. *SIAM Journal on Computing*, 20(1):22–40, February 1991.
45. A. C. Yao. The complexity of pattern matching for a random string. *SIAM Journal on Computing*, 8(3):368–387, August 1979.

Author Index