Contents

0 Explanatory Remarks

It seems to me that it may be the appropriate time to submit my book, with tentative title Efficient Graph Representations, to a publisher. It is not completely polished at this point, but to polish it up before getting comments from referees which might change substantial sections of the book seems a bit misguided. The final version of this book may be individually written, or jointly written with Ross McConnell.

The book is intended to be a monograph, for researchers and advanced graduate courses. I view the following parts of the book as the most important contributions.

1) I feel that studying representation issues on graphs is very natural, and has not been studied in and of itself before. I have included a large number of open problems in the field, which I hope will stimulate research.
2) The issues raised in the optimization chapter, refining the notions of what it means to solve a problem on a class of graphs, have struck some people I have talked to as radical ideas, but I think that they are completely correct, natural, and important.
3) The recognition chapter gives a much more current view of important algorithmic developments in the field of intersection graph classes than is currently available (the current standard is Golumbic's book from 1980).
4) Some of the individual classes of graphs are important, and not covered adequately in any current text.

A number of chapters include material which has not been published in any form before. Of particular interest in this context is the chapter on matrices, and the section on induced visibility graphs. Much of the material in the chapter on intersection of graph classes is also new, but the results there are much more partial and is included primarily to introduce new open problems.

I have left out many topics that are at least somewhat relevant. For example, the work started by Galparin and Wigderson relating time complexity of very simple problems to graph classes with extremely efficient representations could be considered for inclusion. I may feel particularly guilty about this omission since it was my introduction to thinking about graph representation size, but have currently chosen not to include it since the techniques used are quite different from those stressed in the rest of the book. Comments on including this and other work for inclusion in the book would be most welcome.

I should add one other general comment on an issue that could disturb some readers. There is a certain type of proof which is important to include in a journal article, but which I think should not be included in a text. For example, suppose that we show that if x is adjacent to y in a graph of a certain type, we can always find a forbidden induced subgraph, and this can be proved by simply examining a large number of cases. It is important to know that this result is correct, which is why a proof must be verified in a journal. However, the reader only needs to know that x and y are nonadjacent, and if simply stating this as a fact clarifies the description of an algorithm, I prefer to give the result without a proof. Thus, in many sections, proofs are left to the journal and omitted from this book, and I do not plan to include the proof in a later version of the book. Of course, I may have omitted (or inadvertently included) individual proofs in a fashion that is not consistent with the rest of the book, and comments on including or excluding other proofs are welcome.

1 Introduction

Graph theory has developed into a very broad field since its introduction by Euler in the 18th century. Graphs are an intuitively pleasing and flexible method for dealing with relationships between objects, and are used in so many areas of mathematical research that they need no justification for study in their own right. This book is written from both a mathematical and computer science perspective. In the mathematical tradition, graphs and classes of graphs are studied with no specific applications in mind, although many of the classes discussed arose from important applications. Since my research area is graph algorithms, I place much greater emphasis on algorithms for dealing with problems than is found in most mathematical texts. Hopefully, the reader will agree that synthesizing the two traditions opens a number of interesting new research areas, and new open problems can be found throughout this book.

I have always been interested in representations of graphs. There are at least two natural areas of research which spring to mind if one talks of graph representation. One approach is to think of how to make a graph, typically described by a set of connections between points, into a picture which is easily understood by a human. This area of research is called graph drawing. The field of graph drawing has become very popular recently, and while I enjoy reading books on this area such as [54], I am not the person to write such a book. As well as not being my research area, anyone who has seen my office or choice of clothing would agree that I am not qualified to work in an area in which one must decide whether the result is aesthetically pleasing!

The subject of this book can be viewed as the reverse process of graph drawing. In graph drawing, we imagine a representation of a graph is given which is easy for a machine to work with, and we want to convert this into a representation which can be viewed easily by a human. This book talks about methods for storing graphs which make them easy for the computer to work with. I choose to call this topic efficient graph representation.

The closest generally accepted area of research to the work here is the field of intersection graphs. Golumbic's book [79] has had a major effect on both the study of intersection graphs, and my own research. In general, I have tried to avoid going deeply into subjects which are well covered by Golumbic. Other good books which are related or have affected my own work include [167], [176], [65], [121].

1.1 Graph Theory Background and Terminology

A graph $G = (V,E)$, where V is a set of vertices, and E is a set of edges. Each edge corresponds to a pair (x,y) of vertices. If the edges are ordered pairs of vertices, we call G a digraph.

In this book, we will not in general deal with multigraphs or loops; that is, there at most one edge (x,y) in a graph, and there is no edge (x,x). Note that if G is a digraph, there may be both an edge (x,y) and an edge (y,x).

Two vertices x,y of a graph are adjacent if they are connected by an edge. The neighbors of x, sometimes denoted N(x), are the set y such that $(x,y) \epsilon E$.

The subgraph of $G = (V,E)$ induced by a set S of vertices is the graph $H = (S,E_H)$, where E_H is the set of edges of E such that both endpoints are in H.

We will refer to the number of vertices in a graph as n, and the number of edges as m. We will assume that vertices of a graph are labeled as integers from 1 through n, unless noted otherwise.

If C is a class of graphs, the class of co-C graphs is defined to be those graphs G such that the complement $\overline{G}$ of G is in C The complement $\overline{G}$ of G has the same vertex set as G, and x is adjacent to y in $\overline{G}$ if and only if x and y are nonadjacent in G.

1.2 Algorithm Background and Terminology

This book is intended to be of interest to both mathematicians and computer scientists. In most cases, algorithmic topics will be presented separately from other representation issues, but there is one notational device from the world of computer science which is absolutely essential. We will rely heavily on order notation, as described below.

Order notation is a form of mathematical notation devised to ignore constant factors, and preserve larger differences between functions. We will use the following pieces of order notation: O, o, Ω, Θ, which respectively mean $\leq$, $<$, $\geq$, and $=$, if constant factors are ignored. We give formal definitions below. Although there are slight variants of definition in the literature, and I am not using the most standard definitions, there are no functions used in this book where the different definitions make a difference. I use this form of definition so that the reader who is unfamiliar with order notation can understand all other definitions from the formal definition of O.

Formally, f(n) is O(g(n))if there are constants c and N such that for all $n > N$, f(n) $\leq c$g(n). We will say f(n) is Θ(g(n)) if f(n) is O(g(n)) and g(n) is O(f(n)). f(n) is o(g(n)) if f(n) is O(g(n)), but g(n) is not O(f(n)). Finally, we use f(n) is Ω(g(n)) to denote that g(n) is O(f(n)).

Measures of time complexity in graph algorithms are based on the number of vertices and the number of edges in the input graph; these will be denoted as n and m respectively. For graphs, we use the term linear time to mean O(n+m) time; the term generally means time proportional to the size of the input.

Some readers might notice that while we present upper bounds in the form of algorithms with certain time complexities for solving problems, we never present lower bounds which say that the problem requires a certain amount of time. There is a good reason for this; we know next to nothing about proving nonlinear lower bounds for practical problems, unless we place restrictions on the type of operation which is allowed. In my opinion, this is one of the great open problems in computer science, and may be at the core of why it is so hard to prove that P is not equal to NP. I do not know of any polynomial time solvable problem for which a lower bound is known larger than the size of the input and/or the size of the output, if all operations are allowed on the data. Thus, any nonlinear algorithm in the book is theoretically a candidate for improvement. Our only 'lower bounds' on polynomially solvable problems will be pointing out when an improvement might be difficult to find, since it would improve the time complexity of a problem such as matching which has been studied extensively.

Our other notion of a 'lower bound' is NP-completeness. Readers who want more information on the subject of NP-completeness are advised to study [71]. A decision (yes/no) problem is in P if it can be solved in polynomial time. A decision problem is in NP if it can be solved in polynomial time by a nondeterministic Turing machine; for such problems, you can give a polynomial length proof for any yes instance of the problem. A problem is NP-complete if the problem is in NP, and all problems in NP can be polynomially transformed into the problem. If a problem is NP-complete, we can at least say that finding a polynomial time solution for the problem would require a magnificent breakthrough in research.

Important NP-complete problems on graphs include the clique problem, the chromatic number (coloring) problem, independent set, Hamilton circuit; many others can be found in [71]. We will define the problems which come up repeatedly in the book at this point; many of these problems are tractable for classes of graphs studied in the book. Other optimization problems will be defined when they arise.

A clique C in a graph is a set of vertices such that each pair of vertices in C is connected by an edge. The clique problem takes a graph G and an integer k, and asks whether G contains a clique of size k. The maximum clique problem involves finding the largest clique in the graph. A clique C is maximal if no vertex can be added to C such that the resulting set is still a clique.

An independent set (also called a stable set) is the complement of a clique; that is, every pair of vertices in the set must be nonadjacent. Maximum and maximal independent sets are defined analogously to the definitions above.

A coloring of a graph is an assignment of numbers (called colors) to vertices such that no pair of adjacent vertices is assigned the same color. The chromatic number of G is the minimum number of colors used in any valid coloring of G. The coloring, or chromatic number problem, takes a graph G and an integer k and asks whether G can be colored with at most k colors.

We now must deal with one issue that can cause confusion, known as the unit-cost assumption. In this assumption, standard operations such as addition, multiplication, and finding the element at position i in an array are assumed to take constant time if the input numbers are of size $O(n^k)$, that is, the number of bits in the input is $O(\log n)$. I will not discuss the reasoning behind the unit-cost assumption; if we used different assumptions, our time bounds for standard algorithms would differ from those used in the literature.

The reason that we bring up the unit-cost assumption here is that we will *not* be using the unit cost assumption with respect to space complexity. When we discuss a storage mechanism for a class of graphs, we will be counting the number of bits in the representation rather than the number of integers in the range $1..O(n)$. We must count bits of space, since we are discussing what the limits are in a compact representation of a graph. We will, however, still measure space complexity in terms of order notation, since this makes our analysis much cleaner and clarifies a number of open problems.

1.3 Representation Background

As a researcher in computer representation of graphs, the treatment in some introductory discrete mathematics and data structures texts is particularly irritating. Most texts introduce the following two forms of graph representation, which we will also use in this book.

The adjacency matrix A of a graph G is a matrix with n rows and n columns, such that $A[x,y] = 1$ if and only if there is an edge from x to y. Adjacency matrices may represent either directed graphs or undirected graphs.

Adjacency lists store a graph by keeping a list of the neighbors of each vertex. Generally, one stores these as an array of lists, so that one can have access to the list of neighbors of i in constant time. A graph is stored by storing the adjacency lists of all vertices. In standard terminology, an adjacency list keeps an unordered list of neighbors of each vertex; the lists can be sorted in linear time if desired (exercise 1).

These definitions are presented adequately in most texts. However, the explanation that goes with them is often faulty. Combining misstatements from several texts, we get the following explanation.

Straw Man's comments on graph representations: There are two methods for representing a graph in a computer; adjacency matrices, and adjacency lists. It is faster to work with adjacency matrices, but they use more space then adjacency lists, so you will choose one or the other depending on which resource is more important to you.

Of course, there are many interesting forms of graph representation; that is one of the points of this book! The notion of time/space tradeoff between the two forms of representation given is also very misleading, since there are problems which can be provably solved faster on adjacency lists than adjacency matrices, and the amount of space used to store a random graph is larger with adjacency lists than with adjacency matrices. Let us look at the time/space tradeoff a bit more carefully.

If all information about the input graph comes from queries of the form 'is there an edge between x and y?', then adjacency matrices are clearly faster, since a query can be answered in constant time, whereas this can involve traversing a long list if adjacency lists are used. However, many algorithms use operations such as 'find a neighbor of x' or 'get the next neighbor of x'. These operations take constant time if an adjacency list is used, but can take time proportional to n if you are only given an adjacency matrix as input. Thus, most linear time solvable problems on graphs, such as asking whether an input graph is bipartite, connected, acyclic, ..., require $\Omega(n^2)$ time if input is in adjacency matrix form (exercise 4).

We mention in passing a well known theorem about time required to test properties if the input is in adjacency matrix form. A graph property is nontrivial if the answer may be either yes or no, and monotone if adding edges to a graph which has the property always produces a graph which has the property. Thus, for example, connectivity is a monotone property, while properties such as being disconnected or acyclic are not. Any nontrivial monotone property requires $\Omega(n^2)$ time to test if the input is given in adjacency matrix form [136].

Before dealing with space complexity of the representations, we should mention a well known (to people in the field) trick in graph representation which allows some problems to be solved in linear time. To the best of my knowledge, this approach is first used in problem 2.12 of [10]. One might think that any algorithm which uses adjacency matrices requires $\Omega(n^2)$ time, since the matrix takes that much time to initialize. Exercise 2 shows that it is possible to use the adjacency matrix effectively to test adjacency in constant time without paying initialization costs. Exercize 3 gives a good example of the use of this representation; it is easy to develop a linear time algorithm to recognize the class of graphs if you know the representation, but if you do not know the representation it is quite difficult. In fact, several journal papers and theses have been written recognizing the class in linear time; these algorithms manage to solve the problem using $O(m+n)$ space, but it is quite tricky to do so!

Now let us address space complexity of adjacency matrices and adjacency lists. Adjacency matrices use the same amount of space for every graph. Although there are slight variants which are useful for some applications, such as storing only elements above the diagonal for undirected graphs, there are clearly $\Theta(n^2)$ entries in the matrix, and each entry is a single bit, so $\Theta(n^2)$ bits are used to store the graph.

Adjacency lists will have a total of $2m$ entries for an undirected graph, since each edge appears on two lists. Each entry holds a number in the range $1..n$, and thus takes $\log n$ bits, making the total space complexity $\Theta(m \log n)$ bits.

The number of graphs on n vertices is $2^{n(n-1)/2}$, since there are $n(n\text{-}1)/2$ possible edges and we can choose to include or exclude each edge. Thus, any form of representation must use at least $n(n\text{-}1)/2$ bits; this result holds for both the worst case and the average case. Thus, every form of representation must use $\Omega(n^2)$ bits, and adjacency matrices use $O(n^2)$ bits, which means that adjacency matrices are within a constant factor of the minimum possible amount of space. For the purposes of this book, if a set has $2^{\Theta(f(n))}$ elements, any representation of members of the set which uses $O(f(n))$ bits will be called space optimal; adjacency matrices are a space optimal representation of graphs. On the other hand, adjacency lists are not space optimal; they use $\Theta(n^2 \log n)$ bits on complete graphs, since every entry uses $\log n$ bits. In fact, if all of the $2^{n(n-1)/2}$ graphs on n vertices are considered equally likely, the average number of edges in a graph is $n(n\text{-}1)/4$, and adjacency lists use $\Theta(n^2 \log n)$ bits in the average case as well. Thus, while adjacency lists save space on sparse graphs, in our

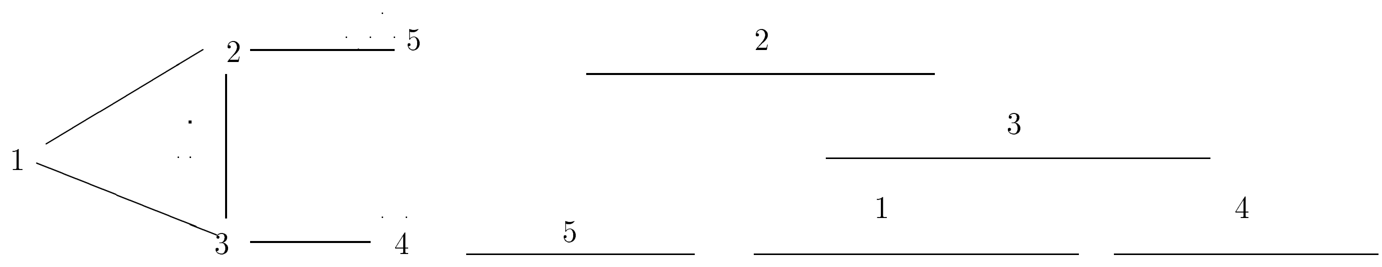


Figure 1: An interval graph and its model

formal sense they use more space than adjacency matrices.

This book examines many aspects of representations of graphs. There are many possible qualities we might look for in a representation, such as space optimality, speed for adjacency testing, ease of finding the representation, help with solving optimization problems, correct modeling of a type of problem, and others. We will formalize some of these notions, and examine aspects of graph representation for many graph classes which have been studied in the literature.

McConnell [] has introduced the notion of a 'partially complemented' representation of a graph. In this form of representation, there is a list associated with each vertex, and a special bit which indicates whether this list is of the neighbors of the vertex or the nonneighbors of the vertex. He uses this to show that many operations on a graph can be solved in time which is linear in the partially complemented representation. For example, the fact that you can perform depth first search and breadth first search in linear time with respect to the partially complemented representation means that you can determine whether the complement of a graph is connected in $O(m+n)$ time. Similarly, one can tell whether the complement is chordal, interval, and many other properties in linear time using this strategy. There is no problem which can be done in linear time in the size of the adjacency lists, and is known to require superlinear time on the partially complemented lists; here, linear time means in the size of both the input and the output (so that marking all edges of G is not considered to be superlinear with respect to the partially complemented list).

1.4 Example of a Nice Representation

We introduce a well known class of graphs called interval graphs , which have played a key role in defining this field. Interval graphs are particularly satisfying in that they have an important real application, nice (but not obvious) mathematical characterizations, and are defined by a simple geometric rule. For a more complete look at all aspects of interval graphs, see [65].

In an interval graph, vertices correspond to intervals on the real line, and two vertices are adjacent if and only if the corresponding intervals have a nonempty intersection. An example of a graph and its interval model is given in Figure 1.

Benzer's Nobel prize winning work on arrangement of genes in the chromosome is directly related to interval graphs; for more on the application, see [13]. He speculated that genes were arranged in a simple linear fashion. Mutations would correspond to a damaged segment of the chromosome; this damaged segment would be an interval on the chromosome. Each mutation could damage a different set of genes. At that time, the only information that could be gathered was the set of deformities caused by a mutation. A graph can be formed from this data, by making each mutation into a vertex and adding an edge between two vertices if the mutations share a common deformity. If Benzer was correct in his supposition that genes were arranged in linear fashion, this graph will be an interval graph. He found that the graphs were interval graphs, and this was considered to be important evidence supporting his theory.

This led to a great deal of work on interval graphs and variants of interval graphs, some of which we will look at in chapter 3. For now, let us think of how we would represent an interval graph on the computer. Clearly, we could use adjacency matrices or adjacency lists. This does not seem to capture any special properties of interval graphs. For example, since a clique of size n is an interval graph, an adjacency list representation will not be space efficient. However, the interval model itself gives us a natural form of representation.

Label the endpoints of intervals in increasing order of their appearance on the line. This assigns to every vertex two integers in the range $1..2n$. For each vertex v, we store the labels of the endpoints of v; this uses $\mathrm{O}(\log n)$ bits per vertex. Suppose that we want to test whether x and y are adjacent. It is easy to determine this simply from the labels stored at x and y. Thus, we can completely reconstruct the graph from the $\mathrm{O}(\log n)$ bits at each vertex, and interval graphs can be stored in $\mathrm{O}(n\log n)$ space. Testing whether x and y are adjacent also takes constant time with this form of representation, so we have managed to combine a space efficient representation and fast adjacency testing for this class of graphs.

This raises a number of other problems, which are discussed in various chapters. Suppose that we are given an interval graph, but are not given the interval representation (as in, for example, Benzer's problem); can we find this representation efficiently? It is natural to think of letting vertices correspond to other geometric objects; how does this change the answer above? What can we do efficiently on this special class of graphs? How else can we represent a graph class with a small amount of space, and still test adjacency efficiently? This book looks both at the formalizations of these questions, and specific classes of graphs with interesting forms of representation.

Specifically, we try to provide a formal model of efficient representation in chapter 2, and make the questions we have about a class more precise. Chapter 3 surveys some of the well known classes of graphs defined by variants on the interval model above. Chapter 4 introduces an algorithmic technique which is useful for dealing with many of these graph classes. Chapters 5-12 examine classes of graphs which pose especially interesting problems with respect to finding good representations. Separate chapters are devoted to the recognition problem (chapter 13) on a large number of these graph classes, and a new view of optimization problems which comes from focusing on issues of representation (chapter 14).

1.5 Perfect Graphs

One class of graphs which will not be studied in this book, but is relevant to the study of some optimization problems on these classes, is the class of perfect graphs. Readers interested in studying perfect graphs are advised to consult [79].

A graph G is perfect if for every induced subgraph H of G, the chromatic number of H is equal to the maximum clique size of H. There are many fascinating facts and open problems on perfect graphs. Among these are the following.

Perfect Graph Theorem [115]: G is perfect if and only if the complement of G is perfect.

Strong Perfect Graph Conjecture [15]: G is perfect if and only if neither G nor its complement contains any induced cycles of length $2k+1$ for $k > 1$.

Induced cycles of length > 3 will also be called holes. The strong perfect graph conjecture is still open. However, even if it is true, we still would not know whether perfect graphs can be recognized efficiently, since we do not know how to recognize 'odd holes' in polynomial time. Thus, the following problems are also open.

Perfect Graph Recognition: Is there a polynomial time algorithm for recognizing perfect graphs?

We note that the problem is in co-NP as a consequence of results in [29] and [84]; see also exercise 8.

Odd Hole Recognition: Is there a polynomial time algorithm to determine whether a graph has an odd hole?

The following beautiful result due to Grötschel, Lovász, and Schrijver will be relevant to us, since perfect graphs contain a good number of the classes discussed in this book.

Perfect Graph Optimization [84]: The maximum clique, maximum independent set, chromatic number, and clique cover number problems can be solved in polynomial time for perfect graphs.

Since perfect graphs cannot at this time be recognized efficiently, it is interesting to ask what this algorithm does if the input graph is not perfect. If the input graph G is not perfect, the algorithm may either provide evidence of the imperfection of G, or solve the optimization problem correctly on G. Thus, the result of the algorithm is reliable, whether or not G is perfect. Algorithms with this property for other classes are studied in chapter 14. We note that although the optimization algorithms for perfect graphs are polynomial, the time bounds are unusually large, so that we will study optimization algorithms for subclasses of perfect graphs as well.

1.6 Exercises

1) Normally, adjacency lists are viewed as unsorted, i.e. the neighbors of vertex x may occur in any order on x's list. Suppose we want to arrange the lists so that neighbors of each vertex occur in ascending order on its adjacency list. Show that all adjacency lists of G can be sorted in $O(n+m)$ time.

2) Suppose that you are given the set of edges in a graph. Show that you can use $O(n^2)$ space (for this problem only, a number in the range $1..n^2$ is considered to take constant space), and $O(n+m)$ time initialization so adjacency testing is $O(1)$. Note that your problem is to distinguish whether the entry you look up is a valid entry of the matrix, or garbage left in the matrix since you did not clean it out during initialization. This is a challenging problem without a hint; if you need a hint, see problem 2.10 of [10].

3) A series reduction in a graph takes a degree two vertex y with neighbors x and z, eliminates y, and adds an edge from x to z, if that edge does not already exist. A graph is series-parallel if all edges can be eliminated through a sequence of series reductions. Design a linear time algorithm to test whether a graph is series-parallel. Note: the name comes from the fact that this is usually defined as a class of multigraphs which can be made edgeless using a sequence of series and parallel reductions; a series reduction would add the edge (x,z) whether or not it exists already, and a parallel reduction eliminates one of two edges between the same pair of vertices. For a relation between series-parallel graphs and cographs studied in chapter 9, see [172].

4) Show that if the input is given in adjacency matrix form, determining whether G is connected takes $\Omega(n^2)$ time.

5) A vertex v is a supersink in a directed graph if v has in-edges from all other vertices, and no out-edges. Show that we can determine whether G has a supersink from its adjacency matrix in $O(n)$ time; note that this is not a monotone property.

6) Show that almost all graphs are not interval graphs.

7) Show that interval graphs cannot contain holes of size greater than 3 .

8) A graph on n vertices is partitionable if $n = \alpha\omega + 1$, and for every vertex x of G, the largest independent set on G-x = the clique cover number of G-x and the largest clique in G-x = the chromatic number of G-x. It can be shown that every minimal imperfect graph is partitionable, and every partitionable graph is imperfect [79]. Use these facts to show that perfect graph recognition is in co-NP. Note: recognizing partitionable graphs in polynomial time is an open problem.

9) Show that if G is perfect, then either G or its complement has a clique of size at least $n^{.5}$. Note: It would be interesting to develop fast algorithms for finding a clique or independent set of this size very efficiently. One variant of this problem, which corresponds to restricting the graphs to be permutation graphs, is to find an increasing or decreasing subsequence of length $\Omega(n^{.5})$ in a permutation in $O(n)$ time; this can be solved in $O(n\log\log n)$ time by using algorithms for finding the largest independent set in a permutation graph when the model is given.

10) Show that there are triangle-free graphs with arbitrarily large chromatic number.

11a) Show how to perform depth first search on a partially complemented list in linear time.

11b) Show how to perform breadth first search on a partially complemented list in linear time.

12) The following problem was posed to me by Andreas Stathopolous. Suppose that, to save space, an undirected graph is stored in the following form. For every edge (x, y), instead of storing x on the adjacency list of y and y on the adjacency list of x, you store only one of these two vertices on the adjacency list of the other. Show that it is possible to determine whether G is connected in $\mathrm{O}(n+m)$ time given input of this form, where you are only allowed to use $\mathrm{O}(n)$ extra space (this is $\mathrm{O}(n \log n)$ bits) and are not allowed to modify the space used for storing the input.

1: (4,7)
2: (2,6)
3: (5,9)
4: (8,10)
5: (1,3)

Figure 2: Efficient Representation of the Graph in Figure 1

2 Implicit Representation

This chapter introduces a form of graph representation which is quite general, and includes most of the representation schemes for graphs which are easy to work with. The ideas in this chapter were discovered independently by Muller [129] and Kannan, Naor and Rudich [104].

The interval graphs discussed in the previous section are a good starting point for understanding the appeal of implicit representation. The most obvious characteristics we want from a computer representation of a graph are that we use a small amount of space, and are able to test adjacency quickly. An interval representation can be transformed easily to meet these criteria. Endpoints of intervals can be given integer values in the range $1..2n$. Each interval is then defined by a pair of endpoints; we can store the graph as a sequence of pairs (e1,e2) as in Figure 2. This representation uses O(log$n$) space per vertex, and thus O($n$log$n$) space overall. Note that since any clique is an interval graph, standard adjacency matrices or adjacency lists would use $\Omega(n^2)$ space to store an interval graph on n vertices. It is also easy to test whether two vertices are adjacent using this representation in constant time. Intervals (i,j) and (k,l) are adjacent if and only if i or j is in the range k..l, or k or l is in the range i..j. Thus, we use O($n$log$n$) space to store the graph, and can test adjacency in constant time.

If we try to generalize the reason that adjacency testing of intervals is efficient, we come naturally to the notion of an implicit representation. The interval representation works because we are storing only a small number of small integers at each vertex, and adjacency between two vertices x and y can be tested using only the numbers stored at x and y. More formally, we make the following definition.

R is an implicit representation of a graph G if R assigns O(logn) bits to each vertex of G, and there is an adjacency testing algorithm which decides whether x and y are adjacent based only on the bits stored at x and y.

We note that this is still not a completely formal definition; we will delay a more technical definition of the adjacency testing mechanism until later in this chapter.

First, let us compare and contrast implicit representations with representations which are space optimal. It is easy to see (at least for the labeled case) that the number of interval graphs is $2^{\Omega(nlogn)}$; this implies that the implicit representation is space optimal. The unlabeled case is treated in exercise 1. Consider graphs in which each of the vertices in the set 1,...,$n/2$ is matched with exactly one vertex in the set

$n/2+1,..,n$. Each such graph is an interval graph, and there are $(n/2)!$ such graphs. Since
$(n/2)! = n/2*(n/2\text{-}1)*..*n/4*(n/4\text{-}1)*..*1 > n/2*(n/2\text{-}1)*..*(n/4+1)*n/4 > (n/4)^{n/4}$
$= 2^{((n/4)log(n/4)}$
which is $2^{\Omega(nlogn)}$, representing all interval graphs requires $\Omega(n\log n)$ bits. We achieve $O(n\log n)$ bits with the implicit representation, so the implicit representation is (asymptotically) space optimal.

However, it is easy to invent a generic form of space optimal representation which would not be a good mechanism for storing a graph. Suppose we know that there are f(n) labeled interval graphs on n vertices. This number could be determined (in theory) by running through all possible graphs on n vertices, and testing whether each one is an interval graph; this test can be performed in $O(n+m)$ time, as we will see in chapter 13. We can place an ordering on all graphs with n vertices; for example, graphs which contain the edge (1,2) come before graphs which do not contain (1,2), then divide each set of graphs depending on whether they contain (1,3), etcetera. An interval graph could be given the number i if there are exactly i-1 interval graphs on n vertices which come before this graph in the ordering. There are f(n) interval graphs, and each interval graph is given a number between 1 and f(n); this number has log(f(n)) bits, so it is clearly a space optimal representation. It is possible to test whether two vertices are adjacent using this representation, though the natural method is unbelievably slow, since it would mean constructing all possible graphs on n vertices, and testing whether each is an interval graph, before we could deduce that we have the graph that we are seeking and can answer questions about adjacency.

In an implicit representation, all information is distributed to the vertices themselves. We need to require adjacency to be tested using only information which is 'local' to the vertices in question, to avoid representations as in the previous paragraph, in which bits of the number representing the graph are distributed to the vertices, but reconstructing the graph would still involve using bits from all of the vertices. This property of local adjacency testing is what makes adjacency tests efficient for all known forms of local representation, and led Muller to call this form of graph representation a 'local structure' representation.

The first possible question regarding implicit representation is whether every graph class has an implicit representation. The answer to this must be no, since there are classes of graphs which have $2^{\Theta(n^2)}$ members. Any representation of graphs in such classes must use $\Omega(n^2)$ bits for the average graph in the class, while an implicit representation uses $O(n\log n)$ bits. Examples of such dense classes of graphs include the class of all graphs, bipartite graphs, triangle-free graphs, asteroidal-triple-free graphs, and many more.

It is also possible to show that there are graph classes with $2^{O(nlogn)}$ members which do not have implicit representations. Consider the class of graphs which contain at most $7n$ (7 could be replaced with any other constant) edges for all values of n. We will call these graphs 7-sparse. A standard adjacency list representation uses $O(n\log n)$ bits to represent any such graph, so there are $2^{O(nlogn)}$ graphs on n vertices in this class.

Suppose that the 7-sparse graphs have an implicit representation. Consider the set of all graphs on n vertices; we know that there are $2^{\Theta(n^2)}$ such graphs. Let G be an arbitrary graph on n vertices. Add n^2 isolated vertices to G, and we have a graph G' which is 7-sparse. G' must have an implicit representation. Adjacencies between vertices in an implicit representation can be determined using only the bits stored at the vertices, so G can be reconstructed from the bits stored at the vertices of G within the representation of G'. There are $\mathrm{O}(\log(n^2+n))$ bits stored at each vertex of G'; this implies that G can be reconstructed from a set of $\mathrm{O}(n\log n)$ bits. This implies that there are $2^{O(nlogn)}$ graphs on n vertices, which is certainly false, so there cannot be an implicit representation of the 7-sparse graphs.

The construction above is based on a particular property of 7-sparse graphs which some people find intuitively unsatisfying. We can take an arbitrary (non 7-sparse) graph, add vertices, and get a graph which is 7-sparse. If we think of 7-sparse graphs as a potentially 'easy' class of graphs, it seems strange that we can take a 'hard' graph, add vertices to it, and get an 'easy' graph. To formalize this concept, we need a new definition.

A class of graphs is hereditary if for every graph G in the class, every induced subgraph of G is also in the class. Interval graphs, planar graphs, and triangle-free graphs are hereditary classes of graphs; deleting vertices will not disturb the interval and planar representation, and deleting a vertex cannot make a triangle in a graph. Connected graphs and trees are not hereditary classes of graphs, but forests are a hereditary class.

Consider the problem of finding an implicit representation for uniformly c-sparse graphs, that is, graphs such that every subgraph on x vertices has no more than $cx\log n/\log x$ edges. Uniformly c-sparse graphs are contained in the c-sparse graphs, but one cannot use the construction above to embed an arbitrary graph into a uniformly c-sparse graph. In fact, a graph has an orientation in which every vertex has indegree at most k if and only if $|E(X)| \leq k|X|$ for every subset X (which gives a representation using k numbers per vertex); distinction from the general class of sparse graphs shows the importance of a property being hereditary.

Kannan, Naor, and Rudich made the following very interesting conjecture, which remains open.

Implicit Graph Conjecture: Every hereditary class of graphs which has $2^{O(nlogn)}$ graphs on n vertices has an implicit representation.

In the course of this book, we will introduce classes which are open with respect to the implicit graph conjecture, and explain the difficulties involved in representing each type of class.

Let us look at some classes which can be represented implicitly, for obvious and nonobvious reasons. An implicit representation of interval graphs follows very easily from the definition of interval graphs; the same holds true for some other well known classes of graphs which will not be covered extensively in this book, such as circular-arc graphs, permutation graphs, boxicity k graphs, dimension k posets, circle graphs, and many other intersection graphs mentioned in books by Golumbic [79] and Brandstadt, Le, and Spinrad [23].

Let us look at some other well known classes of graphs. It is obvious that there are $2^{O(nlogn)}$ trees on n vertices, since an adjacency list representation uses O($n$log$n$) bits. However, this does not immediately imply an implicit representation, since we cannot store all adjacencies of a vertex at that vertex, as vertices can have arbitrarily high degree. A simple implicit representation of a tree can be obtained by arbitrarily choosing a root for the tree, and storing only the parent of each vertex v at v. Two vertices v and w are adjacent if and only if v is the parent of w or w is the parent of v, so this storage technique gives an implicit representation.

Planar graphs are candidates for having an implicit representation, since it is well known that the number of edges in a planar graph is O(n). An implicit representation of planar graphs comes from the property that every planar graph has a vertex of degree at most 5. We can repeatedly remove vertices of degree at most 5 from the graph, storing all remaining neighbors of the vertex with the vertex. Given any pair of vertices x and y, x and y are adjacent if and only if y is stored at x or x is stored at y.

The planar graph representation above can be generalized further. Any hereditary class of graphs with at most cn edges has an implicit representation. Suppose a graph class is guaranteed to have at most cn edges for a graph with n vertices. The total size of all adjacency lists is $2cn$, which implies there must be some vertex with at most $2c$ neighbors. We remove this vertex, storing all neighbors. Since the class is hereditary, the remaining graph has at most $c(n\text{-}1)$ edges, the total size of adjacency lists is at most $2c(n\text{-}1)$, and there is again a vertex with at most $2c$ neighbors. Repeating this process until all vertices are removed gives an implicit representation with at most $2c$ integers stored at each vertex.

Of course, many classes of graphs which can have large number of edges, such as interval graphs, also have implicit representations. Not all implicit representations are obvious from the definition. Consider, for example, the class of distance hereditary, or completely separable graphs, first studied in [12].

Distance hereditary graphs are graphs in which all induced paths between pairs of vertices have the same length. It is not clear that the number of such graphs is $2^{O(nlogn)}$, so that an implicit representation is even possible. Hammer and Maffray showed [91] that distance hereditary graphs could be built from a single vertex by repeatedly adding either a 'pendant vertex' adjacent to exactly one old vertex, or a 'twin vertex' of another vertex; twins have the same adjacencies to the old vertices, and may or may not be adjacent to each other.

This characterization shows that the number of distance hereditary graphs is $2^{O(nlogn)}$; the graph can be reconstructed from the sequence, and each part of the sequence can be stored using O(log$n$) bits, since there is only 1 number in the range $1..n$ to store. However, it is not possible to determine whether x and y are adjacent given only information stored with x and y; for example, if x is a twin of a and y is a twin of b, we need to know whether a and b are adjacent. Since distance hereditary graphs are a hereditary class of graphs, we might wonder whether they contradict the implicit graph conjecture. However, the distance hereditary graphs are a subset of the circle graphs, which have an implicit representation as a consequence of their definition, so the distance hereditary graphs satisfy the implicit graph conjecture.

Before continuing with new material, we give one more example of a class of graphs which satisfies the implicit graph conjecture for nonobvious reasons, to show that implicit representations do not always come directly from the definition of a graph class. The intersection graphs of paths in a tree, which are discussed at greater length in chapter 6, are defined as follows. A graph G is in the class if there exists a tree T such that vertices of G can be mapped to paths in T, such that the 2 paths in T share a common vertex if and only if the corresponding vertices are adjacent. In exercise 7, we see that if there is such a tree, then there is a tree with $\mathrm{O}(n)$ vertices. This implies that the number of graphs in the class is $2^{O(nlogn)}$, since the graph can be reconstructed from an n vertex tree and the 2 endpoints corresponding to each vertex of the graph. However, this does not give an implicit representation, since global information (the tree) is necessary to determine adjacencies. We leave the construction of an implicit representation for this class of graphs to exercise 8.

Other ways to define graph classes which do not give immediate implicit representations include graph classes defined by forbidden subgraphs or by taking intersections of known classes of graphs. For example, if we forbid triangles as induced subgraphs, we still have $2^{\Theta(n^2)}$ graphs in the class, and cannot have an implicit representation, while if we forbid induced paths on 4 vertices there is an implicit representation, although this is not obvious from the definition. If we take well known classes of graphs with many members and define a new class by intersection, we may or may not get a class which is small enough to have an implicit representation. For example, both chordal graphs and comparability graphs (these classes will be defined later) contain $2^{\Theta(n^2)}$ members, but there are $2^{\Theta(n^2)}$ graphs which are both chordal and cochordal (such graphs are called split graphs), while there are only $2^{O(nlogn)}$ graphs which are both comparability and co-comparability graphs. The latter class is equivalent to a geometrically defined class of graphs called permutation graphs, and has an implicit representation, though this is not obvious when the class is defined by intersection of the two larger classes. Classes of graphs defined by forbidden subgraphs and intersection of larger classes are studied in chapters ? and ?.

2.1 Implicit Representation and Universal Graphs

Although relatively little is known about which classes of graphs have implicit representations, there is a connection between implicit classes of graphs and the older field of universal graphs with respect to a class. In order to clarify this connection, we need to give a more formal definition of an implicit representation.

In our earlier discussion, we stated the requirement of a test for determining whether 2 vertices are adjacent, using only bits stored at the 2 vertices. Formally, we require that there is a Turing machine M which takes 2 binary strings B1 and B2 as input, and produces either yes or no as output. We interpret this to mean that vertices with bit strings B1, B2 stored at x and y are adjacent iff M answers yes on input B1, B2.

Kannan, Naor, and Rudich (but not Muller) add the restriction that M is a polynomial time Turing machine. This has the desirable property of guaranteeing that an implicit representation is not only efficient with respect to space complexity, but that adjacency testing is also fast. Since there are O(log$n$) bits stored at x and y, and M is polynomial, the adjacency test is at worst polylogarithmic. When we generalize the notion of an implicit representation in the next section, polynomial time Turing machines may not necessarily give fast adjacency testing, so it is not clear whether to retain this restriction or whether we should allow M to be any recursive Turing machine. Since the issue does not arise on any known class of graphs, we will retain the polynomial restriction for our adjacency test, though it is not known whether this affects which classes of graphs are representable. Muller also considers the possibility of inputs which do not correspond to valid labels of vertices, and what the Turing machine can do on invalid pairs of inputs.

Let S be a set of graphs. G is vertex-induced universal if every graph in S is an induced subgraph of G. Let Cn be the set of graphs in the class C which contain at most n vertices. Kannan, Naor, and Rudich [104] showed that if a class C has an implicit representation, then for every n there is a vertex-induced universal graph for Cn which contains O(n^k) vertices. Muller [129] adds conditions so that a class has a variant of an implicit representation if and only if the universal graphs fulfill certain conditions.

The universal graph U can be created as follows. Each bit string corresponding to a vertex in an implicit representation of a graph in Cn contains O(log$n$) bits; that is, at most clog$n$ bits for some value of c. The vertices of U correspond to n copies of all possible bit strings of size at most clog$n$; there are at most n*2^{clogn} $= n^{c+1}$ such vertices. Add an edge between two vertices x,y in U if and only if the Turing machine answers yes when given the bit strings corresponding to x and y as input. This graph can be constructed in polynomial time.

Let G be a member of Cn, and let R be an implicit representation of G. To get G as an induced subgraph of U, label vertices of G from 1 to n and select for vertex i of G the ith copy of the bit string corresponding to i in R. Each vertex x of G is mapped to a vertex with the same bit string x has in R, and vertices in U are adjacent iff vertices with these bit strings are adjacent in R, so G is an induced subgraph of U.

We note that the proof of [104] reduces the number of vertices in U to n^c by assuming that every vertex is assigned a unique bit string. We leave this problem to exercise 13.

2.2 Generalized Implicit Representation

By definition, implicit representations are only possible for classes of graphs which have $2^{O(nlogn)}$ members on n vertices. This restriction was probably made to let implicit representation be closely tied to other 'good' representations; all known implicit representations also allow constant time adjacency testing. However, there is nothing which prevents us from generalizing the notion of implicit representation to larger classes of graphs, keeping the property that our representation should be asymptotically space optimal, and that adjacency testing can be performed using only information stored at the vertices being tested.

Let C be a class of graphs. Suppose that C has $2^{f(n)}$ members with n vertices. We want our representation to be asymptotically space optimal, so it should use $\mathrm{O}(\mathrm{f}(n))$ space overall. Distributing information to the vertices means that we allow $\mathrm{O}(\mathrm{f}(n)/n)$ bits of information to be stored at each vertex. Local adjacency testing means that adjacencies between x and y can be tested using only the bits stored at x and y. This form of representation is called a generalized implicit representation. A more general form of the implicit graph conjecture remains open.

Generalized Implicit Graph Question: Does every hereditary class of graphs have a generalized implicit representation?

For example, let us consider the class of all graphs; the same argument holds for any class of graphs with $2^{\Omega(n^2)}$ members. We are allowed to use $\mathrm{O}(n^2)$ bits overall to represent such graphs, distributing $\mathrm{O}(n)$ bits to each vertex. A simple implicit representation can be obtained by storing with each vertex the corresponding row of its adjacency matrix.

We note that storing the adjacency list of each vertex rather than the row of the adjacency matrix will not give a generalized implicit representation. An entry of the adjacency list uses as many as $\log n$ bits, and storing, for example, the complete graph uses $\Theta(n^2 \log n)$ bits if adjacency lists are used.

By using generalized implicit representations, we will be able to consider larger classes of graphs. Many parameters used to characterize classes of intersection extend naturally to arbitrarily high dimension, as we shall see in chapter 3. We will deal with open problems on well known classes of graphs and matrices which have more than $2^{O(nlogn)}$ members in chapters 10 and 11. For some other classes, the number of graphs in the class is unknown, but we would hope for a generalized implicit representation even if the number of graphs proves to be larger than $2^{O(nlogn)}$.

No book can hope to deal with all aspects of the classes of graphs we discuss in this book. For example, this book will ignore problems dealing with extremal graphs for particular classes, and special theorems for the particular classes of graphs. We will focus on the following problems which are particularly tied to the problem of representing the graphs efficiently. These problems are closely related to each other, so it is not always possible to say we are dealing with each problem in a separate section, but it is helpful to remember that each problem on a class of graphs is potentially different.

1) Counting

If we want a good representation (often a generalized implicit representation), we need to know the number of graphs in the class. For the purposes of this book, we are satisfied with a result that says the number of graphs in the class is $2^{\Theta(f(n))}$, which implies that we can hope for a representation which use O(f(n)) bits.

2) Representation

Representation is the primary focus of this book. In general, we like representations which are asymptotically space optimal and allow fast adjacency testing, and generalized implicit representations. For some classes, such as chordal bipartite graphs and visibility graphs, for which no representation is known which meets these criteria, we discuss forms of representations which come close to these goals.

3) Constructing Representations

Some classes of graphs have efficient representations, often as a consequence of the definition of the class, but these representations may be very hard to construct. For such classes, we often distinguish between the complexity of a problem if we are given a good representation, and the complexity of a problem if the input graph is given in standard adjacency matrix or adjacency list form. Thus, we are interested in the question of efficient construction of a representation, as well as the question of the existence of a representation.

The construction problem is often related to the recognition problem discussed below, but this is not always the case. It is sometimes possible to construct an implicit representation for a class of graphs even without recognizing the class. This may be done by constructing a representation for a class of graphs which contains the class in question. For example, even though planar Hamiltonian graphs are difficult to recognize, we can easily construct an implicit representation of planar graphs. In other cases, a recognition algorithm may not imply a good representation. For example, the chordal bipartite graphs discussed in chapter 10 can be recognized efficiently, but the algorithm does not produce an efficient representation.

4) Recognition

The recognition problem for a class of graphs is the problem of determining whether an input graph is in the class. Sometimes, algorithms for recognizing whether a graph is in the class lead directly to efficient algorithms for constructing a representation for the class. In other cases, we may suspect that an input graph is from a class which allows a good representation or can be optimized efficiently. In such cases, we first perform a recognition algorithm to see if the graph is in the class, and if it is in the class we construct a representation or solve the optimization problem in question.

5) Characterization

In the characterization problem, we are looking for a representation which is possible if and only if the graph is in the class. The following open problem may help to make the distinction between characterization and representation more concrete.

A number of interesting graph classes are defined on the basis of whether the graph and/or its complement is chordal and/or comparability graphs. Interval graphs are equivalent to graphs which are both chordal and co-comparability graphs. Permutation graphs are equivalent to graphs which are both comparability and co-comparability graphs. It is natural in this context to consider graphs which are both chordal and comparability graphs. These graphs were shown to have an implicit representation in [117] where it is shown that they were a subset of comparability graphs of posets which have dimension at most 4. However, this representation is not a characterization; many comparability graphs of 4 dimensional posets are not chordal comparability graphs. In fact, comparability graphs of 4 dimensional posets are NP-complete to recognize, while chordal comparability graphs can be recognized efficiently. To many researchers, this is intuitively unsatisfying; the representation may be good as a storage mechanism, but it does not get to the heart of what makes the class easy to deal with.

Open Problem: Find an implicit representation which characterizes chordal comparability graphs.

6) Optimization

One reason to study special classes of graphs is that we often find that problems which are intractable on general graphs are solvable efficiently on special classes of graphs. Sometimes efficient algorithms on a class of graphs can only be found after a deep study of properties of the graphs; optimization algorithms for perfect graphs due to Grötschel, Lóvasz and Schrijver [84] are among the deepest results in the study of algorithms. In this book, we restrict our attention to algorithms which arise fairly directly from the representations of graph classes being studied.

For classes of graphs which have good representations, one must make the distinction between solving the optimization problem if the representation is given as input, and solving the problem if the input is the adjacency list representation of a graph which happens to be in the class. This distinction is often ignored, even in the literature of algorithms for special classes of graphs.

Suppose that the input graph is given in adjacency list form. The assumption is often made that in order to solve an optimization problem on a class of graphs which has a special representation, we must first run a recognition algorithm for the class, then construct a representation, and perform the optimization procedure on the representation. Of course, this assumption is made because there are many cases for which the representation is the key to constructing an efficient algorithm for dealing with the class of graphs. We will encounter examples where this assumption is completely invalid; there are classes of graphs which are NP-complete to recognize, but natural optimization problems can be solved in polynomial time even if the input is given in adjacency list form.

Surprisingly, the author knows of no natural problems which can be shown to require a special representation in order to find a polynomial algorithm. Thus, I give the following open problem, which is not precisely stated; I freely acknowledge that the words 'natural' and 'well-known' are subjective, but are nevertheless important to making a solution to the problem interesting. I prefer to leave the definition imprecise rather than formal, since there are a number of 'trick methods' for finding such a problem which do not address the real issues involved. For example, the term nontrivial usually means that the answer is not always 'yes' or always 'no'; I want to use a slightly broader meaning here. This rules out such 'solutions' as asking to construct an interval dimension k representation of a graph (which is NP-hard if G is given in adjacency list form) with and without an interval dimension k representation. We will return to this open problem in the chapter on optimization.

Open Problem: Find any well-known NP-complete problem on a class of graphs with a natural efficient representation such that the problem is intractable under standard complexity theory assumptions if the input graph is in the class and the representation is not given, but is polynomial and nontrivial if the representation is given.

In some sense, an implicit representation is one method for capturing the notion that a graph class can be stored using little space, and so that adjacency testing can be done nicely. Other approaches are also possible; for example, Ian Stobert has proposed using the size of the representation times the time needed for an adjacency test as a measure of the effectiveness of a representation.

2.3 Representations with Very Short Labels

The implicit graph question may seem overly ambitious. There are many ways to define graph classes with $2^{O(nlogn)}$ members but which have no obvious implicit representation, as we will see during the course of this book. What reason is there to believe that all of these classes can be represented implicitly? Indeed, the implicit graph question is very much open; there is no consensus in the field that it should be true or false.

This section presents a result due to Scheinerman [143] and Scheinerman and Zito [145] which show that if we place what seems to be a small restriction on the graph class, then the class can be represented extraordinarily efficiently. Specifically, any hereditary class of graphs which has at most n^{kn} members, where k is less than .5, has an implicit representation which uses only a constant number of bits per vertex! Since n^{kn} is $\Theta(2^{\Theta(nlogn)})$, this implies that every hereditary class of graphs which has $2^{o(nlogn)}$ members has $2^{O(n)}$ members. We prove a slightly weaker version of their result, showing that this is true for a smaller value of k.

To prove this, the authors introduce a relationship between vertices which we will call a *twin relationship*. Formally, two vertices v,w are twins if N(v) - w is equal to N(w) - v. The reader familiar with the concept of modules or homogeneous sets, which will be defined in later chapters, may recognize twins as vertex pairs which form a module of size 2. The twin relationship [1] is an equivalence relation on the

[1] It is interesting to note that Ed Scheinerman, who wrote about the twin relationship in this paper, regards me as his 'evil twin'. This is because Ed and I both did graduate work on special classes of intersection graphs at Princeton at the same time; however, since he was in math and I was in computer science, we did not know of each other's existence until we were almost finished

set of vertices (exercise 17), and this equivalence relation partitions the vertices into sets, each of which must form either an independent set or a clique. The key result is that if the number of graphs in the hereditary class is small, then the number of equivalence classes is bounded by a constant.

For this result, we need to introduce a basic result from Ramsey theory. For every n, there is a value N such that every graph on N vertices contains either an independent set of size n, or a clique of size n. This result is not trivial, so readers unfamiliar with Ramsey theory must take this result on faith; for a good book on Ramsey theory, see [82].

The proof divides into two cases. If there are graphs in the class such that the twin relationship has an arbitrarily large number of equivalence classes, we will show that the number of graphs in the class grows too quickly, while classes in which the twin relationship never has more than a constant number of classes can be represented using a constant number of bits per vertex.

Suppose that the number of equivalence classes of the twin relationship for a class of graphs is unbounded. For any n, there is some graph G in the class which has at least N equivalence classes, where N is the Ramsey theoretic bound discussed above. Let S be a set with one vertex from each equivalence class. The graph induced by S must contain a clique or independent set of size at least n; let C be a clique or independent set of size $n/2$ in the subgraph of G induced by S.

Choose a minimal set of vertices Y such that for any pair of vertices u,v in S, there is a vertex in Y adjacent to exactly one of u and v. It is not hard to see that Y has size smaller than $n/2$ (exercise 18). Add vertices arbitrarily to S and Y to get a set of n vertices, and consider the subgraph F induced by this set. F has at least $n/2$ equivalence classes with respect to the twin relationship, and thus at least $(n/2)!$ different labelings of the vertices. This can be seen by fixing all labels except those in S, and any of the $(n/2)!$ labelings of S produces a different labeled graph, since there is some vertex of a fixed label which distinguishes each pair of vertices. Since the class of graphs is hereditary, F must be a graph in the class, and thus all relabelings of F produce a graph in the class.

Note that S is chosen to be a clique or independent set in the proof so that vertices of S cannot distinguish other vertices in S; this restriction is not strictly necessary, but may make the proof easier for some readers to see.

Therefore, if the number of equivalence classes of the twin relationship for a class is unbounded, then for any value of n we have at least $(n/2)!$ different labeled graphs in the class.

Suppose that the number of graphs on n vertices in a class is at most $2^{o(nlogn)}$. By the argument above, the number of equivalence classes for graphs in the class is bounded above by a number T. We did not argue that all graphs have the same (that is, isomorphic) set of equivalence classes. Therefore, we cannot use the simplest

with our degrees. Our work has continued along similar paths, with mine stressing algorithms for intersection graph classes, and his dealing with nonalgorithmic issues on similar classes.

strategy of assigning each vertex a number from 1 to T, and having our adjacency test know whether vertices in a particular pair of equivalence classes are adjacent or nonadjacent. This idea can be modified easily, however, to give a representation with a constant number of bits per vertex.

Let G be any graph in the class. Let e be the number of equivalence classes of the twin relationship for G. Construct a matrix C which has a row and column for each equivalence class of G, with the entry $C[i,j] = 1$ if and only if vertices in classes i and j are adjacent; the diagonal element $C[i,i] = 1$ if and only if equivalence class i is a clique. Store with each vertex v the matrix C and the number of the equivalence class of v. This form of representation uses $O(e^2 + \log e)$ bits for a general graph; since e is bounded above by a constant T for the class, we store a constant number of bits per vertex, and adjacency can be tested locally.

This result seems to depend heavily on the fact that graphs are labeled, so that certain simple subgraphs correspond to an enormous number of different labeled graphs. It would be interesting to see whether the results of this section can be extended to the unlabeled case as well. For example, there is something inherently unsatisfying about using the fact that the number of labeled matching graphs (graphs in which each vertex has degree 1) is $(n/2)!$, and that thus any class which contains the matching graphs is large; nevertheless, this is completely consistent with the definition of implicit representation.

2.4 Exercises

1) Show that the number of unlabeled interval graphs is $2^{\Omega(n \log n)}$.

2) The line graph of G is a graph G' such that vertices of G' correspond to edges of G, with two vertices being adjacent in G' iff the corresponding edges share a common endpoint. Show that line graphs have an implicit representation.

3) A caterpillar is a graph in which the vertices of degree greater than 1 form a single induced path. Show that caterpillars have an implicit representation.

4) A k-tree is defined as follows. A clique of size k is a k-tree, and if G is a k-tree then the graph formed by making a new vertex adjacent exactly to some k-clique in G is a k-tree. A partial k-tree is a graph which can be obtained from a k-tree by deleting 0 or more edges. Show that for all k, the class of partial k-trees has an implicit representation.

5) A poset is N-free if there is no set of 4 elements a,b,c,d which induce an 'N' in the Hasse diagram (transitive reduction); i.e. there are no set of 4 vertices a,b,c,d such that the edges (a,b),(c,b),(c,d) are the only edges present between these vertices in the transitive reduction.

The following characterization of N-free posets comes from [85]. Let P_1, P_2 be two posets, A a subset of sources of P_1 and B a subset of sinks of P_2. A new poset is created by keeping adjacencies between vertices from the same poset, and adding edges from each vertex in B to each vertex in A (as well as those implied by transitivity from this addition).

a) Use the characterization above to show that the number of N-free posets on n vertices is $2^{O(nlogn)}$

b) Show that the set of N-free posets does not have an implicit representation. Hint: any poset can be transformed into an N-free poset by introducing a new vertex inside every edge of the original poset.

6) Let C be the class of graphs which are the transitive closure of directed trees. In other words, a member of C corresponds to a tree with edges added from all ancestors to all descendants. Give an implicit representation for this class of graphs.

7) Consider the intersection graphs of paths in a tree discussed on page ?. Show that if a graph G with n vertices is the intersection graph of paths in a tree, then G is the intersection graph of paths in a tree T with O(n) vertices.

8) Show that intersection graphs of paths in a tree have an implicit representation.

9) A graph is a cograph if it can be constructed using the following rules. A single vertex is a cograph. If G1 = (V1,E1) and G2 = (V2,E2) are cographs then G3 = (V1 ∪ V2,E1 ∪ E2) and G4 = (V1 ∪ V2,E1 ∪ E2 ∪ V1xV2) are cographs. Show that the number of cographs is $2^{O(nlogn)}$.

10*) Show that every cograph is both a comparability graph and a co-comparability graph; this implies using results from chapter ? that cographs have an implicit representation.

11) The total graph T(G) of G has a vertex for each vertex and edge of G. Two vertices of T(G) which correspond to vertices of G are adjacent iff the vertices are adjacent in G, and vertices which correspond to edges of G are adjacent iff they share a common endpoint (as in the line graph). If v_t is a vertex of T(G) corresponding to a vertex v of G and v_e is a vertex of $T(G)$ corresponding to edge e of G, then v_t is adjacent to v_e iff v is an endpoint of e. Show that the total graphs do not have an implicit representation.

12*) An autograph is a graph which can be derived from a set of n positive integers by making a vertex for each integer, and letting i and j be adjacent iff $|i$ - $j|$ is also in the set. Show that the autographs do not have an implicit representation.

13) Show how to modify any implicit representation so that no pair of vertices receives the same label.

14) Show that comparability graphs, chordal graphs, and chordal co-chordal (also known as split) graphs all have $2^{\Omega(n^2)}$ members. What does this imply about the existence of implicit and generalized implicit representations for these classes?

15*) Let Ck be the class of graphs defined by forbidding paths of length k as induced subgraphs. Can you determine the first value of k such that Ck does not have an implicit representation?

16*) A graph is c-decomposable if for every induced subgraph H there is a set S of at most c vertices such that no component of G - S contains at most 2/3 of the vertices in H. Show that for any value of c, every c-decomposable graph has an implicit representation.

17) Show that the 'equivalent neighborhood' relationship is an equivalence relationship, and thus partitions the vertex set of a graph. Show how to partition the vertex set into these equivalence classes in O(n+m) time.

18) Let G be an arbitrary graph, and let U be a set of vertices such that no pair of vertices in U are twins. Show that there is a set S of at most $|U|$-1 vertices such that any pair of vertices is distinguished by a vertex in S.

19) Show that every graph with at least 6 vertices contains either an independent set of size 3 or a clique of size 3, and that this is not true for some 5 vertex graph.

20) In this book, we generally deal only with labeled graphs; this largely avoids the difficulties of dealing with problems relating to the number of labelings of isomorphic copies of the same graph. Show that there is a constant c such that if a graph class contains at least 2^{cnlogn} members on n vertices, a graph class has $2^{O(f(n))}$ labeleed members if and only if the class has $2^O(f(n))$ unlabeled members.

Note: many of these exercise come from examples in [129].

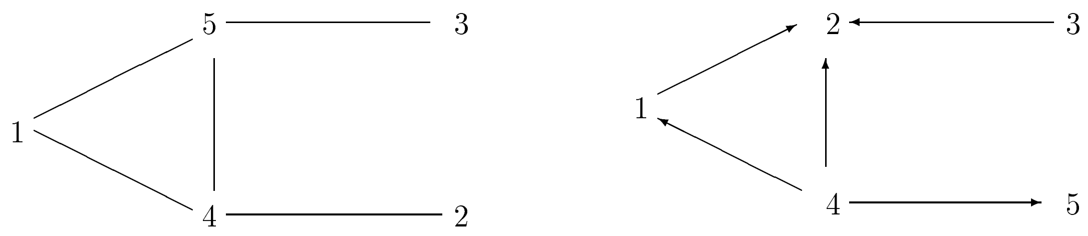


Figure 3: Perfect Elimination Scheme 1 2 3 4 5, and Transitive Orientation

3 Intersection and Containment Representations

In this chapter, we discuss classes of graphs defined by simple intersection and containment models. We will introduce a variety of types of graphs, and discuss techniques which apply to the classes. Recognition of many of these graph classes is studied in chapter 13. We start by introducing two graph theoretic properties which a good number of these graph classes satisfy, and allow various optimization problems to be solved efficiently.

3.1 Chordality and Transitive Orientation

A graph G is chordal if there are no chordless cycles in G of length greater than three. Chordal graphs always contain a vertex v such that the neighborhood of v is a clique; such a vertex is called a simplicial vertex. This gives rise to a 'perfect elimination scheme' v_1 v_2 ... v_n in which each v_i is simplicial in the graph induced by v_i,$v_i + 1$ .. v_n. Many algorithms on chordal graphs can be solved efficiently by stepping through the graph in the order defined by the perfect elimination scheme. For example, to find the maximum independent set in a graph, we can greedily select vertices from the perfect elimination scheme if they have no previous neighbor in the independent set; in the graph of Figure ?, we select 1,2,3 and reject 4 and 5.

A graph G is a comparability graph if the edges of G can be directed to produce a transitive digraph G'; in other words, whenever $x \to y \to z$ in G', there must be an edge from x to z. The directions assigned to edges are called a transitive orientation of G. We can solve certain problems efficiently on comparability graphs by stepping through the vertices in an order defined by any linear extension of the transitive orientation of G. To color a comparability graph optimally, we take a linear extension of the transitive orientation, and give each vertex the smallest color not used on an earlier neighbor. If we use the extension 3 4 5 1 2 in the graph of Figure ?, we color 3 and 4 with color 1, 5 with color 2, 1 with color 2, and 2 with color 3.

Some interesting generalizations of chordal and co-comparability graphs are studied in chapter 8. Specifically, we generalize chordal graphs to weakly chordal (also called weakly triangulated) graphs, in which chordless cycles of length four are allowed, but no longer chordless cycles are allowed in G or its complement. It has been observed that various problems seem to be more easily solvable on co-comparability graphs than on comparability graphs. Attempts to explain this phenomenon have led to the study of asteroidal triple-free graphs, which contain the co-comparability graphs. These graphs will be defined and studied in chapter 8.

Chordal graphs and comparability graphs, as well as their complements, are subclasses of perfect graphs. Although this implies polynomial algorithms for a number of problems using the algorithm of [84] outlined in chapter 1, these problems are solved much more easily on the smaller classes. The clique and independent set problems are solvable in linear time on chordal graphs; these algorithms come from the fact that the perfect elimination scheme can be constructed in linear time, as described in chapter 13. The clique problem can be solved in linear time on comparability graphs. This comes from the fact that a transitive orientation can be found in linear time, although some interesting issues arise from the fact that comparability graphs cannot yet be recognized in linear time; see chapters 4 and 14 for details. The independent set problem can be solved on comparability graphs by a reduction to bipartite matching; it might be difficult to improve upon this bound since improvements would lead to improvements in the time for finding the size of a bipartite matching (see chapter 14, exercise 9).

3.2 Techniques for Interval Graphs

Interval graphs are a particularly nice class of graphs in that many different algorithmic techniques produce efficient algorithms for interval graphs. We will review some of these techniques, and see that as interval graphs are generalized in the following section in different ways, some of these techniques will still be effective while others no longer apply.

It is not hard to see that interval graphs are both chordal and co-comparability graphs, and thus any problems solvable for the larger classes are solvable on interval graphs. In fact, interval graphs are exactly equal to chordal co-comparability graphs [77].

Many problems which are difficult on general graphs can be solved on interval graphs using a simple greedy approach. Consider the independent set problem. It is easy to see that the interval which ends first can be included in a maximum independent set. To solve the independent set problem on interval graphs when we are given a representation, we can place this interval in our independent set, delete the vertex and its neighbors, and continue until all vertices are deleted.

A graph class defined by a geometric intersection model has the Helly property if for every clique C, there is some single point p such that every vertex of C includes the point p. Interval graphs have the Helly property; clearly, this means that the maximum clique problem is solvable efficiently given an intersection model, since we only need to check how many vertices cover each point. This simple observation leads to some less obvious algorithms when we deal with classes for which the geometric model cannot be constructed efficiently. It is not hard to see that the Helly property implies that there are O(n) maximal cliques in an interval graph. There are algorithms which enumerate all maximal cliques in time polynomial with respect to the number of cliques [][]; this implies that there is a polynomial algorithm for the clique problem, even if we do not know how to construct an interval model. This will be useful for finding maximum cliques in some classes defined in the next section which are NP-complete to recognize.

Interval graphs also are amenable to other algorithmic techniques. Since the greedy technique is so effective on interval graphs, relatively few algorithms in the literature use the more complex techniques on interval graphs, but we will present them on interval graphs and discuss whether they are effective or ineffective for various generalizations of interval graphs.

Dynamic programming is also an effective technique for many problems on interval graphs and their generalizations. In dynamic programming, we build up a solution for a large problem by solving the problem first on all its subproblems; for this to be effective, the number of subproblems must be reasonably small (for example, we cannot solve an exponential number of subproblems), and the solution to a large problem must be able to be deduced quickly from the solutions to all the smaller problems.

Once we have the idea of using dynamic programming on interval graphs, most of us would bound the number of subproblems as follows. Consider the points of the real line which correspond to an endpoint of at least one interval of the interval graph, and number these from 1 to (at most) 2n on the basis of position in the line. For each pair of integers $i < j \leq 2n$, we can define a subgraph by restricting ourselves to the region between i and j. Depending on the problem, we may look either at intervals such that both endpoints occur within the interval (i,j), or intervals with at least one endpoint in (i,j). In either case, there are O(n^2) subproblems to consider, and subproblems are considered in increasing order of j - i.

A relatively new algorithmic technique which works for a number of the classes described in this chapter is based on minimal separators. A set S of vertices separates a from b if there is no path from a to b in G - S; a separator is minimal if removing any vertex from S leaves a path from a to b. In general, the number of minimal separators of a graph is exponential (exercise ?). For interval graphs, any minimal separator corresponds to the set of vertices which pass through some point in the interval model (exercise ?). This implies that interval graphs have O(n) minimal separators, and these can be found efficiently. For general graphs, all minimal separators can be enumerated in O(??) time, where ? is the number of minimal separators in G. Problems that can be solved efficiently using this technique are given in [].

3.3 Generalizations of Interval Graphs

There is a great deal of literature on interval graphs, since they have important practical applications as well as interesting mathematical structure. Since interval graphs have proved to be very easy to deal with, many attempts to generalize the

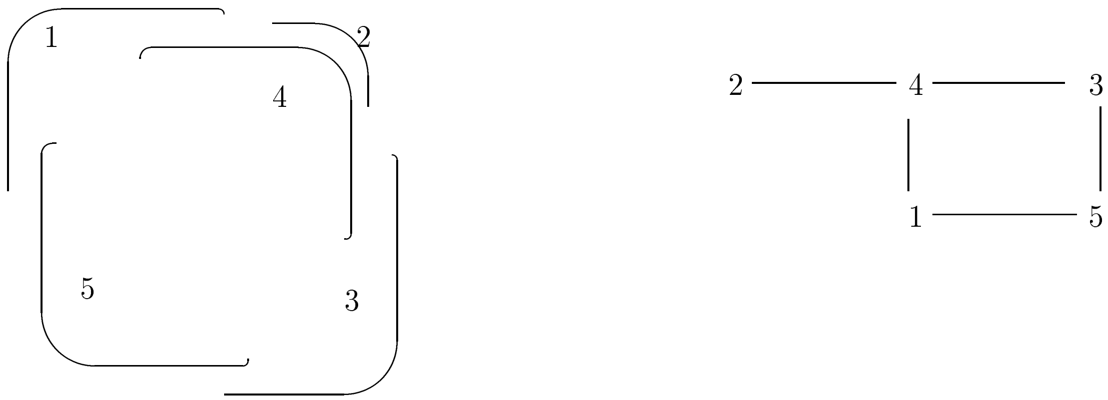


Figure 4: circular-arc model for a graph

class in different directions have been made. We will not attempt to cover all the work on generalizations of interval graphs, but we will try to familiarize the reader with some of the more popular representations which have been attempted, and introduce some open problems related to them.

If an interval graph is viewed in a certain light, it is easy to conceive of a number of generalizations. In an interval graph, each vertex is a line segment, and these line segments are placed on the real line. We will start by considering generalizations of the real line, where each vertex will still correspond to a single segment.

Circular-Arc Graphs

One popular generalization of interval graphs is the class of circular-arc graphs, in which vertices correspond to arcs of a circle rather than intervals on a line; a circular-arc model is shown in figure ?. Circular-arc graphs are not perfect, since cycles of any length are circular-arc graphs.

Natural greedy algorithms are more complex than or not as effective for circular-arc graphs than they are for interval graphs. Coloring, which can be solved optimally by a greedy algorithm for interval graphs (exercise ?) is NP-complete for circular-arc graphs. Consider the independent set problem for circular-arc graphs. If we start at a point p on the circle, and there are a number of mutually overlapping arcs which cross p, we cannot decide locally which arc to include in the maximum independent set. There are algorithms which solve this problem in $O(n)$ time given the model; these step through the model in order, maintaining the greedy independent set if we start from each candidate starting point in an efficient manner [78], [122],[99] but getting this time bound is much harder than for interval graphs.

Many problems are solved on circular-arc graphs by reducing the problem to an interval graph problem. For example, the independent set problem can be solved easily in $O(dm)$ time, where d is the minimum degree of any vertex in G, even if the model is not given. This can be done because removing all neighbors of any vertex in a circular-arc graph leaves an interval graph. If v is the minimum degree vertex, we try including v and each neighbor of v in the independent set, removing all neighbors of the vertex included, and solving the problem optimally on the remaining interval graph. Independent set on circular-arc graphs if the model is not given is also discussed in chapter ?.

The number of minimal separators in a circular-arc graph remains polynomial, since any minimal separator corresponds to the set of arcs which pass through two points of the circle (exercise ?). This allows problems such as ? to be solved in polynomial time on circular-arc graphs []. These algorithms are similar conceptually to dynamic programming on circular-arc graphs, since the natural dynamic programming approach is to deal with regions defined by cutting the circle at two points (examples of DP?).

Boxicity

Boxicity is a natural way of generalizing interval graphs to higher dimension. A graph has boxicity k if it is the intersection graph of isooriented boxes in k-dimensional space. It is NP-complete to determine whether a graph has boxicity k for $k > 1$; for $k = 1$, this is simply interval graph recognition. However, boxes in k-space still have the Helly property, so the number of maximal cliques is $O(n^k)$ for boxicity k graphs. Although it is not immediately obvious, every graph can be represented as the intersection graph of boxes in d-dimensional space for some d; the minimum such d is called the boxicity of G. To see that every graph can be represented, note that two boxes defined by a set of intervals in d dimensions intersect if and only if the intervals intersect in all dimensions. Therefore, we create a dimension for each vertex x. Let x correspond to a point in this dimension which is covered by all neighbors of x and uncovered by nonneighbors, and have all vertices except x intersect outside this point.

Disk Intersection

These graphs are the intersection graphs of circles in the plane, or more precisely, of disks in the plane since circles are considered to be adjacent if one circle completely contains another. (q on whether clique in P? look up MR claimed $O(n^2)$ algorithm in Chazelle and Lee, On a circle placement problem, Computing 36, 1-18,1986; now looks like unit circles to me, must find paper. Maximum independent set NP-complete in Wang and Kuo, A study on two geometric location problems, IPL 28, 281-286, 1988. Read Breu paper for other techniques)(color, ind set, dom NPC even for grid Gs = subset unit disk; clique in P given model; Clark Colbourn Johnson Unit Disk Graphs DAM 86 1990 165-177; rec NP-hard unit disk Breu Kirkpatrick)

Interval Number

A graph has interval number k if there is a mapping from each vertex to a set of k intervals on the line, such that two vertices are adjacent in G if and only if at least one pair of the corresponding intervals have a nonempty intersection. Recognition of interval number k graphs is NP-complete for all $k > 1$.

Interval number, like boxicity, is a parameter which can be used to measure ‘complexity’ in graphs. Clearly every graph has an interval number (exercise ?). Since graphs with interval number k can be represented using $O(kn\log n)$ bits, there must be graphs with interval number at least $n/\log n$. In fact, there are graphs with interval number $\Theta(n)$, though the interval number of a random graph is $\Theta(n/\log n)$. If we want to define a graph class with $2^{\Theta(f(n))}$ members for values of f(n) between $n\log n$ and n^2, we can generally pick an appropriate value for one of these parameters; for example, we can define a class of graphs as having interval number at most $n^{.5}$. This class cannot have an implicit representation, since there are too many graphs in the class, but the natural representation is usually generalized implicit.

Interval number has been generalized to such parameters as rectangle number (each vertex is associated with at most k rectangles in the plane), circular-arc number, and in other similar ways. One open problem which has generally been ignored on such classes is the problem of constructing an efficient representation in polynomial time; this may be be possible even if the parameter is NP-complete to compute. We will discuss such problems in more detail in chapter ?

There are few, if any, examples, of algorithms which work efficiently on interval number 2 graphs.

Examples of DP algorithms? Complexity of clique,IS?

Interval Count

Most of the graph classes in this chapter are defined by an implicit representation. The graph class studied here has many of the properties required for an implicit representation, but varies in one key way which makes finding an implicit representation impossible for the class.

Consider a representation of a graph in which each vertex corresponds to a set of intervals, with an edge between two vertices if and only if some intervals from each set have a nonempty intersection. Interval count differs from interval number in that the number of intervals per vertex is not fixed; bounds are placed instead on the total number of intervals in the set. If we bound the interval count by cn for some fixed c, an interval representation is space efficient, local in the sense that all information about adjacency between x and y is stored with x and y; however, the number of bits at a single vertex may be larger than $\Theta(\log n)$.

We can also view an interval graph as the intersection graphs of paths in a larger path. Circular-arc graphs generalize this by being intersection graphs of paths in a cycle. We can also think of intersection graphs of paths within more complex graphs. For these classes it can make a difference (sometimes a surprisingly large difference) as to whether we define path intersection as sharing a common vertex or sharing a common edge. For example, path graphs are the intersection graphs of paths in a tree if we define intersection as meeting at a common vertex, while EPT (meaning edge intersection of paths in trees) graphs are the intersection graphs of paths on trees if we define intersection to mean sharing a common edge. Path graphs can be recognized in polynomial time, while EPT recognition is NP-complete. Furthermore, path graphs are chordal, implying that such problems as clique and coloring can be solved in polynomial time on these graphs, while EPT graphs are not even necessarily perfect, and the chromatic number problem is NP-complete on EPT graphs [163].

Intersection graphs of paths on grids [], planar graphs [], and series-parallel [] graphs have also been considered. Intersection graphs of paths on trees can still be stored using a small amount of space, since paths are uniquely defined by a set of endpoints, while path graphs on larger classes cannot since specifying the path requires using a large amount of space.

3.4 Permutation Graphs and Their Generalizations

A variety of intersection graph classes are defined by making objects correspond to sets defined on two parallel lines, rather than one parallel line as in interval graphs. The simplest, and most heavily studied of these classes, is called the permutation graphs. In a permutation graph, each vertex corresponds to a line segment drawn between the two parallel lines, with vertices being adjacent iff the corresponding segments cross in the model. Permutation graphs are exactly equal to graphs which are both comparability and co-comparability graphs [53] [79].

Note that it is more usual to define permutation graphs in terms of a single permutation of the vertex set; this corresponds to labeling vertices from 1 to n on the basis of the order of the endpoints along one line, and specifying only the order of endpoints along the other line.

The next most popular class defined using similar structures is the class of trapezoid graphs. These are the intersection graphs of trapezoids defined by pairs of intervals, one on each parallel line. Many dynamic programming and separator based algorithms on permutation graphs also work on trapezoid graphs.

Corneil and Kamula defined a pair of graph classes which lie between the permutation graphs and the trapezoid graphs. In both classes, vertices correspond to triangles with all endpoints on the two parallel lines. In the first class, called PI (for point interval), all triangles have one endpoint on the top line, and two endpoints on the bottom line. In the second class, called PI^*, each triangle may have one or two endpoints on the top line. Although there are polynomial time recognition algorithms for permutation graphs and trapezoid graphs, the problems of recognizing PI and PI^* graphs remain open.

[66] studies a further generalization to d-trapezoid graphs. In this generalization, we use d parallel lines rather than two. Each vertex is assigned two points on each parallel line; a d- trapezoid is the object created by connecting the left (right) endpoint of each line i with the left (right) endpoint on line i+1, and the left and right endpoints on the first and last line.

These classes are all co-comparability graphs, since we can order nonadjacent vertices according to whether the corresponding geometric object is to the left of the other; clearly if x is to the left of y and y is to the left of z, then x is to the left of z. Many problems on these classes are solved using dynamic programming. In its simplest form, we can consider all 'regions' of the diagram defined by cutting each line at two points in increasing order of size; a variety of more complex variants of dynamic programming have also been developed []. A more recent technique to solve problems on these classes uses the fact that the number of minimal separators is polynomially bounded, since separators correspond to sets that cross a 'scanline' defined by a point on each line. For a variety of problems such as treewidth, a value can be computed by optimizing over all possible separators, and computing a value for each component with the separator removed. If the number of minimal separators is bounded, and the number of different pieces which can be encountered is polynomial, this technique can be used to solve some problems in polynomial time even when the model is NP-complete to construct [].

The generalization to two lines has itself been extended by defining objects which have endpoints on two circles. The first such class studied is called circular permutation graphs, in which vertices map to chords which go between two concentric circles. This is not quite an intersection graph model, since vertices are defined to be adjacent if the corresponding chords intersect exactly once, rather than if there is any intersection. I prefer a containment graph characterization of this class which will be described in the next section; these are containment graphs of arcs in the circle. Circular trapezoid graphs have also been studied [66].

Circle graphs are the intersection graphs of chords in the circle. These generalize permutation graphs, since any permutation diagram can be transformed to a circle graph representation by connecting the two lines at each end to form a circle. Circle graphs are no longer perfect, since a cycle of any length is a permutation graph; the ? and ? problems are NP-complete on circle graphs. Some problems on circle graphs, such as the maximum clique problem, are easily solved in polynomial time by cutting the circle at appropriate places and solving corresponding permutation graph problems. The recognition problem for circle graphs, however, works rather differently, and is discussed in chapter ?. number separators?

[55] defines an infinite number of classes lying between the permutation graphs and the circle graphs. G is a k-polygon graph if G is the intersection graph of chords in a k-gon, where any chord must have endpoints on two different sides of the k-gon. (techniques, complexity?)

We can generalize circle graphs by allowing vertices to correspond to line segments in the plane. This class is NP-hard to recognize, even if lines are restricted to being parallel to the x and y axes; the restricted class is called grid intersection graphs [], []. Perhaps the ultimate generalization within the plane is the string graphs, in which vertices correspond to arbitrary non-self-intersecting strings in the plane. These classes are generally difficult to deal with.

Instead of varying the circle, it is also possible to make variations on the object inside the circle. ? suggested classes defined by mapping vertices to polygons inscribed in the circle, with vertices being adjacent if the polygons intersect. For any fixed k-gon, this gives an implicit representation. The most natural class to study is the intersection graphs of triangles inscribed in the circle; very little is known about this class. The class defined by intersection of arbitrary k-gons (also called spider graphs) inscribed in the circle is studied in [107].

3.5 Containment Representations

We can also define graphs by letting vertices correspond to geometric objects, but letting edges correspond to other types of relationships besides intersection. The most common other type of relationship in the literature is to allow an edge from x to y if object x contains object y. This naturally corresponds to a directed graph, and the graphs which come from the representation are partial orders.

The first containment model to consider is defined by a set of intervals on the line. The corresponding class of partial orders is the two dimensional partial orders. The term dimension will be used in a number of places in this book, and the terms come out of the definition of partial order dimension.

A linear extension of a partial order is an ordering of the elements so that every edge of the partial order goes forward in the ordering. The dimension of a partial order P is the minimum number of linear extensions of P which produce P as their intersection. In other words, this is the minimum number of orderings of the elements such that for all edges (x,y), x comes before y in all orderings, and for every nonadjacent pair of elements (x,y), x precedes y in some ordering while y precedes x in another. A set of linear extensions which represent the poset in this way is called a realizer. A realizer of size k can be used as the basis of an implicit representation, since we can determine whether x and y are adjacent from the positions of x and y in each of the k realizers. Trotter [167] has written an excellent book on partial order dimension theory.

The relationship between two dimensional partial orders and containment graphs of intervals was first noted by Dushnick and Miller [53], and is not difficult to see. If we are given a set of intervals, we form a list of left endpoints of the intervals ordered from left to right, and a list of right endpoints of the intervals ordered from right to left. Then interval i contains interval j if and only if i comes before j in both lists, so all interval containment graphs are two dimensional partial orders. If we are given a realizer consisting of two linear extensions L_1, L_2 of a poset, combine these into a single ordering L_1 followed by the reverse of L_2. For each element x, the corresponding interval goes between the two endpoints in this ordering; clearly interval x contains interval y if and only if x contains y in L_1 and L_2. A partial order is two dimensional if and only if the undirected graph formed by removing all directions is a permutation graph, as defined above, and a graph is a permutation graph if and only if every transitive orientation is a two dimensional partial order (exercises).

Containment graphs of arcs on the circle are more commonly associated with an equivalent intersection model, the circular permutation graphs (move discussion from recognition chapter here?). Algorithms on this class commonly work by cutting the circle, and turning the problem into a permutation graph problem.

Containment graphs of circles have received a good deal of attention in the partial order community. One well known open problem is whether every three dimension partial order is a circle order. There is a conceptually interesting application of the more general notion of spherical orderers in k-space. We say that event a could have caused event b if a precedes b in time by a sufficient amount that a particle could travel from a to b in the intervening time. Causality orders of events in k- dimensional space are containment graphs of ?-dimensional spheres. Status of recognition ?? (intersection of spheres in n-space, see Breu's thesis)

Other classes of graphs for which containment graphs have been studied include angles on the plane [],[], regular polygons, and general polygons in the plane []. Complexity status ??

3.6 Overlap Representations

We can also define a graph class on a set of objects by saying that two vertices are adjacent if and only if they 'overlap'; that is, the intersection is nonempty and neither object contains the other. In general, overlap representations are less well studied than either intersection or containment representations.

Once again, the natural model to start with is overlap graphs of intervals on the line. This class is exactly equal to the circle graphs discussed above. To turn an interval overlap model into a circle graph model, simply connect the endpoints of the start and finish of the line to create a circle. To turn a circle graph model into an interval overlap model, cut the circle at any point which is not the endpoint of a chord. The proof that this is a valid transformation is deferred to exercise ?.

Although overlap graphs of each of the other classes of geometric objects can be defined, these have not been studied in general. Cenek and Stewart [] have studied some general theorems on overlap representations, one of which is described below.

The starting point for this work is Gavril's paper solving optimization problems on circle graphs, by reducing them to interval graph problems []. As one example, let us look at his algorithm for solving the maximum independent set problem on circle graphs; we will assume that the input is in the form of intervals, in which x is adjacent to y in G if and only if the corresponding intervals overlap.

The algorithm considers intervals in increasing order of interval size, computing the maximum independent set on the intervals contained in each v. We add a single interval to cover all others; when we find the maximum independent set on vertices contained within this longest interval, we are done.

Let v be the current vertex such that we are computing the maximum independent set on vertices contained in v in the overlap model. Let w be an interval contained in v; give w a weight of 1 + the maximum independent set on vertices contained in w. We then compute the weighted maximum independent set on the interval graph of the intervals contained in v, i.e. we consider the intersection model instead of the overlap model. For any independent set I in the intersection model, we can construct an independent set in the overlap model with this weight by finding the maximum independent set contained within each vertex of I. Similarly, any independent set I' in the overlap model corresponds to an independent set in the weighted intersection model, consisting of the maximal intervals in I', with weights corresponding to the weights of the maximum independent sets contained within each of these maximal intervals.

Cenek and Stewart note that this generalizes to other classes of overlap besides intervals; if you can compute the weighted independent set in the intersection family, you can solve the independent set problem in the overlap family. For example, this allows you to solve the independent set problem on the overlap graph of subtreees of a tree; in exercise ?, this is shown to contain such classes as chordal graphs, circle graphs, co-comparability graphs, and spider graphs.

A similar, somewhat more complex condition is given for solving the clique problem on a class defined by an overlap model.

3.7 Generalized Intersection Models

In this section, we discuss necessary and sufficient conditions due to Scheinerman [144] for a class of graphs to be characterized exactly by some intersection representation.

include similar for containment or just refer ???

Note that exact characterization means something quite different from finding a good representation. In order to characterize a class, the objects used in the intersection representation must be able to represent graphs in the class, and not be able to represent any graph not in the class. It is possible for a class of graphs to have an implicit representation in terms of intersection of objects, but not have an intersection characterization which characterizes the class. This can happen when a class has an implicit representation by virtue of being contained in a larger class. Thus in chapter ? we will see that every distance-hereditary graph is the intersection graph of chords of a circle, but the converse is not true; chords on a circle give an implicit intersection representation of the class, but they are not a characterization of the class.

The first necessary condition for a graph to be characterized exactly by an intersection representation is that the class is hereditary. If vertices correspond to sets so that vertices are adjacent if and only if the sets have a nonempty intersection, we can represent any induced subgraph by choosing the corresponding sets from the original intersection representation.

The second simple necessary condition for having a characterization is that the class is closed under vertex expansion, which takes any vertex v and replaces it by a clique C such that each vertex of C is adjacent to the same vertices of G - v. If G is representable, we can clearly represent the new graph by letting each c in C correspond to the same set as the set corresponding to v.

The final condition is slightly harder to see. There must be a countable sequence of graphs G^1, G^2, ... such that each G^i is in the class, each G^i is an induced subgraph of G^{i+1}, and every graph in the class is an induced subgraph of some G^j. The sequence G^1,G^2,... is called a composition sequence.

To see that classes defined by set intersection must have a composition sequence, let G^i be the intersection graph of all sets which correspond to some vertex of a graph G with at most i vertices which is in the class. Since all sets from G^i remain in G^{i+1}, G^i is an induced subgraph of G^{i+1}, and every graph with i vertices in the class is an induced subgraph of G^i.

Scheinerman showed that these necessary conditions are also sufficient for a graph class to have a characterization as the intersection of a collection of sets [144]. We will not present the proof here, but we will give examples which illustrate that it is often quite simple to use these properties to determine whether various classes of graphs are intersection classes.

Connected graphs, for example, do not have an intersection characterization, since connectivity is not a hereditary property. On the other hand, connected graphs clearly are closed under vertex expansion; they also have a composition sequence (exercise ?).

Forests (acyclic graphs) clearly are a hereditary class of graphs, but they are not closed under vertex expansion. They also have a composition sequence (exercise ?, or more generally by exercise ?).

Consider the class of graphs which do not contain both C_4 and C_5 as induced subgraphs. This class is obviously hereditary; it is also closed under vertex expansion (exercise ?). However, it cannot have a composition sequence. Otherwise, C_4 would be an induced subgraph of some G^i in the composition sequence, and C_5 would be an induced subgraph of some G^j in the composition sequence; without loss of generality assume that $i < j$. Since G^i is an induced subgraph of G^j, G^j must contain both C_4 and C_5; thus G^j would not be in the class, contradicting the assumption that this is a composition sequence.

3.8 Exercises

1) G is B1-orientable if and only if one can assign directions to the vertices so that if x and y both have an edge to z, there is an edge between x and y.

1a) Show that G has an acyclic B1-orientation if and only if G is chordal.

1b*) Design an O(n^3) algorithm to find a B1-orientation of a graph.

1c) Design an algorithm which verifies whether an orientation is a B1-orientation which runs in o(n^3) time.

2) Show that a co-comparability graph cannot contain any chordless cycles of length > 4.

3) Show that the largest possible number of maximal cliques of a boxicity 2 graph is O(n^2).

4) Show that there are graphs with boxicity 2 which are not circular-arc graphs.

5) Show that overlap graphs of intervals on the line are exactly the circle graphs.

6) Design an O(n+m) algorithm which takes a representation of a circle graph on the circle, and produces an adjacency list representation of the graph.

7) Show how to solve the clique problem on circle graphs in polynomial time; you can assume there is a polynomial time algorithm for the clique problem on permutation graphs.

8) Greedy coloring on interval graphs works as follows. Process vertices in order of leftmost endpoint, and assign v the smallest color not assigned to a neighbor of v. Show that this algorithm colors an interval graph using the optimal number of colors.

9a) Design a polynomial time algorithm for solving the dominating set problem on interval graphs, and analyze its time complexity.

9b) Design a polynomial time algorithm for solving the dominating set problem on permutation graphs, and analyze its time complexity.

10) Show that any minimal separator of a circular-arc graph corresponds to the set of vertices which pass through a pair of points on the circle in any arc model.

11) Show that every circular-arc graph has interval number at most 2.

12a) Show that it is possible to have a clique of size k in a represented as a circular-arc graph in which no point of the circle is covered by more than $k/2 + 1$ arcs.

12b) Show that if a circular-arc graph contains a clique of size k, there must be some point of the circle covered by at least $k/2$ arcs in any circular-arc representation.

Note: the bounds of 10a and 10b can be easily made to match, but I did not want to make the reader deal with the arithmetic issue of of whether theclique size is even or odd.

13) Show that there can be cliques in the model of a disk intersection graph such that no point in the model is covered by all circles in the clique.

14) Show that there can be an exponential number of maximal cliques in a disk intersection graph.

15) Show that the interval number of a complete bipartite graph with n vertices on each side is $\Omega(n)$; the precise formula is given in [83].

16a) Show that graphs with interval count at most cn are not hereditary.

16b) Show that graphs with interval count at most cn do not have an implicit representation.

17) exercises on relationship between PGs and 2D posets

18) Show that there are $2^{\Theta(n^2)}$ intersection graphs of paths on the grid.

19) Show that every graph has interval number at most n-1.

20) Show every poset has a dimension.

21) A Hiraguchi poset is a height 1 poset with k source vertices and k terminal

vertices, such that s_i has an edge to every terminal except for t_i. Show that the Hiraguchi posets have dimension k. In fact, every poset has dimension at most $n/2$, so these are extremal posets for dimension [].

22) Show that the number of minimal separators in a graph can be exponential.

23) Show that if S is a minimal separator of an interval graph, then every vertex of S passes through a common point on the line.

24) Show that every partial order is the containment graph of a star.

25) Show that G is the overlap graph of sets with an element common to all sets if and only if G is a co-comparability graph.

26) Show that each of the following classes of graphs is contained in the overlap graphs of subtrees of a tree.

a) chordal graphs
b)circle graphs
c) co-comparability graphs
d) spider graphs (i.e. intersection graphs of polygons inscribed in a circle)

27a) Show that connected graphs have a composition sequence.

27b) Show that forests have a composition sequence.

28) Show that graphs without C_4 or C_5 as induced subgraphs are closed under vertex expansion.

29) Show that any graph class closed under disjoint union has a composition sequence.

30) Show that comparability graphs and perfect graphs have intersection characterizations.

31) Show that the containment graphs of trapezoids with endpoints along two parallel lines are exactly equal to 4 dimensional partial orders.

32) A k-tree is a graph formed by starting with a clique of size k, and successively adding vertices which are adjacent exactly to a clique of size k-1 in the old graph. Show that every k-tree is chordal, but not every chordal graph is a k-tree.

33) Show that in every trapezoid graph, there is at least one pair of vertices x,y such that N(x) is a subset of N(y); this shows that trapezoid graphs are a subset of domination graphs, which have such a pair for every induced subgraph.

34) Give an example of a circle graph which is not an interval graph, and an interval graph which is not a circle graph.

4 Substitution Decomposition and Vertex Partitioning

This chapter deals with an operation which has proved to be useful in a number of algorithms for graphs with a variety of intersection models. The operation, called vertex partitioning, is used in finding the transitive orientation of a graph, recognizing permutation graphs, and finding the substitution decomposition of a graph, recognizing circular-arc graphs and trapezoid graphs, and other problems. We will try to clarify these various algorithms by focusing on the unifying idea behind them. We will discuss two different implementations of the partitioning algorithm, which yield time bounds proportional to n^2 and $m\log n$.

Partitioning is a fundamental algorithmic technique; an early survey of the field by Tarjan [164] mentions partitioning along with such methods as greedy algorithms, divide and conquer, etc., as a design method for algorithms. For some reason, partitioning has become much less well known as a technique than the other methods; it is very useful for dealing with geometrically defined classes of intersection graphs.

I should mention that Ross McConnell and myself [124] have recently discovered on O(n+m) algorithm for vertex partitioning, which speeds up all the algorithms described in this chapter. However, the new algorithm is currently too complex to describe easily, while I think that the reader can understand the versions described here. The first O(n^2) partitioning algorithms [152], [130] were similarly complex; I hope and believe that in a number of years the linear algorithm can be simplified as well.

We first define the operation which we call vertex partitioning. The name partitioning is chosen because of similarities to the set partitioning problem described in [10]. This operation has been called 'refinement' in previous papers by this author. In general, partitioning is a way of looking at operations on vertex sets, and this is often more efficient than thinking of operations on edges, as we will see for one important problem below.

Suppose that we are given a graph G = (V,E) and an initial partition P of V into disjoint blocks B_1, ..., B_k. The graph partitioning problem asks us to find the *coarsest* (having fewest blocks) partition of V, say P' = E_1, ..., E_j such that:

1) Each E_i is a subset of some B_h.

2) x, $y \in E_i$ implies N(x) - E_i = N(y) - E_i.

The second condition can be restated as follows. Each block E_i must be *indistinguishable* by any vertex x in V - E_i; i.e., x is either adjacent to every vertex in E_i, or x is adjacent to no vertex of E_i. For the reader familiar with the substitution decomposition, this corresponds to saying that each E_i is a module. A module is sometimes called an autonomous set, or a partitive set.

Before describing how to solve the partitioning problem, I would like to describe a typical use. We will first look at a sample application, to the transitive orientation problem, and then examine the principle behind the application. Later, we will go into this specific application in greater detail.

4.1 Transitive Orientation: Overview

In the transitive orientation problem, we are given an undirected graph $G = (V,E)$, and we want to assign a direction to each edge so that if $a \to b$, $b \to c$, then in the new graph $a \to c$. The classic algorithm for this problem works by orienting an edge, and then looking at which orientations are 'forced' from this. For each edge, you look at all neighbors, which leads to an O(nm) algorithm.

Using vertex partitioning, we take a different approach to orientation. Suppose that you have somehow oriented all edges which go between two sets of vertices X, Y in a manner consistent with a transitive orientation of G. Edges which are entirely within X, or entirely within Y, remain unoriented. Let x be any vertex of X. Vertex x is adjacent to some set of vertices Y1 from Y, and is nonadjacent to some set of vertices Y2 from Y. We can now orient all edges (y_1,y_2) which go between Y1 and Y2. If $x \to y_1$, the edge must be directed from y_2 to y_1, while if $x \leftarrow y_1$, then we must have $y_1 \to y_2$ to preserve transitivity. This leads to having all orientations between Y1 and Y2. The algorithm can continue in this manner until each block of the partition is indistinguishable.

In a general use of graph partitioning, you begin with an 'approximation' of your final solution, which completely specifies the relationship between vertices from different blocks of the partition, while leaving unspecified the relationship between vertices in the same block. In the case of transitive orientation, this relationship is the orientation of edges. While doing the partitioning, you achieve a better approximation of the solution by subdividing the blocks, maintaining the property that all relationships between vertices in different blocks are specified.

One difficult portion of the algorithms which use partitioning is to find the initial sets X and Y, such that all relationships between X and Y can be computed. This step is relatively problem specific, and is not a part of the vertex partitioning operation itself. Fortunately, for problems such as transitive orientation, finding the sets X and Y is not the bottleneck point of the algorithm.

Another question arises on what to do once the partitioning is finished. In general, some blocks can contain many vertices, and the relationships between vertices in the same block remains unspecified. The general approach in the algorithms described here is to rely on the substitution decomposition. If you are given the substitution decomposition of G, problems such as transitive orientation can be reduced to the transitive orientation within individual indecomposable factors of the decomposition tree. When the partitioning begins within any of these factors, it must end with all blocks consisting of a single node, so that all relationships between vertices will be specified.

4.2 Partitioning Algorithms

We now describe two algorithms for performing vertex partitioning. Algorithm A solves the vertex partitioning problem in O(n^2) time. We assume that G is stored as an adjacency matrix for algorithm A. A vertex v *splits* a block B when v is used to divide B into two subblocks, $B \cap$ N(v) and B - N(v). For each vertex v, we keep a list ACTIVE[v] consisting of vertices which are in different blocks from v, but whose blocks have not yet been split by v. We choose any vertex w in ACTIVE[v], and v splits the block containing w. The relationship between any pair of vertices x, y is examined twice; once when the block containing y is split by x (and y is removed from ACTIVE[x]), and once when the block containing x is split by y (and x is removed from ACTIVE[y]).

Algorithm A:

Partition (B_1,...,B_j);
for each block B_i do
for each vertex x in B_i do
ACTIVE[x] := V - B_i;
SPLITTER := V;
while (SPLITTER not equal $\emptyset$) do begin
v := any vertex in SPLITTER;
C := the block containing the first member of ACTIVE[v];
C_1 := C - N(v);
C_2 := C - C_1;
ACTIVE[v] := ACTIVE[v] - C;
if (ACTIVE[v] = $\emptyset$) then SPLITTER := SPLITTER - v;
for each vertex x in C_1 do ACTIVE[x] := ACTIVE[x] $\cup$ C_2;
for each vertex x in C_2 do ACTIVE[x] := ACTIVE[x] $\cup$ C_1;
if C_1 and C_2 are nonempty then SPLITTER := SPLITTER $\cup$ C;
end;

Algorithm B partitions a graph in O($m$log$n$) time. The algorithm chooses a vertex v which either i) has not been used previously as a splitting element, ii) is in a block B which is at most one half the size of the block which contained v the last time v was chosen as a splitting element, or iii) is in a block which will not be split further. Thus, a vertex v can be chosen as a splitting element at most log$n$ + 2 times. Vertex v is used to split every block which does not contain v.

The splitting caused by v can be performed in O(N(v)) time. Each block B is stored as a doubly linked list, and the number of vertices in the block is stored at a special node at the beginning of the block. Each vertex has a pointer to the front of its block. To split B, we first check that v is not in B; we then place neighbors of v in a new block B', which is kept linked to B until the end of the split caused by v.

If no vertex satisfies criteria i or ii for choosing a splitting element, any vertex from the largest active block C remaining will satisfy criterion iii. Every other vertex

v has tried to split C, since v was separated from C at some point, and is now in a block which is at most the size of C.

Algorithm B:

Partition(B_1,...,B_j);
for each block B_i do lastused[B_i] := $infinity$;
READY := B_1,...,B_j;
while (G is nonempty) do begin
while (READY not equal to $\emptyset$) do
begin
C := any block from READY;
lastused[C] := |C|;
for each vertex x in C do begin
split all blocks except C on their adjacency to x;
for each block B split into B,B2 by x do begin
lastused[B2] := lastused[B];
if |B| $\leq$ lastused[B]/2 then add B to READY;
if |B2| $\leq$ lastused[B2]/2 then add B2 to READY;
end
end
end;
C := the largest remaining block;
output C as part of the final partition;
use each vertex from C to split other blocks as before;
G := G - C;
end;

4.3 Substitution Decomposition

As was mentioned earlier, vertex partitioning is closely related to the substitution decomposition. Most of the uses of vertex partitioning for developing best known algorithms start by constructing a tree which represents the substitution decomposition. This tree can be constructed in O(m+n) time [123, 44, 48]. We will describe an O(n^2) algorithm, and an O($m$log$n$) algorithm.

The substitution decomposition of a graph can be defined as follows. A module is a set M of vertices such that vertices of V - M are adjacent to either all vertices of M, or no vertex of M. We decompose a graph recursively into subsets, each of which is a module of G. The procedure stops when all sets have a single vertex.

A module which induces a disconnected graph is called a parallel module. If the current set S is disconnected, partition S into connected components; make the decomposition tree for each connected component a child of a new internal node, which is labeled as a parallel module. If the complement of a module M is disconnected, M is called a series module. If the current set S is a series module, make the decomposition tree for each connected component of the complement of S a child of a new node, and label the new node as a series module.

Modules which are both connected and complement-connected are called prime modules. If S is a prime module, divide S into maximal proper submodules; exercise ? shows that every vertex of a prime module M is in a unique maximal proper submodule of M, so this decomposition is unique. The decomposition tree for each maximal proper submodule is made a child of a new node, which is labeled as a prime module.

Let us first note that there is an obvious polynomial time algorithm for finding the substitution decomposition of a graph. Whenever a graph or its complement is disconnected, decompose the graph into connected components in the graph or its complement. Assume the graph is connected. For each pair of vertices x, y, it is not difficult to find in O(m+n) time the smallest module which contains both x and y (exercise ?). Any time a module M is found, we can decompose (G - M) $\cup$ x, and replace x in this structure by a recursive decomposition of M. A simple analysis shows that we take O(n^2m) time on a single decomposition step, and thus O(n^3m) time to decompose G. This can be reduced by the observation that if x and y are not contained in a nontrivial module, they cannot be contained in a nontrivial module in any subgraph. Thus, no pair x, y needs to be tested more than once, which reduces the time complexity to O(n^2m) time.

The first O(n^3) algorithms for computing the substitution decomposition are [28] and [87]. We will describe an O(n^3) substitution decomposition algorithm, and modify this to get an O(n^2) decomposition algorithm. The first O(n^2) algorithm for this problem appears in [152]; we describe a simplification of the algorithm in [130].

Consider adding one vertex v at a time to the decomposition tree. If v is adjacent to 0 nodes in the tree, we add v either as a new child of the root if the root is a P node, or add a new root labeled P with v as one child and the rest of the tree as the other; the case where v is adjacent to all nodes of the tree is similar, with S replacing P in the description.

Otherwise, we start at the root. For each child c of the current node, we can decide whether v belongs in a submodule together with descendants of c in O(n) time, by taking any descendant x of c, and testing whether v has exactly the same adjacencies to nondescendants of c as x. If we find that v belongs in a module with a single child c, we recursively place v in c. If we find that v belongs in a module with at least 2 children of c, this must be an S or P node, and v is added as a new child of the node.

Suppose that v does not belong in a module with any single child of the current node p. If p is a series or parallel node, we have a number of simple cases. We treat only the case of a parallel node; the series node is handled symmetrically. Children of p which are completely nonadjacent to v are grouped under a single parallel node x_1, and children of p which are completely adjacent to v are grouped under a parallel node x_2. If no child of p is split by v, make a series node x_3 with v and x_2 as children, and make x_3 and x_1 children of p. If either x_1 or x_2 has exactly one child, replace that node by its child. If some child of p is split by v, we make x_2 and x_3 children of p, and continue to the procedure of the next paragraph.

Suppose that p is a prime node, or that v splits some child of p and does not belong in a submodule with any child. We make p a prime node with v as one child, and split each child c of the node as follows. First, mark all nodes in the subtree of descendants of c by whether all leaf descendants are adjacent to v, all nonadjacent to v, or whether some are adjacent to v while others are nonadjacent; this marking clearly can be done in time proportional to the size of the subtree rooted at c.

Traverse the subtree rooted by each child c of p in postorder. Do nothing if we reach a node which is completely adjacent to v or completely nonadjacent to v. Suppose we reach an internal node i that contains both neighbors and nonneighbors of v. If i is a prime node, make all remaining children of i children of p. If i is a series or parallel node, split i into a set of children which are completely adjacent to v, and a set which are completely nonadjacent to v. Place each set under a new node marked with the same label as i, and make these new nodes children of p.

Finally, we traverse the tree in postorder to remove internal nodes of degree 1; any such node encountered is replaced by its child.

Each vertex is added to the tree in $\mathrm{O}(n^2)$ time, so the algorithm takes $\mathrm{O}(n^3)$ time. After we decide where v is placed, we take $\mathrm{O}(n)$ time to construct the tree, since this is a simple postorder traversal on a number of nonintersecting subtrees of a node. Thus, we focus on the placement phase of the algorithm. To make our task simpler, assume that we have preprocessed the tree so that every internal node has a marker indicating whether descendants of the node are completely nonadjacent to v, completely adjacent to v, or are split by v.

Placing v in a parallel or series module is quite easy. We place v in a child of p if and only if v splits that child, and v is completely nonadjacent to all other children if p is a parallel module, and completely adjacent to all other children if p is a series module. Thus, the placement work takes time proportional to the number of children of p; the regrouping of children during the placement step is also clearly doable within this time bound.

However, placing children in a prime module is more expensive. If we compare v's adjacency list to the adjacency lists of all children of p, we take time proportional to the square of the number of children of p. For random graphs, the structure is very likely to be a single prime node with no submodules [127], so adding a vertex would take $\Theta(n^2)$ time. We need to keep information which distinguishes the adjacencies of different children of a prime node more efficiently. A complex data structure for keeping this information is described in [130]; the simplification here is due to McConnell.

For each pair of children c_1, c_2 of a prime node, we store a vertex which is not a descendant of c_1 or c_2, and which is adjacent to descendants of exactly one of c_1,c_2. Suppose that we have this information stored. We want to know which child v should be placed in, if any. Originally, we place all children in a set of candidates. While greater than two candidates remain, take any pair of candidates c_1, c_2, and let w be the vertex which is adjacent to descendants of exactly one of c_1,c_2. We remove whichever candidate disagrees with v on w from the candidate set. When we have one candidate c left, we check whether v agrees with c on all descendants of p which are not in c. If v disagrees with c on some such vertex, v is added as a new child of c, and v is added to c if the two agree.

The time taken to place c in p is proportional to the number of children of p, so c is placed in $O(n)$ time. Our only remaining task is to find vertices which distinguish pairs of children in a prime module. There is at most one prime module which has children added to it during the addition of v; this is the module with parent p in which v is placed as a child. We will show how to compute distinguishing vertices for children of p.

We start by choosing distinguishing vertices for v and each other child of p. When we removed each child c of p as a candidate module for containing v, we found a vertex w which could distinguish v from c; we store the value of w with c. Each node moved to be a child of p during the traversal of descendants of c is distinguished from v by w.

Now consider other pairs of nodes moved to be children of p by the addition of v to the graph. Note that if two vertices are in different submodules of a prime module, they will never be merged into a common child of p. Therefore, if a pair of vertices already has a distinguishing vertex, we do not need to update its value. Suppose that our traversal of descendants of c splits a parallel or series module M into a set S_1 of children which are completely adjacent to v, and a set of children S_2 which are completely nonadjacent to v; S_1 and S_2 are made children of p. We make v the distinguishing vertex for each pair $s_1 \in S_1$, $s_2 \in S_2$. If the parent of M is a series or parallel module, let w' be any descendant of the parent. We then climb the tree from M until we reach a prime module; let x be the next internal node encountered. If x has a different label from M, then all vertices in $S_1 \cup S_2$ are distinguished from descendants of x by any node from the other set (i.e. s_1 or s_2), and if x has the same label as M, then these vertices are distinguished from vertices in M by w'.

The time taken to find distinguishing pairs of vertices is proportional to the number of vertex pairs distinguished, even if we use a simple strategy such as traversing the entire subtree of x to find the descendants of x. Each pair of vertices is distinguished at most once, so the entire cost of finding distinguishing pairs is $O(n^2)$.

We note that the algorithm has been phrased as if we need distinguishing pairs of vertices for internal nodes. It is trivial to maintain a single leaf descendant with each internal node as we run the algorithm; since each internal node is a module, two internal nodes are distinguished by the distinguishing vertex for their leaf descendants.

We now present an $O(m \log n)$ algorithm due to McConnell for finding the substitution decomposition of a graph. We note that although this algorithm is much shorter to describe than the $O(n^2)$ algorithm given above, the proof of correctness is much less obvious. Readers interested in a correctness proof for this algorithm may find the ideas in [123, 124], each of which give variants of this algorithm modified to perform slightly different tasks.

This decomposition method is based on the O($m$log$n$) partitioning algorithm given above. The algorithm works in two phases. The first phase finds a subset of the modules of the decomposition tree using partitioning, and produces an ordering of the vertices so that all other modules will be found by choosing different vertices as our splitting elements.

We now describe the basic procedure. Whenever our partitioning process outputs a set S of size greater than 1, S is a module of G. We choose any vertex s from S, divide S into s and S - s, and restart our partitioning procedure. We keep a tree of all set partitions which occur during this procedure. Initially, the tree has a root, s as a left child, and V - s as a right child. Whenever a set is split into neighbors and nonneighbors, make a new internal node, with the neighbors as the left child, and nonneighbors as the right child. If S was found to be a module, mark the internal node corresponding to S. An example of the first phase is given in figure ? (NOTE: deferring figures here, since Ross has an alternative approach to this chapter which I am considering using).

We note that this procedure will find modules which do not contain the arbitrarily chosen vertex s, but will not find modules containing s. These will be found by the second phase of our procedure; the idea is to order the vertices so that we always choose splitting vertices for the second phase which do not appear together with s in a submodule. The ordering which makes this happen is the same as the ordering which chooses a source vertex in our transitive orientation algorithm, and is easy to construct from the partitioning tree. Number vertices during a standard postorder traversal, in which all descendants of the right child of a given node will get larger numbers than all descendants of the left node of the same child.

The second phase of the decomposition procedure is the same as the first, except that when we have a module S, we always choose a vertex x with highest number from S with respect to the ordering produced in the first phase, split S into x and S - s, and continue as before. If sets are stored as heaps, the total time taken finding highest numbered vertices is O($n$log$n$), so the two phases finish in O($m$log$n$) time.

The decomposition tree is produced by combining modules found during the first and second phases of the decomposition process; our choice of splitting vertices guarantees that all important modules are found during one of the two phases, though this is not obvious. First, we describe a procedure for 'cleaning up' the trees found during the first and second phases; this is necessary since series or parallel modules with many children will be arranged differently from their arrangement in the modular decomposition tree, as shown in figure ?. This rearrangement has two parts; adding a few simple modules which might not have been identified as such in the partitioning process, and consolidating chains of series or parallel nodes to a single level.

For each of the two trees constructed, look at each module M. If there is a child c of M which has no edges to any other children of M in G, remove c from M and add a node labeled P with c as one child, and M as the other. Similarly, if there is a child c' of M with edges to all other children of M, remove c' from M and add a node labeled S with c' as one child, and M as the other. This takes O(n+m) time, using techniques similar to those used in exercise ?.

We then consolidate the trees found in each of the two stages as follows. Traverse the tree in depth first search order. If we encounter a series or parallel node x, which has a parent p of the same type, make all children of x children of p.

Finally, we combine modules found during each of the two phases into a single decomposition tree. Let T1, T2 be the trees found during the two phases. Look at all nodes of T1 and T2 in nondecreasing order of number of leaf descendants; for example, all leaves are considered first. When a module M is encountered, we add M if M is not already part of the combined tree. To see whether a module M with k leaf descendants is already in the tree, it is sufficient to check whether the smallest numbered vertex in M is already in a module of size k. If we store the smallest numbered vertex in each module, and update a variable to i for a vertex v whenever v is found to be the smallest numbered vertex in a module of size i, this step takes constant time per module. Adding a module M to T is also easy. Each child c of M must occur somewhere in T, and for each root r of the current tree which contains a child of M, r becomes a child of M in the tree. Any natural method of marking a tree while finding roots so the same path is not traversed twice will allow us to combine two trees in this way in O(n) time .

4.4 Transitive Orientation

The following algorithm produces a transitive orientation of a graph, if such an orientation exists. We start by decomposing the graph with respect to the substitution decomposition.

Traverse the substitution decomposition tree in postorder; i.e. the children of x are visited before x is visited. When a node which represents a module M is visited, we orient all edges which have both endpoints in M.

If M is a parallel module, we do nothing; all edges in M have already been oriented. If M is a series module, we choose any linear order $c_1,\ldots,c_k$ of the children of M, and orient an edge from a vertex in c_i to a vertex in c_j if and only if i < j. If M is a prime module, we choose one vertex v_i from each child c_i of M, and find a transitive orientation T of the induced subgraph. We orient an edge from a vertex in child i to a vertex in child j if and only if $v_i \to v_j$ in T.

Since the least common ancestor of the two endpoints of every edge can be found in linear time [93], the problem can be reduced to the time to find the substitution decomposition tree, plus the time to orient the subgraphs induced by the prime modules. We note that this task can be easily performed in O(n^2) or O($m$log$n$) time, if simpler algorithms are preferred. To get an O(n^2) algorithm, traverse the tree in postorder, maintaining a list of all descendants of a node with the node. When an internal node is visited, we check each pair of vertices from different children, and assign any edges found to this internal node. We leave the task of finding an

O($m$log$n$) algorithm for this problem to exercise ?.

As we saw earlier in this chapter, if we are given two sets X and Y with a valid transitive orientation on the edges which go between X and Y, we can use vertex partitioning algorithm and maintain the property that all edges between blocks are consistent with a transitive orientation of G. The result of partitioning any subgraph induced by taking one node from each prime module must have one vertex in each block, so at the end of the partitioning process all edges will be oriented. The only remaining problem is to find the initial sets X and Y, together with a transitive orientation on the edges which go between these sets.

In [151], a complex O($n + m$) algorithm is described for this subproblem, but there is a very simple algorithm for finding a source s in transitively orientable graph which is prime with respect to the substitution decomposition in linear time. Once s is identified, we create partitions s and V - s, orienting all edges out of s, and run the graph partitioning algorithm.

Choose any vertex v as a start vertex. Create a set C of candidate source vertices, originally defined to be V - v. While more than two vertices remain in our C, choose any vertex w of V - C. If w has at least one neighbor in C, remove all nonneighbors of w from C. When C has one vertex remaining, make this vertex into a source s. The proof of correctness for this simple algorithm is left for exercise ?; since every adjacency list is scanned at most once, the time taken to find s is O(n+m).

While we are performing partitioning, it is easy to maintain a linear extension of the current orientation. Initially, we start with s in front of the block V - s. Suppose that a block of B of the partition is split into neighbors and nonneighbors of some vertex w. If w occurs before B in the ordering, place $B \cap$ N(w) after B - N(w); place these in the reverse order if w occurs after B.

Suppose that we want to perform transitive orientation on the complement of G rather than G itself. If we take the complement of G and perform transitive orientation, we clearly pay a $\Theta(n^2)$ price. However, the steps of finding a linear extension for the complement involve repeatedly dividing sets into neighbors and nonneighbors. These steps can be done by looking at adjacency lists in G rather than adjacency lists in the complement. In fact, the algorithm of [124] finds a linear extension of a transitive orientation either of G or its complement in linear time, and this can be be used to recognize a number of graph classes efficiently.

The procedure described in this section finds an orientation even if no transitive orientation is possible. However, the orientation produced is transitive if G is a comparability graph. If we must check whether the orientation is transitive, the best known method is to compute A^2, where A is the adjacency matrix of A, to see whether this is equal to A. The current best matrix multiplication algorithm takes O($n^{2.376}$) time; thus, this is the best known bound for comparability graph recognition. The open problem of recognizing comparability graphs more efficiently is discussed in chapter ?; the difference in time complexities between the verification procedure and the orientation procedure has been a great source of inspiration and frustration to me since I first designed an algorithm for efficient orientation many years ago. In the next section, we see examples of problems for which the orientation can be checked in a different way, and thus algorithms which use transitive orientation can take less time than comparability graph recognition. More examples of such uses of transitive orientation appear in chapter ?.

We will also discuss briefly an alternative method for verifying transitivity; this was first communicated to me by McConnell, though the idea is from []. It is well known that the constants involved in the $O(n^{2.376})$ matrix multiplication are very large; [] feels that no algorithms yet developed which are faster than $O(n^{2.79})$ will ever be practical. We will discuss a method which is good for verifying sparse graph transitivity. It is trivial to verify transitivity in $O(nm)$ time, by doing a graph traversal from each vertex; we will show how to improve this to $O(m^{3/2})$, which is better for sparse graphs.

The technique is very simple. For each edge $x \to y$ in the transitive orientation, we look at all edges from y; whenever $y \to z$, we verify that there is an edge from x to z, and declare that the orientation is not transitive if we find some violation. The amount of work is clearly proportional to the number of triangles in G.

It is not so obvious that the number of triangles in a graph is $O(m^{3/2})$. For any triangle x,y,z, assign a cost of 1 to the edge between the two smallest degree vertices, where ties are broken arbitrarily. We will see that every edge accumulates $O(m^{.5})$ charges, so that the total number of triangles is $O(m^{1.5})$.

Consider any edge x,y. Each charge assigned to this edge corresponds to a distinct neighbor z of x such that the degree of z is at least as large as the degree of x. If x has degree at most $m^{.5}$, there cannot be more than $m^{.5}$ such vertices z. On the other hand, there are $O(m^{.5})$ vertices in a graph of degree greater than $m^{.5}$ (exercise or say since sum of all degrees is $2m$). Thus, if x has degree at least $m^{.5}$, x,y also accumulates $O(m^{.5})$ charges.

4.5 Permutation Graphs and Two Dimensional Partial Orders

The transitive orientation algorithm of the previous section immediately gives an $O(n^2)$ algorithm for permutation graph recognition and two dimensional partial order recognition. If we use the linear time algorithm for finding a transitive orientation, the time bound for each of these problems becomes linear. We will describe the algorithm for two dimensional partial order recognition.

A partial order represented by the transitive graph G is two dimensional if and only if $\bar{G}$, the complement graph of G, has a transitive orientation. If $\bar{G}$ has a transitive orientation, then a realizer for G can be constructed by adding G and a transitive orientation T of $\bar{G}$ to get the first list, and adding G to the reverse of T to get the second [79]. We can then check whether the two lists represent G by checking whether the relative position of each pair of vertices x, y in the two lists represents the adjacency of x and y correctly. We note that if the original graph G is not transitive, then one of the two lists created is not a linear order; this can be tested easily. To test whether an undirected graph G is a permutation graph, we transitively orient G and test whether the result is a two dimensional partial order.

The procedure described above requires $\Omega(n^2)$ time, since it involves taking the complement graph. Suppose that we have a transitive directed graph G, and a linear extension L of the transitive orientation of $\bar{G}$. It is easy to place each vertex v in the correct position of the two lists in time proportional to $|\mathrm{N}(v)|$, which means that the lists can be constructed in $\mathrm{O}(m+n)$ time.

Let $out_>(v)$ ($in_>(v)$) be the number of outneighbors (inneighbors) of v which come after v in L, and let $p(v)$ be the position of v in L. In the first list, v comes before all outneighbors of v, and all vertices after v in L which are neither outneighbors nor inneighbors of v in G. In the second list, v comes before all outneighbors of v, and all vertices before v in L which are neither outneighbors nor inneighbors of v in G. Since these positions can be computed easily from the variables $out_>(v)$, $in_>(v)$, the number of inneighbors and outneighbors of v in G, and $p(v)$, we can find the position of all vertices v in the two lists in linear time.

We can then check whether the two lists represent G correctly in $\mathrm{O}(n+m)$ time (exercise ?). Thus, given linear time algorithms for finding linear extensions of a transitive orientation for G and $\bar{G}$, two dimensional partial orders and permutation graphs can be recognized in linear time. This also allows us to construct a representation as containment of intervals in linear time, as described in chapter ?. Although the linear time algorithms are difficult at this time, the $\mathrm{O}(n^2)$ and $\mathrm{O}(m \log n)$ are described completely above.

4.6 Other Uses of Partitioning

The transitive orientation algorithm itself is useful for a number of other problems. Linear time transitive orientation for G and $\bar{G}$ gives linear algorithms for a number of optimization problems on comparability graphs (chapter ?), and linear time recognition of cointerval graphs (chapter ?). Recognition of a number of other graph classes reduces to recognition of permutation graphs; some of these are discussed in chapter ?.

Partitioning is used in recognition algorithms for circle graphs and circular-arc graphs ([59] and chapter ?). For these classes, and other classes with geometric models, we find an approximate geometric layout, and use partitioning to force correct intersections between sets.

Partitioning variants are used to decompose graphs with respect to the join decomposition [120] and to produce an ordering for chordal bipartite graphs. The chordal bipartite application is described in chapter ?. We present a join decomposition algorithm in chapter ?, which is slower than the approach based on partitioning. In that algorithm, we test each edge e of a spanning tree to determine whether there is a split in G such that e; each edge is tested in $\mathrm{O}(n+m)$ time. Since there are n edges of the spanning tree, we need O(n) tests, and thus require O(nm) time even if G has no split. A partitioning based approach tests whether a pair x,y belongs on the same side of a split, and creates successively finer approximations of what must be on the other side of the split. Although a single test of this form takes $\Theta(n^2)$ time, the fact that in any trio of vertices some pair must be on the same side of a split can be used to reduce the overall complexity from O(nm) to O(n^2); the algorithm itself can be found in [120]. This approach came from thinking of operations on sets of vertices rather than edge operations; this idea was also important in developing the transitive orientation algorithms given earlier.

The problem of recognizing comparability graphs without using matrix multiplication has been of interest to me for a long time. The following natural approach, designed by myself and Gustedt, may be of interest; note, however, that this problem is as hard as determining whether a graph is triangle free (exercise ? chapter ?)

Assume G is prime; otherwise, each prime component of G is tested separately. Use partitioning to create an orientation G' of G. Create a different orientation G'' of G, by randomly making other choices each time we have a choice of splitting element during partitioning. If G' and G'' are not identical up to reversal, answer that G is not a comparability graph. If we repeat this some number of times and always reach the same orientation, answer that G is a permutation graph.

Problem ? shows that if G is not a comparability graph, there are choices which make G' and G'' different, but it may be the case that some orientations occur with very high frequency. Therefore, to make this algorithm efficient, we need to ask whether there is a small function f(n) such that if G is not a comparability graph, after f(n) steps, the probability that we will find two different orientations is greater than 1/2. Alternatively, it would be interesting to exhibit a class of graphs for which the expected number of steps before finding a violation is $\Omega(n)$. As a weaker first step involving two notions of randomness, can we show that this algorithm is efficient in the randomized sense for proving that a random graph with fixed edge probabilities is or is not a comparability graph?

4.7 Exercises

1) Give an O(n+m) algorithm which finds the smallest module containing both x and y, for any x and y.

2) Show that the source finding algorithm correctly finds source in a comparability graph (hint of considering first vertex removed which has edge to s?).

3) Is the following property maintained during the transitive orientation algorithm? Between any pair of sets, all edges between the sets are oriented in the same direction.

4) Can there be graphs (obviously not comparability graphs) for which the orientation algorithm can produce a cycle?

5) Show maximal proper submodules of a prime module unique.

6) Show how to verify that k = two lists represent G in O(n+m) time (this problem is open for larger values of k).

7) Give an O($m$log$n$) algorithm which takes a decomposition tree and a set of m edges from a graph on n vertices, and assigns each edge (x,y) to the least common ancestor of x and y in the tree.

8) It is well known that a two dimensional partial order which is prime with respect to the substitution decomposition has a single two dimensional realizer. Show that in general, a prime partial order can have exponentially many realizers of minimum size.

9) Suppose that you are given a decomposition tree, but are not told whether internal nodes correspond to prime nodes, series nodes, or parallel nodes. Show how to determine the type of module for each internal node in O(n+m) time. You might find it useful to show that the total number of edges in all representative graphs for modules is O(m).

10) Suppose that you are given 1 list L of vertices in a partial order, and you want to determine whether this is part of a realizer of size 2. Design a linear time algorithm to solve this problem. It is an open question to determine whether L is part of a realizer of size 3 in polynomial time.

11) Give an algorithm which finds the transitive reduction of a two dimensional partial order in O(n^2) time.

12) Given 2 lists, show how to construct the adjacency list for the corresponding 2 dimensional partial order in O(m+n) time. Note: this is open for dimension $d > 2$; the best known seems to be ???

13) A fascinating theorem due to Schnyder [146] shows that a graph G is planar iff the vertex/edge inclusion poset has dimension at most 3. Show that graphs with vertex/edge inclusion posets having dimension 2 are (exactly?) forests.

14) Show that the algorithm which uses pivoting on the smaller set can have Ω($m$log$n$) time complexity, i.e. that there are graphs for which this actually takes time proportional to $m$log$n$.

15) Show that the pivot on the smaller set algorithm is also an O(n^2) algorithm.

16) Show that if G is not a comparability graph, it must be possible for the partitioning algorithm to create two orientations of G which are not the reverse of each other.

5 Geometrically Defined Classes of Graphs

This chapter deals with a very natural method for defining a class of graphs which gives us problems when we try to find an implicit, or other space efficient, representation. We will talk about defining graph classes using simple geometric objects such as circles. Although circles can be represented using a small number of variables corresponding to the center and radius of the circle, these variables do not naturally correspond to small integer values. We will discuss the problems of counting, representing, and recognizing graph classes of this type, and point out some of the better known classes for which the problem of efficient representation remains open.

In a previous chapter, we discussed classes of graphs which correspond to intersection graphs and containment graphs of boxes in 2 and other k-dimensional space, where boxes are aligned with the coordinate axes. It is at least as natural to consider the intersection graphs and containment graphs of circles, disks, and k-dimensional spheres. The reader may feel differently, but I see no particular reason to believe that either class of graphs would be more complex than the other, if we view them as a set of graphs which happen to have mappings to boxes and spheres. If we think about efficient representation, however, the problem of representing circles and disks is much more complex than the problem of representing boxes.

The best class to use for illustrating the issues and techniques involved has been called the class of circle intersection graphs, though the term disk intersection graphs may be more precise in that if one circle completely encloses the other, we say that the circles do have a nonempty intersection. Let us briefly consider the obvious method of storing a representation; for each vertex, we store an x coordinate, a y coordinate, and the radius of the corresponding circle. If all these values are integers in the range $0..n^k$ for some value of k, then this is an implicit representation. However, these values are not necessarily even rational, and it is not clear at all that there is any grid size large enough so that we can use integer values for all centers and radii. In the previous chapter, the reason we know that the graph classes have $2^{O(nlogn)}$ members has nothing to do with studying the class of graphs; instead, the O($n$log$n$) bit representation itself tells us that the number of graphs in the class is $2^{O(nlogn)}$. For circle and disk intersection graphs, we do not know any efficient representation, so it is not clear that the graph class is even small enough to hope for an implicit representation. In the next section, we will describe a powerful counting technique which enables us to prove that the number of disk intersection graphs, as well as many other geometrically described classes of graphs, is $2^{O(nlogn)}$. However, no good representation is known, despite the fact that disk intersection graphs are clearly a hereditary class of graphs, leading to the following open problem.

Open Problem: Is there an implicit representation of the disk intersection graphs?

The same problem is open for other classes, such as containment graphs of circles (more commonly known as circle orders; might as well introduce and discuss causality), intersection graphs of lines in general position on the plane, and many other classes.

5.1 Warren's Theorem

Warren's theorem is the key mathematical tool which allows us to put good upper bounds on the number of graphs in many geometrically defined classes. Discussing the history of Warren's theorem may put the author in the strange position (for an author of a mathematical text, at least) of being on the fringe of debates about political correctness [2]. Warren's theorem was published in 1968 in a paper entitled 'Lower Bounds for Approximation by Nonlinear Manifolds' as a way of obtaining a bound on how closely polynomials can approximate more general subspaces. Given the subject area, the paper was completely unknown to those of us working with graphs. Warren's paper was later discovered by Alon in the 1980s, who realized that this gave a tool for counting the number of posets such that elements can be described using a fixed number of variables. The same tools have been used to count many other classes of graphs, and it often can give good bounds quite simply. However, the technique is still unknown to many of us in the field, since it does not appear in any book at the time of this writing, and one still finds papers which ask whether graph class A is a subset of graph class B, when class B can be shown to be much smaller than class A using Warren's theorem. I would like to add that most of us, including myself, would not be able to bound the numbers of graphs in the classes without using Warren's theorem, making Warren's theorem one of the deepest and most important results which we will discuss in this book.

Warren's theorem is stated in terms of the number of sign patterns of a set of polynomial equations. A sign pattern is a sequence of +s and -s, corresponding to whether each polynomial equation is nonnegative or negative. For example, suppose that we are given three polynomial equations $5x^2$-$7x$+2, $3x$-9, $4x^2$+$13x$-124. One possible sign pattern for this set of equations is +,-,-, since if $x = 2$ the first equation evaluates to a positive number while the second and third are negative. The number of sign patterns is clearly bounded above by 2^k, where k is the number of equations, but the number is not in general nearly as large as 2^k. In our example, it is easy to see that no sign pattern for this set of equations can begin -,+; if the second equation evaluates to a positive number, then $x > 3$, but it is easy to see that this implies that the first equation evaluates to a positive number. This argument, however, involves looking at the particular set of equations, and says nothing about the number of sign patterns with respect to a general set of polynomial equations.

Before stating Warren's theorem, let us define some parameters which affect the number of possible sign patterns. The number of variables in the set of polynomial equations will be n, and the maximum degree of any individual polynomial equation is d. The total number of equations will be denoted as m.

[2] In late 20th century America, there has been an odd point of political contention over the statement 'Columbus discovered America'. How could he have discovered it, says one side, when the Native Americans already lived there? However, I would say that Noga Alon discovered Warren's theorem, in the sense of bringing it into the realm of knowledge of the graph theory community, even though it was clearly already known to others, including Warren; it is in this sense that Columbus 'discovered' America.

Warren's Theorem [174]: The number of possible sign patterns is at most $(cdm/n)^n$, where $c = 4$e.

We will make no attempt to prove Warren's theorem here. Typically, we might have a constant number of variables for each vertex, and a single equation to determine adjacency between a pair of vertices. Sometimes, as for visibility graphs, we need n^i equations to test adjacency between a pair of vertices. In any case, Warren's theorem tells us that the number of sign patterns, and hence the number of graphs, is at most $(n^k)^{jn}$ for some constants j and k, which is $2^{O(nlogn)}$. This will give a method for showing that a graph class is small without giving an efficient representation.

Let us see how Warren's theorem can be used to show that the number of graphs in a class is small. We will begin with the disk intersection graphs. For each vertex v, we have three variables v_x, v_y, v_r, which represent the x and y coordinates of the center and radius of the circle which corresponds to v. We now want to write a polynomial in these variables which changes signs depending on whether the corresponding circles have a nonempty intersection. For this class, creating such a polynomial is not difficult. For each pair of vertices v, w, we have

$$(v_x - w_x)^2 + (v_y - w_y)^2 \text{ - } (v_r + w_r)^2$$

The first two terms of the equation give the square of the distance between the centers; the circles intersect if and only if the sum of the radii is larger than the distance between the centers, so this equation is less than 0 if and only if the circles have a nonempty intersection. This means that if you give me the sign of the result of each equation, I can reconstruct the graph. Since the number of sign patterns is $2^{O(nlogn)}$ by Warren's theorem, the number of circle graphs is $2^{O(nlogn)}$.

No method is known for constructing implicit representations of graph classes which can be shown by Warren's theorem to be sufficiently small so that these might exist. Any such constructive proof would be a major breakthrough.

Alon and Scheinerman [11] characterize classes of partial orders in terms of 'degrees of freedom', which is essentially the number of variables per vertex in the polynomials which describe graphs in the class. This is used to count the number of orders in the class. We could define similar notions of graph classes.

Call a graph class 'Warren representable' if for all n, there is a set of $O(n^k)$ polynomial equations on $O(n)$ variables such that for every graph G in the class, adjacencies in G can be determined from the signs of the polynomials. We could make a conjecture similar to the implicit graph conjecture, that every hereditary class of graphs with $2^{O(nlogn)}$ members is Warren representable. In general, there is no known containment relationship in either direction between implicitly representable graphs and Warren representable graphs.

Note that showing that a graph class is Warren representable is somewhat less useful than showing that it has an implicit representation, since Warren's theorem is a counting tool while implicit representations are useful for fast storage and adjacency testing. Nevertheless, the Warren representable graphs pose a major challenge for the implicit graph conjecture. A variety of Warren representable graph classes are known which are not known to have an implicit representation; these will be discussed at various points in this book.

5.2 Continuous Nongeometric Variables

The size of a number in a representation can also pose a problem for classes of graphs which are not defined geometrically. One of the more natural classes for which this problem arises is representing a graph using a dot-product representation. Dot-product representations were introduced in [64].

A k-dot product graph is a graph in which each vertex can be mapped to a set of k real numbers, where x and y are adjacent if and only if the dot product of the corresponding numbers (that is, the sum of the values obtained by multiplying the ith number for x by the ith number for y) is at least 1.

Since every graph is a k-dot product graph for some value of k (exercise ?), there is also a natural notion of dot-product dimension; the dot-product dimension of G is the smallest k such that G is a k-dot-product graph.

If every number is a small integer, a k-dot product representation is clearly implicit. From the definition, however, numbers are not necessarily even rational, much less small integers.

Warren's theorem applies trivially to dot product representations, since each adjacency test is defined by testing whether a polynomial on a set of $2k$ variables is at most 1; subtracting 1 clearly makes the adjacency of two vertices depend only on the sign of the corresponding polynomial. Therefore, for any fixed k, the number of k-dot product graphs on n vertices is $2^{O(nlogn)}$, but it is not clear that any dot-product representation is space efficient.

[64] were able to prove that numbers in the dot-product representation may be restricted to be rational without loss of generality, and can claim a finite bound on the number of digits. However, this bound is quite high; the number of digits in the representation is not known to be polynomial, making membership in NP an open problem even for 2-dot product representable graphs. Thus, finding an implicit representation for k-dot product graphs is an open problem. We state a few variants of this problem, mostly chosen from a larger set of questions in [64], which are not equivalent but which may be related to each other. The problems seem to be open for any fixed $k > 1$; dot-product dimension 1 graphs are equal to threshold graphs, which have nice representations for a variety of reasons.

EARLIER sec on threshold G?

Implicit Representation: Find an implicit representation for k-dot-product representable graphs.

Bound the Representation Size: Can all k-dot product representations use numbers which are bounded by $O(log^i n)$ bits, or failing this a polynomial number of bits?

k-dot-product graph recognition: Can you determine whether a graph is a k-dot product graph in polynomial time?

k-dot product graph optimization: Are there NP-complete problems which are polynomially solvable on k-dot-product graphs? This problem may be interesting both if a k-dot-product representation is given, and if it is not; see chapter ? for more on optimization.

A variety of graph classes in which vertices correspond to intervals and there is a 'tolerance' value associated with each interval have been defined, where vertices x,y are adjacent if the overlap length exceeds some function on the tolerances of x and y. These classes are all open for recognition and implicit representation, and can be shown to have $2^{O(nlogn)}$ members; the most popular of these classes is discussed in exercise ?.

5.3 Decidability Results

Many of us are used to the notion that all problems we are likely to encounter are at least decidable. This is probably reinforced by our teaching of NP-completeness as a practical notion of difficulty, relegating decidability problems to a more theoretical course in automata theory.

For these natural recognition problems, the decidability question is by no means obvious. We certainly cannot step through, for example, all possible placements of circles on the plane to decide whether a graph is a disk intersection graph. Just as Warren's theorem can be invoked to show a bound on the number of graphs in a large number of geometrically defined classes, we will present a theorem due to Tarski which will show that the recognition problem on many of the same classes is decidable. Often, this is the only positive result we have on recognition of these classes.

We will state a limited form of Tarski's theorem, which is sufficient for showing that the geometrically defined classes which are defined in this book are decidable.

Consider any finite set of polynomial equations, inequalities, and 'inequations' (i.e., the polynomial function should be not equal to a value rather than equal). It is possible to decide in a finite number of steps whether there are real numbers which satisfy all of these.

Thus, there is an algorithm to determine (for example) whether a graph is a disk intersection graph; we can write polynomial inequalities in the model which corresponding to adjacencies in the graph.

Unfortunately, the methods for determining whether there is a solution are doubly exponential, so that these general techniques are not in general able to produce practical algorithms for recognizing geometrically defined classes of graphs.

5.4 Exercises

1) Consider the class of intersection graphs of a set of lines in the plane, when the lines can take arbitrary positions (i.e. these are not restricted to be parallel to the x and y axes). Show that this class has $2^{O(nlogn)}$ members.

2) One can define a partial order on a set of angles in the plane by $x \rightarrow y$ if and only if the portion of the plane inside of the angle defined by x contains the portion of the plane inside the angle defined by y. P is an angle order if and only if it corresponds to a partial order for some set of angles on the plane. Show that the number of angle orders is $2^{O(nlogn)}$.

3) Show that the number of circle orders is $2^{O(nlogn)}$.

4) Show that every graph is a k-dot-product graph for some value k.

5) A tolerance graph is defined by a set of intervals, and a 'tolerance' associated with each interval, which can be an arbitrary number. Two vertices x,y are adjacent if the length of the overlap between x and y is greater than the tolerance of x or is greater than the tolerance of y. Show that there are $2^{O(nlogn)}$ tolerance graphs. Recognizing tolerance graphs and finding an implicit representation for this class are open questions.

6) It has been shown that tolerance graphs are a subset of weakly chordal graphs. Another subclass of weakly chordal graphs has been called 'domination graphs'; G is a domination graph if every induced subgraph H of G contains a pair of vertices x, y such that $\mathrm{N}(x)$ is a subset of $\mathrm{N}[y]$ in H.

a) Show that there are domination graphs which are not tolerance graphs.

b) Show that every tolerance graph is a domination graph.

7) A graph is a bounded tolerance graph if it has a tolerance representation in which the tolerance of every interval is less than the length of the interval.

a) Show that every bounded tolerance graph is a trapezoid graph.

b) Show that there is a tolerance graph which is not a trapezoid graph.

6 Classes Which use Global Information

In this chapter, we will study representations of graph classes which have efficient representations with regard to space complexity, but which use global information for adjacency testing. Typically, adjacency testing using natural storage methods will be slow when compared to testing in classes with efficient representations. We will show how to produce implicit representations for these classes, when such a construction is known. Visibility representations also are examples of graph classes which use global information; visibility graphs are covered in chapter 7. Distance hereditary graphs also have a characterization which uses a small amount of space but global information; see chapter 12.

6.1 Paths in Trees

There are a number of classes of graphs defined using paths in a tree. These are probably the clearest examples of graph classes defined with a natural $O(n \log n)$ bit representation, but adjacency testing is not local (or efficient) if this representation is used.

These classes can all be described loosely as the intersection graphs of paths in a tree, with variations coming from alternative definitions of trees, and intersection of paths. Trees can be interpreted as directed or undirected graphs; in a directed tree all paths would be on a single path from the root, while in an undirected tree a path may not be contained on a simple path from the root. Different notions of intersection arise depending on whether we count two paths which cross at a vertex as intersecting, or whether we require them to share an edge; this distinction comes from viewing a path as a sequence of vertices or as a sequence of edges.

We can find implicit representations of all these classes using techniques described in chapter ?, when we introduced intersection graphs of paths in trees. In all cases, we can show that the tree can be assumed to have size $O(n)$, and we can determine the adjacencies of vertices by storing enough information to compute ancestor information, and keeping a small set of vertices for each path. The different variants are explored in the exercises. Note that containment graphs of paths in a tree can be handled similarly; see problem ?.

(other algorithms on these classes?)

(dimension vs containment of paths in tree open?)

6.2 Chordal Comparability Graphs

One of the organizing themes for Golumbic's excellent book [79] is whether a graph G and/or its complement is chordal, and whether it or its complement is a comparability graph. Thus, there are chapters on comparability graphs, chordal graphs, split graphs (= chordal and co-chordal graphs), permutation graphs (= comparability and co-comparability graphs), and interval graphs (= chordal co-comparability graphs). This leads to good characterizations of almost all graph classes defined by a pair of properties from the set chordal, co-chordal, comparability,co-comparability; note that the complement of a graph class, such as the co-chordal comparability graphs, are well characterized as the complements of interval graphs. The only pair of properties from this set which is not dealt with in Golumbic's book is the class of chordal comparability graphs, and their complements.

First, we will show that the number of chordal comparability graphs is $2^{O(nlogn)}$. We will show this by giving a natural O(nlogn) bit representation for any graph in the class. However, the representation is not implicit, because it uses global information to test adjacency. We will then show that chordal comparability graphs are contained in a class which has an implicit representation, which gives an implicit representation for chordal comparability graphs. The representation is not entirely satisfactory; we will see that it does not characterize chordal comparability graphs, and why this is intuitively unsatisfying. The problem of creating a natural implicit representation which exactly characterizes the chordal comparability graphs remains open; see open problem ?.

The first procedure for finding an O($n$log$n$) bit representation of a chordal comparability graph G is not difficult to understand. Choose any transitive orientation G' of G, and find the transitive reduction G'' of G'. We will show that G'' contains at most 2(n-1) edges. G'' is an O($n$log$n$) bit representation of G, since G' can be reconstructed from G'' by using a transitive closure algorithm, and G can be derived from G' by removing directions from all edges.

Theorem ?: Let G be a chordal comparability graph, G' be any transitive orientation of G, and G'' be the transitive reduction of G'. Then G'' contains at most 2(n-1) edges.

Proof: Consider a perfect elimination ordering of G. Add vertices in the reverse order of the perfect elimination scheme; this guarantees that the next vertex v added is always simplicial in G. Let G_1 be the transitively reduced graph before v is added, and G_2 be the transitively reduced graph after v is added. Any edge of G_2 - G_1 must have v as one endpoint; if an edge was redundant without v in the graph, it remains redundant with v in the graph.

There can be at most one inedge to v and at most one outedge from v in G_2. Suppose v has an edge to x and y in G_2. Since v is simplicial, x and y must be adjacent; without loss of generality, assume that there is an edge from x to y in G'. Since v has an edge to x and there is an edge from x to y in G', there is no edge from v to y in G''.

Since a 1 vertex graph has no edges, and each vertex added adds at most 2 edges, the number of edges in G'' is at most 2(n-1). QED

We will use the intersection representation of chordal graphs from chapter ? to show that chordal comparability graphs are contained in the comparability graphs of 4 dimensional posets. Thus, a chordal comparability graph can be stored by storing with each vertex x the position of x in all linear extensions of a transitive orientation of G, such that x and y are adjacent if and only if the for one of the vertices the ith number stored at that vertex is smaller than the ith number stored at the other vertex for i=1,2,3,4.

Theorem ?: Every chordal comparability graph is the comparability graph of a 4 dimensional poset.

Proof: Let P be any transitive orientation of G; we will show that P has dimension at most 4. Suppose that we are given a representation of G as the intersection graph of subtrees of a tree T, and let T_i denote the subtree of T which corresponds to vertex i of G. Let DFS(T) be an arbitrary depth first search ordering of the vertices of T.

To see how the appropriate linear extensions are constructed, we must think of how subtrees corresponding to nonadjacent vertices u, v of G could be traversed. Without loss of generality, assume that the first vertex of T_u is visited before the first vertex of T_v during DFS(T). We say u and v have a type 1 relationship if T_u is traversed entirely before T_v, and that they have a type 2 relationship if we enter T_u first, and at some point leave T_u and traverse all of T_v before reentering T_u. We design linear extensions which reverse the orders of u and v for each of the two types of relationships.

Our linear extensions of P work as follows. Adjacent vertices of G must occur in the order defined by P. L1 orders type 1 nonadjacent vertices of G on the basis of which subtree is encountered first in DFS(T), while L2 orders type 1 nonadjacent vertices on the basis of which is encountered last in DFS(T). L3 orders type 2 nonadjacent vertices on the basis of which is backed up from first during the depth first search of T, while L4 orders nonadjacent type 2 vertices on the basis of which vertices are backed up from last in T. In all cases, we choose any linear extension of P plus the required orderings to complete our linear extension. More formal definitions for this construction can be found in [117].

In the graph of figure ?, the pairs (b,f),(b,d),(c,d),(d,e),(d,f),(e,f),(f,g) have type 1 relationships; (a,b),(b,c) have type 2 relationships. One possible ordering of the lists is L1 = a d b c e g f (note that b may precede c even though c is encountered before b, since (b,c) is not a type 1 relationship), L2 = a b c f e d g, L3 = b a c f d e g, L4 = a c b d e f g.

If x is adjacent to y, then there is an edge (w.l.o.g) from x to y in P, and x comes before y in all extensions. If x and y are nonadjacent, either T_x is visited entirely before T_y, in which case x comes before y in L1 while y comes before x in L2, or (w.l.o.g) T_y is visited in the middle of the traversal of T_x, in which case x comes before y in L3 while y comes before x in L4. Therefore, all adjacencies are represented properly. We need to prove that L1, L2, L3, and L4 are valid linear extensions of P, and we will have shown that P is 4 dimensional.

Since every pair of vertices is ordered in L1, L2, L3, and L4, we need to show that there is no cycle in any of the orderings. The proofs that L1 and L2 have no cycles are essentially the same, as are the proofs that L3 and L4 have no cycles, so we will

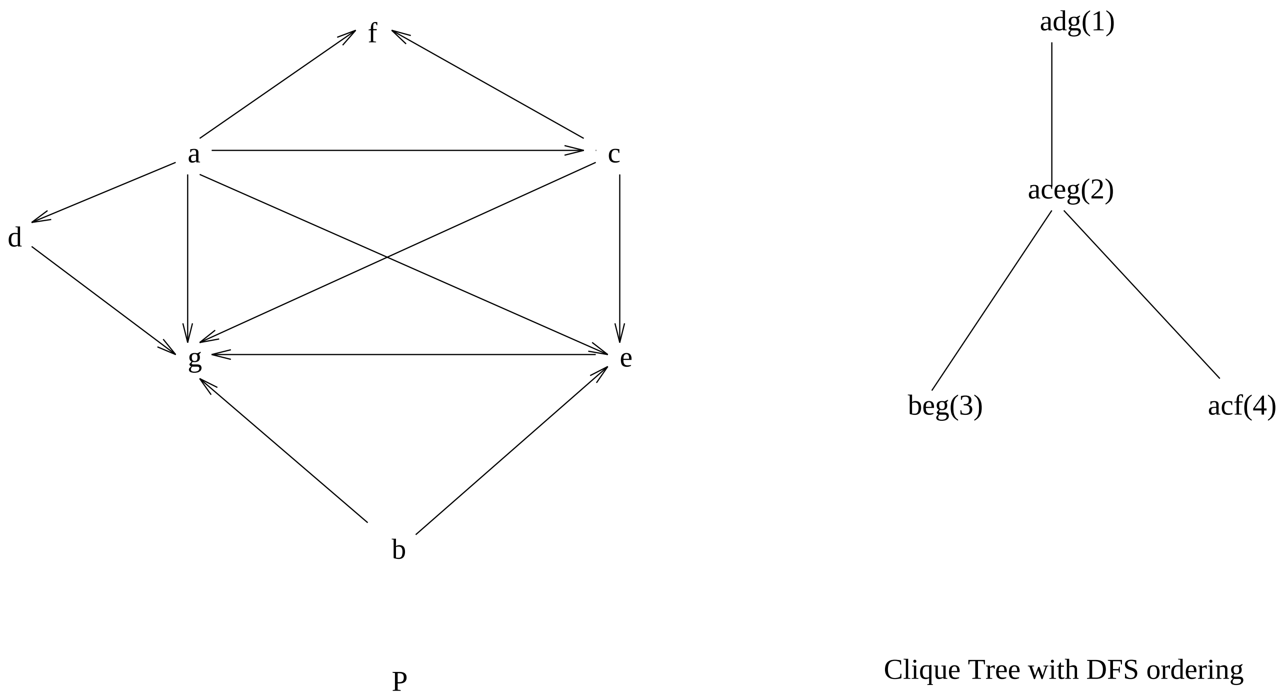


Figure 5: Chordal Comparability Graph

prove that L1 and L3 are acyclic.

Claim: L1 is acyclic.

Proof: Let C be a shortest length cycle in L1. C cannot contain two consecutive edges from P, since P is transitive. C cannot contain two consecutive edges $v \to w \to x$ from E - P, since this would mean that v is visited entirely before w and w is visited entirely before x, so C would have a chord $v \to x$. This implies there are edges $v \to w \to x \to y$ of C such that $v \to w$ and $x \to y$ are in E - P while $w \to x$ is in P. By the definition of L1, T_v is traversed entirely before T_w, and T_x shares a node of T with T_w, which implies that the last vertex of T_x is visited after the last vertex of T_v. Since $x \to y$ is in E - P, the first vertex of T_y is visited after the last vertex of T_x, which implies that T_v is traversed completely before T_y is visited. This creates a chord $v \to$ y, contradicting our assumption that C is a minimum length cycle. QED

Claim: L3 is acyclic.

Let C be a shortest length cycle in L3. C cannot contain consecutive edges from P, since P is transitive. Suppose that $w \to x$ and $x \to y$ are in E - P. By the definition of an edge of E - P, DFS(T) will enter T_w, leave T_w, enter T_x, leave T_x, traverse all of T_y, return to and finish traversing T_x, and return to and finish traversing T_w. This would create an edge $w \to y$ of E - P, contradicting our assumption that C is a shortest cycle in L3.

This implies that there are edges $v \to w \to x$ of C such that $v \to w$ is in E - P, and $w \to x$ is in P. DFS(T) must first enter T_v, leave T_v, and traverse T_w completely before returning to T_v.

Let anc(w) be the last node of T_v visited by DFS(T) before T_w is visited; since this is a depth first, anc(w) is an ancestor of every node of T_w. We claim that every node of T_x is a descendant of anc(w). Certainly the node of $T_x \cap T_w$ is a descendant of anc(w). T_x is connected, and there is only one path between nodes of a tree, so the only way that T_x can contain a nonancestor of anc(w) is if T_x contains anc(w). This would create an edge $v \to x$ (in which case C is not a shortest cycle) or $x \to v$ (which would imply $w \to v$ by transitivity of P, making an edge $v \to w$ of E - P impossible); since neither is possible, L3 cannot contain a cycle. QED

Thus L1, L2, L3 and L4 are all have valid linear extensions. Since adjacencies between vertices are all reflected by these four lists, every chordal comparability graph is the comparability graph of a poset with dimension at most 4. QED

Kierstead, Qin, and Trotter showed that some chordal comparability graphs need 4 linear extensions, in an elegant use of Ramsey theory. Since Ramsey theory is not one of the assumed prerequisites for this book, we will not cover the proof here. Note that (as is typical with Ramsey theoretic proofs) the proof requires an enormous (27^{27}+1) number of vertices before we can be sure that a 4 dimensional poset in the class exists. I will not pose the question of finding the smallest chordal comparability graph which requires 4 linear extensions as an open problem, since extremal problems are not covered in this book.

We will discuss briefly the construction problem for finding a 4 dimensional representation of a chordal comparability graph. Recognizing 4 dimensional partial orders is NP-complete [183] , so we are forced to rely on an algorithm which finds a 4 dimensional representation only for chordal comparability graphs. We will discuss the time complexity involved in constructing the representation by following the proof of theorem ?, doing several depth first searches on the chordal graph representation.

In chapter ?, we show that the intersection model for chordal graphs as subtrees of a tree can be found in O(n+m) time. We will show how to follow the proof of theorem ? to construct the representation in O(n^2) time. No explicit time bound for the construction is given in the original paper [117]. Recall that the algorithm takes the oriented graph, and adds edges to create four different lists as follows. In list 1 edges are added between 'type 1' nonadjacent vertices, with the edge $x \to y$ added if T_x is traversed completely before T_y, and edges in the reverse direction are added in list 2. Lists 3 and 4 add edges between 'type 2' nonadjacent vertices; in list 3 the edge $x \to y$ is added if T_y is traversed in the middle of the traversal of T_x, with list 4 reversing these edges. We will divide the algorithm into two parts. We first identify type 2 and type 3 nonadjacent vertices, and then find the appropriate ordering on each class. The two parts can easily be combined if one is concerned with constant improvements in running time rather than simplicity of the construction.

For each vertex v of G, there are two nodes of the clique tree which we want to identify. These will be called v_1, the first node of T_v visited by DFS(T), v_∞, the last node of T_v visited by DFS(T). As we see in chapter ?, the tree representation of the chordal graph is a maximal clique tree, and the sum of the vertices in all maximal cliques is O(n+m). When we visit each vertex t of T during DFS(T) which corresponds to clique C, we look at every vertex c in C. Set c_∞ to the DFS number of t, and if c_1 is not assigned set c_1 to the depth first search number of t.

It is easy to identify type 1 nonadjacent vertices given these values. Step through the vertices of T in the order of DFS(T). Keep a list F of vertices which are finished with their traversals; F is originally empty. When a new node t of T is visited, label (f,v) as type 1 nonadjacent vertices for all f in F, and all v such that $v_1 = t$. Add all vertices with v_∞ to F. Since there are at most n nodes in T, and the work done at each node is O(n), type 1 pairs are identified in O(n^2) time. All nonadjacent pairs of vertices which are not labeled type 1 are then assigned type 2.

Orientation of edges between vertices of the same type in each list is easy. For any pair x,y of type 1 (type 2) vertices, if x_1 is less than y_1 then add an edge from x to y in L1 (L3) and from y to x in L2 (L4).

Since all edges for L1, L2, L3, and L4 can be found in O(n^2) time, and topological sort can be done in O(n^2) time, we can construct the lists in O(n^2) time.

Q to self: do in O(n+m)? good class open problem; very solvable.

We have already mentioned the open problem of finding an implicit representation that exactly characterizes the chordal comparability graphs. The representation above is good if we are interested only in efficient storage space and adjacency testing. Nevertheless, we know that we are missing some key properties of chordal comparability graphs if they are stored in this way, since there are problems (in particular, the recognition problem) which can be solved more efficiently on chordal comparability graphs than on the more general comparability graphs of four dimensional posets.

Finally, we will look at the recognition problem for chordal comparability graphs. Although chordal graphs can be recognized in linear time, recognizing comparability graphs takes time proportional to matrix multiplication. Recall that this time complexity comes from a two step process; an orientation phase which takes O(n+m) time, and a verification phase which takes time proportional to matrix multiplication. We will show how to use properties of chordal graphs to reduce the verification time to O(n+m).

Let G be a chordal graph, and let G' be the result of running the transitive orientation algorithm on G. Let v_1 v_2 ... v_n be a perfect elimination ordering of G.

Process the vertices in perfect elimination order. Let x be the next vertex visited. We will take O(|N(x)|) time, and either find a transitivity violation in G or determine that removing x will not change whether G' is a comparability graph. If all vertices are eliminated, we declare G to be a comparability graph.

Find all neighbors of x (both inneighbors and outneighbors) in the remaining graph. Let w be the vertex in this set which has an edge to x, and has smallest outdegree of any such vertex. Verify that w has an edge to all vertices in L which have edges from x, and has edges from all other vertices in the set which have an edge to x. If these conditions hold, then remove x and continue with the algorithm.

It should be clear that if G' is a transitive orientation, we will never declare that a transitivity violation exists. Suppose that G is not a comparability graph. If the algorithm declares G to be a comparability graph, then at some point a vertex v_i was removed so that v_i ... v_n induced a noncomparability graph, but v_{i+1} ... v_n induces a comparability graph. There must be a transitivity violation $a \to v_i \to z$, where a and z both come after v_i in the perfect elimination ordering. However, our algorithm checks for an edge from a to w, where w is the vertex with smallest outdegree which has an edge to v_i, and that w has an edge to z. Therefore, if a does not have an edge to z, then the graph has a transitivity violation even after v_i is removed.

Since each vertex is eliminated in $O(|N(x)|)$ time, we can recognize chordal comparability graphs in $O(n+m)$ time.

6.3 Fill-in Schemes

A number of algorithms use a graph as a starting point, and add edges to the graph based on certain conditions. Fill-in schemes can be viewed as the reverse process of elimination schemes studied in chapter ?, though fill-in schemes are less well known than elimination schemes. The name is taken from the notion of fill in Gaussian elimination, where 'pivot' elements chosen add nonzero entries to the matrix, so that an originally sparse matrix can end the Gaussian elimination process with many nonzero elements filled in. Minimizing fill in this process is a well known problem; it was one of the original open problems in Garey and Johnson [71] and was proved NP-complete by Yannakakis [182]. Matrices which can be solved without fill can be modeled as chordal graphs (if M is symmetric) [139] or perfect elimination bipartite graphs [81]; Golumbic gives a good overview of these applications [79].

We discuss several notions of creating graph classes on the basis of fill. These classes may have a natural notion of original information to store with each vertex, but computing the edges of the final graph involves use of global information.

The Steinhaus graphs and bipartite Steinhaus graphs are examples of graph classes defined by a fill process. These classes are unusual in that each graph is defined by a string of n bits; thus the number of labeled Steinhaus graphs is only 2^n.

Consider an n by n symmetric matrix defined as follows. Place a 0 in each position (i,i). The first row of the matrix is filled with a string of n-1 bits given as input. Each remaining row in the upper diagonal is computed by setting the value M[i,j] equal to M[i-1,j-1] + M[i-1,j] mod 2; the lower diagonal is filled in to make the matrix symmetric. For instance, filling in a matrix defined by input 1 0 1 1 gives us the matrix

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

A graph is a Steinhaus graph if the adjacency matrix can be built in this way. One can also treat the arrays constructed as bipartite adjacency matrices; for that class, we fill in all entries (rather than just those above the main diagonal) using the set of rules. For either class, it may be natural to think of storing one bit for each vertex, but it is not possible to test adjacency using only the bit stored with each vertex. The adjacencies come from the 'fill-in' of the matrix.

Neither Steinhaus graphs nor bipartite Steinhaus graphs are a hereditary class of graphs. In fact, even though there are only 2^n Steinhaus graphs on n vertices, every graph is an induced Steinhaus graph (Ex ?). We will not investigate this odd class of graphs further, but this is an example of a graph class defined by a fill-in process.

I understand that some researchers of Steinhaus graphs are proud of the fact that the graphs have no practical applications; they arose from a problem posed by Steinhaus. I urge the gentle reader to refrain from inflicting on this class a host of optimization algorithms which are not suitable for a graph class with $2^{O(n)}$ members.

6.4 Closure Operations

A general technique for dealing with graphs is to use an operation called k-closure. In a single step, an edge is added between a pair of vertices x, y such that the degree of x plus the degree of y is at least k. In another variant of the same idea, we add an edge if the number of vertices adjacent to either x or y is at least k. The k-closure G' of a graph G is the graph formed by repeating this operation until no new edge can be added.

Given a particular G and k, it is not hard to see that G' is uniquely defined (exercise ?). Therefore, for any graph class C with $\mathrm{f}(n)$ members, we can define a new graph class with at most $\mathrm{f}(n)$ members to be the k-closure of graphs in C. Even if C has an implicit representation, it is not clear that the new class will have an implicit representation, since adjacency between x and y in G' may depend on adjacencies between other sets of vertices in G. Thus, for any class of graphs which has an implicit representation, such as planar, degree bounded, or circular-arc graphs, we can define a new class which could be difficult to represent.

We should note that graph classes formed by closure are not necessarily hereditary. For example, consider the class of 10-closures of planar graphs. Clearly, K_5 is not in the class, since K_5 is not planar. However, K_5 is an induced subgraph of a graph in the class (exercise ?).

Vijay's fill-in

Another example of the use of fill-in as an algorithmic technique is recognition of weakly chordal graphs. Recall that a graph is weakly chordal if neither G nor its complement contains a chordless cycle of length at least 5.

The definition of weakly chordal graph implies an $\mathrm{O}(n^5)$ algorithm, since one can determine whether an arbitrary graph has a cycle of length k in $\mathrm{O}(n^k)$ time; this can also be reduced slightly to $\mathrm{O}(n^{k-3}MM)$ [157]. The best known algorithm for weakly chordal graph recognition works somewhat differently, based on a fill-in algorithm described below.

A two-pair in a graph G is a pair of vertices x, y such that every chordless path between x and y contains exactly two edges. [95] shows that every weakly chordal graph has a 2-pair. In exercise ?, one is asked to show that if x, y is a 2-pair, then G is weakly chordal if and only if G + (x,y) is weakly chordal. Thus, the test for G being weakly triangulated is to perform a fill-in procedure; G is weakly chordal if and only if the final result is a clique. Although it is not known how to find a single 2-pair in less than $\mathrm{O}(n^3)$ time, one can store enough information from previous steps so that each 2-pair after the first can be found in $\mathrm{O}(n^2)$ time, which gives an $\mathrm{O}(n^4)$ algorithm for recognition of weakly chordal graphs [150]; hwk?.

It should be noted that it is not possible to add all two pairs in G at the same time. Adding an edge between one 2-pair can make a chordless path between another 2-pair. In order to have a unique notion of fill-in, assume that pairs of vertices are checked in lexicographic order at each step to see whether they are 2-pairs. This does not affect the time complexity of the algorithm; we merely want to define a unique result of the fill-in process.

If the fill-in procedure above is performed in one natural manner, as given in [150], the algorithm will run in $\mathrm{O}(n^4)$ time but has several drawbacks. First, the algorithm does not produce a chordless cycle in G or its complement when the input is not weakly chordal; indeed, if the graph has no two-pair, we do not seem to get any information on where the hole occurs. Second, in order to find two pairs efficiently, the algorithm uses $\Theta(n^3)$ space, keeping track of an entire graph representing possible two-pairs with each vertex x. These deficiencies are overcome in []; the new algorithm finds each 2-pair after the first using $\mathrm{O}(n^2)$ time and space, and identifies a hole in G or its complement if the input is not weakly chordal.

The relationship between the input graph and the graph which is the result of the fill process is not well understood. Suppose that we define a class of graphs as follows. G is in the class if G is the result graph of the fill process for some graph G' with maximum degree d. Storing G by storing the corresponding G' is a space efficient representation, and there is a natural way to store O(logn) bits at each vertex, but we cannot test adjacency in G efficiently at all (it seems to take $\mathrm{O}(n^4)$ time to test a single adjacency), and adjacency testing is not local. Although it is not common to do so, any fill process can define a graph class in this fashion, and these may pose problems for the implicit graph question; depending on your operations, the graph classes may or may not be hereditary.

6.5 Dilworth Number and Poset Width

NEW SECTION! MUST COMPLETE! move to another chapter!

The Dilworth number of a graph is the maximum number of vertices in the graph such that the neighborhood of one vertex is not contained in the (closed) neighborhood of the other. Graphs with fixed Dilworth number have been studied in a number of papers; thus, for example, it is known that graphs with Dilworth number 1 are exactly the threshold graphs, graphs with threshold number 2 are equal to a class called threshold signed graphs studied in [], graphs with Dilworth number 3 must always have a simplicial or cosimplicial vertex [], and graphs with Dilworth number 4 are perfect.

Dilworth number has a close connection with the notion of width of a poset, which is the maximum number of mutually independent elements in the poset. The Dilworth number of G is exactly equal to the width of the poset constructed by adding an edge from x to y if the neighborhood of y is contained in the neighborhood of x.

The easiest way to see that graphs with Dilworth number k and posets of width k have an implicit representation is to use a famous theorem due to Dilworth; the size of the maximum antichain of a poset is exactly equal to the minimum number of chains needed to partition the elements of the poset. Note that this can be viewed as a proof that comparability graphs are perfect; antichains correspond to independent sets and chain partitions to clique covers.

Given k chains which partition the neighborhood containment poset, we can place every vertex v at an appropriate level of each chain, so that vertices above this level are adjacent to v, and vertices below this level are nonadjacent to v. Thus, for each vertex v we can maintain the position of v with respect to k chains, the chain v is in, and vs height in its own chain; from this information, it is possible to determine adjacency between x and y using only the bits stored at x and y.

In general, Dilworth number can be computed by computing the neighborhood containment poset, and then using flow related techniques to find the width of the poset, as described in chapter ?. This would take time proportional to matrix multiplication for finding the poset, and bipartite matching for finding the width. For fixed Dilworth number, these steps can be reduced. The neighborhood containment poset can be found in $O(n^2)$ time for any fixed k [cite Felsner, Raghavan, Spinrad]; make a homework problem asking to recogniize fixed width in $O(n^2)$. or o(MM) for easier, pointer to O(n squared).

6.6 Other Classes

We saw above that we could take an implicitly representable class of graphs, and define an operation called closure so that it is unclear whether the class defined by closures of graphs in the class have an implicit representation. Many operations besides closure can be used to define similar classes. We do not intend to let the reader feel that any of the classes defined here is important in and of itself. Our intention is merely to show methods of defining classes of graphs which are challenging with respect to implicit representation. Hopefully, looking for representations of such classes could lead to new methods for finding implicit representations, or a disproof of the implicit graph conjecture.

For example, a number of problems use the k-neighborhood of a vertex x, meaning the set of vertices at distance at most k from x. For any graph G, we can define the k-neighborhood graph of G, in which x and y are adjacent if and only if there is a path of length at most k from x to y. Note that this particular class is not hereditary, but it is not clear whether, for example, 2-neighborhood graphs of path graphs would have an implicit representation. Many operations beside k-neighborhood can be used; we can define G' such that x and y are adjacent if N(x) contains N(y) in G, x and y are contained in a common nontrivial module, or any other global property.

We could also use optimization problems to define classes of graphs. To choose one random example, we could take a weighted set of vertices, and define x and y to be adjacent if V - x,y can be partitioned so that the sum of weights in each side of the partition is equal. If the weights are of polynomial size with respect to n, giving the weights gives a small space representation of the graph, but deciding whether x and y are adjacent cannot be solved quickly, so storing a graph in this form is not particularly satisfactory. Again, this is not necessarily a hereditary class; readers can construct their own classes based on their favorite decision or optimization problem.

6.7 Exercises

1) There are many variants of classes defined by intersection of paths in a tree; the tree can be undirected, rooted, or directed but not rooted, and paths can be defined as intersecting if they meet at a vertex or intersecting if they share an edge. Perhaps surprisingly, these variants have very different behavior with respect to recognition and optimization; we will look at some differences in the graph classes below. See [] for a systematic study of these classes.

a) G is a path graph if G is the intersection graph of paths on an undirected tree, where paths are defined to be intersecting when they share a common vertex. Show that every undirected path graph is chordal.

b) G is an EPT graph if G is the intersection graph of paths in a tree, where paths are defined as intersecting if they share a common edge. Show that EPT graphs are not necessarily perfect.

c) G is a directed path graph if G is the intersection graph of paths in a rooted tree, with intersection defined as in path graphs. Show that the 3-sun (a triangle, plus three independent vertices adjacent to each pair of vertices from the triangle) is a path graph but not a directed path graph.

d) Show that all the classes defined above have implicit representations.

2) Design an O(n+m) transitive reduction algorithm for chordal comparability graphs.

3) Show that Steinhaus graphs are not hereditary.

4) Show all graphs are induced Steinhaus graphs; that is, every graph is an induced subgraph of a Steinhaus graph. Use this to show that Steinhaus graphs do not have

an implicit representation.

5) (this problem is due to Ian Stobert) Design a method for storing Steinhaus graphs so that the product of (space used)(time of adjacency testing) is o(n^2).

6) Design an O(n^k) algorithm to determine whether G contains an induced cycle of length at least k. If you are more ambitious, design an O(n^{k-3}MM) algorithm for solving the same problem.

7) Let x, y be a 2-pair in an arbitrary graph G. Show that G is weakly chordal if and only if G + (x,y) is weakly chordal.

8) The algorithm for recognizing weakly chordal graphs repeatedly adds edges between some two-pair x,y; if G is weakly chordal, this leads to a clique. Show that if G is not weakly chordal, the final graph G' created is not unique.

9) Suppose that you start with a graph G, and construct a graph G' by choosing a pair of vertices x, y such that $|N(X) \cup N(y)| > k$ for a fixed k, and repeat this until no new edges can be added. Show that the same graph G' is created, independently of the order in which edges are added.

10) Show K_5 is an induced subgraph of 10-closures of planar graphs.

11) A simple elimination ordering is an ordering of the vertices so that when x_i is eliminated, neighbors of x_i form a clique (i.e. x is simplicial), and the closed neighborhoods of the neighbors of x_i are ordered by set containment. Cardinality LBFS is the variant of LBFS discussed in chapter ? such that a vertex of maximum degree in G from the last set is chosen to be last on the output list. Show cardinality LBFS gives strong elimination ordering for chordal comparability graphs.

12a) Use problem ? above to design linear algorithms for the maximum cardinality matching and dominating set problems for chordal comparability graphs.

12b) Show matching is at least as hard as perfect bipartite matching for chordal graphs and comparability graphs, and that dominating set is NP-complete for both classes.

7 Visibility Graphs

Visibility graphs are a popular class of graphs studied in the field of computational geometry, and pose some special challenges as far as finding a good representation. Although the number of graphs in the class is small, the natural geometric representation uses both real numbers to store vertices and requires global information to determine adjacency. The class of graphs is not hereditary, but a closely related hereditary class can be defined, which raises some new questions of interest.

Let P be a simple polygon in the plane. The visibility graph G of P is defined by associating a vertex with each endpoint of P such that (x,y) is an edge of G if the line between x and y is either an edge of the polygon, or lies entirely within the polygon. To understand the definition, think of edges of the polygon as walls of a lit room, and edges correspond to corners which can see each other. Visibility graphs are important to the area of computational geometry; for more on the uses of this class of graphs, see [132], [148]. Recognizing visibility graphs in polynomial time is a major open problem, even if the order of vertices in the polygon is specified. In fact, it is not known whether the problem is in NP. A set of necessary, but not sufficient, conditions for an ordered set of vertices is given in exercise ?; an open problem related to one of these conditions is stated along with the exercise.

Suppose that we try to represent the visibility graph in its most natural geometric form; at each vertex, we can store the coordinates of the corners of the associated polygon, and the next vertex on the polygon. Using this information, the graph can be reconstructed. However, the number of bits stored at each vertex needs to be large, so this form of representation is not space optimal. If we are also looking for an implicit representation, we have another problem, since we need to know all of the polygon edges before we can determine whether two vertices can see each other, even when their coordinates are known.

In the next sections, we study various interesting properties of visibility graph representations. We first show that the number of visibility graphs is $2^{O(nlogn)}$, by using Warren's theorem. However, the integers for representing coordinates must be at least exponential, leaving an important open problem about just how large these numbers must be. We will look at a form of representation that is good, though not optimal, with respect to space. We investigate the class formed if we change visibility graphs in a natural way to get a hereditary class, and finally study an important optimization problem on visibility graphs.

7.1 Counting Visibility Graphs

We can show that the number of visibility graphs is $2^{O(nlogn)}$ as a straightforward application of Warren's theorem.

Conceptually, variables of our set of polynomial equations correspond to coordinates of endpoints of the polygon. To make our counting argument simpler, we will slightly overcount the number of visibility graphs by not checking whether a set of variables corresponds to a simple polygon. Thus, we have $2n$ variables for the n vertices, since we need an x and y coordinate for each vertex.

We can create a polynomial equation for each pair of vertices u, v and each line L of the polygon, such that the value of the equation is negative if and only if the visibility from u to v is obstructed by L. Although constructing the appropriate equation is not trivial, we do not feel that we can improve on the description in the standard algorithms text [37]; the reader can consult that book to see how to write an equation which evaluates to a positive number if and only if two line segments cross. They also present an equation which evaluates whether two segments meeting at a point turn to the left or right at the point.

We need only one distinction in order to reconstruct the graph entirely from the sign pattern of this set of equations. Some works refer to an external visibility graph of a polygon, in which two vertices are adjacent if and only if the line between the vertices is entirely outside the polygon. If the line from u to v crosses no edge of the polygon, then u and v are adjacent either in the visibility graph or the external visibility graph. This can be distinguished by testing which way the lines of the polygon around u turn with respect to each other, and with respect to the segment (u,v).

Thus the number of visibility graphs is bounded by the number of sign patterns. Using the techniques of chapter ?, since we have $O(n^3)$ equations over $2n$ variables, the number of sign patterns is bounded above by $(cn^3/2n)^{2n}$ which is $O(2^{O(nlogn)})$.

7.2 Clique Cover Representation

Although the most natural form of representation for visibility graphs would be to use coordinates of the points, this is not useful if we are looking for a space efficient representation. Lin and Skiena [114] proved that even very simple visibility graphs require endpoints to have exponential sized integers. However, it is not known whether singly exponential sized integers are sufficient; the question is important because if we could guarantee that the number of bits in the integer is polynomial, then visibility graph recognition is in NP.

Open Question: Can all endpoints of a visibility graph be assigned integer coordinates such that the integers use a polynomial number of bits?

There is a natural form of storage studied by Agarwal, Alon, Aronov, and Suri which uses a relatively small number of bits to store a visibility graph. We will study their representation, in this section. We note that the representation has its drawbacks; it is neither space optimal, nor can we test for adjacency in constant time, but it is the most significant reduced space representation which is currently known.

The authors of [2] consider the problem of representing a visibility graph as a covering set of cliques and complete bipartite graphs so that every graph in the set is a subset of G, and every edge is contained in at least one of the graphs of the covering set. Clearly, G can be retrieved from this representation. They measure the size of the representation as the total number of vertices in all the complete bipartite graphs which cover G; in our analysis, we must multiply by a factor of log$n$ since we are concerned with the actual number of bits in the representation. The fact that the covering set of a visibility graph has small size also allows leads to efficient algorithms for counting the number of edges in a visibility graph when the polygon is given as input,and finding the longest diagonal of a polygon [3], [4].

The paper [2] describes an algorithm which constructs a covering set which has O($nlog^4n$) bits. It is not clear whether the algorithm ever can construct a set which uses this many bits, however. They also give a lower bound on the size of the cover; they show that there are visibility graphs which have no cover (whether found by their algorithm or not) using o($n$log$n$) vertices. We note that while it is theoretically possible to have covering sets including a total of Θ($n$log$n$) vertices which can be stored in o($nlog^2n$) bits, this lower bound also can be modified to show that any covering set requires Ω($nlog^2n$) bits on some visibility graphs. Given the gap between upper and lower bounds on this natural form of representation, we have a number of open problems.

Open problem: Give a tight bound (with respect to order notation) on the number of bits used in an optimal covering set of a visibility graph.

Open Problem: Can you find a covering set which matches the above bound in polynomial time?

We would also like tight bounds on the number of bits used for covering sets by the algorithm in [2].

We will not cover the algorithm for constructing the covering sets in detail. The algorithm works roughly as follows. Find a visibility line which divides the polygon into two pieces of approximately equal size using a theorem of Chazelle which will be described in the next section. Find covering sets for edges which are entirely contained within one of the subpolygons recursively. Use a specialized algorithm to cover all edges of the bipartite graph consisting of edges which go from one side of the polygon to the other; this algorithm creates a cover which can be stored using O($nlog^3n$) bits. Standard analysis of recurrences shows this representation uses O($nlog^4n$) bits. It is interesting to note that the covering sets used are all bipartite. This algorithm requires the model as input, which brings up the following open problem.

Open problem: Find a small covering set of a visibility graph in polynomial time, if the input is in the form of an adjacency matrix.

We now turn our attention to the clever lower bound argument on the size of the covering set from [2]. Consider a graph H formed by subtracting a perfect matching M from $K_{n,n}$. It is not difficult to show that H is a subset of a visibility graph on O(n) vertices (exercise ?). We show that any covering set of H requires a total of Ω($n$log$n$) vertices in the sets.

We will call the members of one color class x_1,..., x_n and the other class y_1,...,y_n, with each x_i nonadjacent to y_i. Each member of the covering set for H is a complete bipartite graph which contains at most one vertex from each pair x_i,y_i.

For any covering set of H, randomly assign the value 0 or 1 to each graph in the covering set. Let E_i be the probability that every graph of the cover which contains x_i has value 0, while every graph of the cover which contains y_i has value 1. The sum of the E_is is at most 1; the edge (x_i,y_j) must be covered in some graph, so we cannot have all graphs covering x_i given value 0 while all graphs covering y_j have value 1. If covers(i) is the number of graphs of the cover containing x_i or y_i, clearly $E_i = 1/2^{covers(i)}$. Thus, sum from i=1 to n $1/2^{covers(i)} \leq 1$.

There are n parts of the sum above, which implies that the average value of $1/2^{covers(i)}$ is at most $1/n$; informally, this means that the average number of graphs in the cover set which contain x_i or y_i is at least log$n$, so the total size of the cover is $\Omega(n\text{log}n)$. This can be made more precise by arguing that for at least $n/2$ values of i, $1/2^{covers(i)}$ is no larger than $2/n$, and following the argument above.

Finally, we might also think of covering the edges with more general subgraphs than cliques or complete bipartite graphs; these are chosen because cliques and complete bipartite graphs on k vertices can be described using O($k$log$k$) bits. In the optimization chapter, we will see that there is an efficient method for storing the graph induced by the neighborhood of any 'convex fan' in a visibility graph; this motivates the study of the smallest cover of a visibility graph with cliques, complete bipartite graphs, and neighborhoods of convex fans.

7.3 Induced Visibility Graphs

In this section, we study the class of graphs formed if we force the class of visibility graphs to be hereditary in the most natural way. A graph is an induced visibility graph if there is a simple polygon P such that vertices of G map to a subset of vertices of P, and x is adjacent to y iff x and y are neighbors in P, or the line from x to y is entirely within P.

Previous work on induced visibility graphs includes [148], which showed that a number of classes of graphs were not contained in the induced visibility graphs, and [62], which shows that visibility graphs are not characterized by a finite set of forbidden induced subgraphs. The problems of counting, recognizing, and representing induced visibility graphs are open. Partial results on counting the number of induced visibility graphs are presented in this section.

With a few exceptions, researchers have had difficulty relating problems on visibility graphs to problems on other well known classes of graphs. In the opinion of this author, this difficulty comes in part from the focus on visibility graphs rather than on induced visibility graphs. For example, the graph on three vertices in figure ?-a Is not a visibility graph, because every visibility graph must be Hamiltonian. Therefore any graph class which contains this simple graph is not contained in the visibility graphs. However, if we think of visibility between points on a polygon as a representation scheme, we have the power to represent this graph as in figure ?-b.

Let n be the number of vertices of an induced visibility graph G. If we hope to

Figure 6: An Induced Visibility Graph which is not a Visibility Graph

use a polygon for representing G, we must hope that the polygon does not need too many extra points. If we are worried about the space complexity of a representation, a polygon with n^2 points would be of no use. We will show that G can be represented on a polygon with at most O($n$logn$\alpha(n)$) points, where α is the inverse Ackermann function studied in [162]. Using Warren's theorem, this will give an upper bound of $2^{O(nlog^2 n\alpha(n))}$ for the number of induced visibility graphs on n vertices. There is no matching lower bound on the number of induced visibility graphs on n vertices.

Let P be any polygon which represents an induced visibility graph G. We will find another polygon $P2$ which represents the visibilities between vertices of G correctly, such that $P2$ has O($n$logn$\alpha(n)$) points. The points of P which correspond to vertices of G will have the same coordinates in P and $P2$.

Draw lines between all pairs of adjacent vertices from G in P, as in figure ?, where the polygon is ordered numerically, with certain points which are not part of G but are in the polygon marked by x. These visibility lines divide the plane into regions. Remove all lines which do not touch the outer region; in figure ?, this will remove the line from 4 to 6 and the line from 2 to 5. Note that there are some lines, such as the line from 1 to 3, which touch the outer region at several segments. The outer envelope E of the regions is formed by eliminating all segments except for those on the outer region.

We can get a polygon which represents G as an induced visibility graph by adding one point for each endpoint of the outer envelope, and one point for each segment of the outer envelope, with points of the second type being infinitely close to the corresponding line of the outer envelope. All visibility lines between adjacent vertices are contained within E, and thus will be contained within the new polygon, since all polygon edges are in the outer region. Consider any nonadjacent pair of vertices x, y. The line between x and y cannot be entirely contained in E; any polygon edge which would cut the visibility between x and y would force some edge of the visibility envelope to be outside the polygon. Thus, the edge between x and y must leave the visibility envelope at some point, between points a and b of E. Since a point is introduced infinitely close to the line between a and b in E, the line from x to y goes outside of the new polygon at this region.

Therefore, if we want to bound the number of points in $P2$, we need to get a bound on the number of segments in the outer envelope. We will show that the

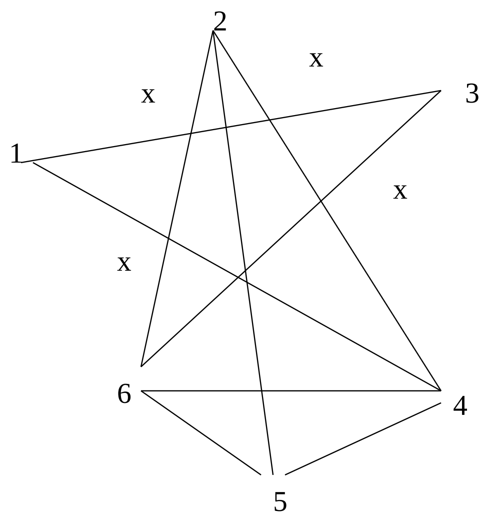


Figure 7: Visibility Lines in an Induced Visibility Graph

number of lines which participate in the outer envelope is O($n$log$n$), and invoke a deep theorem which allows us to say that the number of segments in the outer envelope is O($n$log$n\alpha(n)$).

We will get this result using a simple divide and conquer approach. We divide the polygon P into 2 pieces, and argue that the number of edges of the outer envelope which go between the two sides is O(n). If we can divide vertices of G equally between the two pieces of P, and use the same argument recursively, the number of lines of the outer envelope is bounded above by the recurrence E(n) = 2E($n/2$) + O(n), which works out to O($n$log$n$). We will not be able to divide the polygon into pieces of exactly equal size, but we will be able to divide P into reasonably balanced pieces, which is enough to make the recursive approach give a bound of O($n$log$n$).

Suppose that we divide P along some line l, where l is completely inside the polygon. Call the two pieces of the polygon formed by adding l P' and P''. Any line on the visibility envelope E either crosses l, or is on the visibility envelope of P' or P''. We will show that the number of lines of E which cross l is O(n).

For each vertex of G, consider the 2 lines which go to the other half-polygon which are furthest clockwise, and furthest counterclockwise. We claim that all visibility lines on the outer envelope which go between the two half-polygons come from this set of (at most) $2n$ lines. If a line between x in P' and y in P'' is not one of the lines chosen, it is completely enclosed by the 2 lines from x and the 2 lines from y; an example is shown in figure ?. The portion of the line which travels inside P' is contained within the two bounding lines from x, while the segment within P'' is contained in the two lines out of y.

To make the recursive argument work, we must show that there is a line l which can cut the polygon so that vertices of G are split roughly equally between P' and P''. Note that this can be different than finding a line which splits P into equal size pieces, since vertices of G may form only a small fraction of the vertices in P.

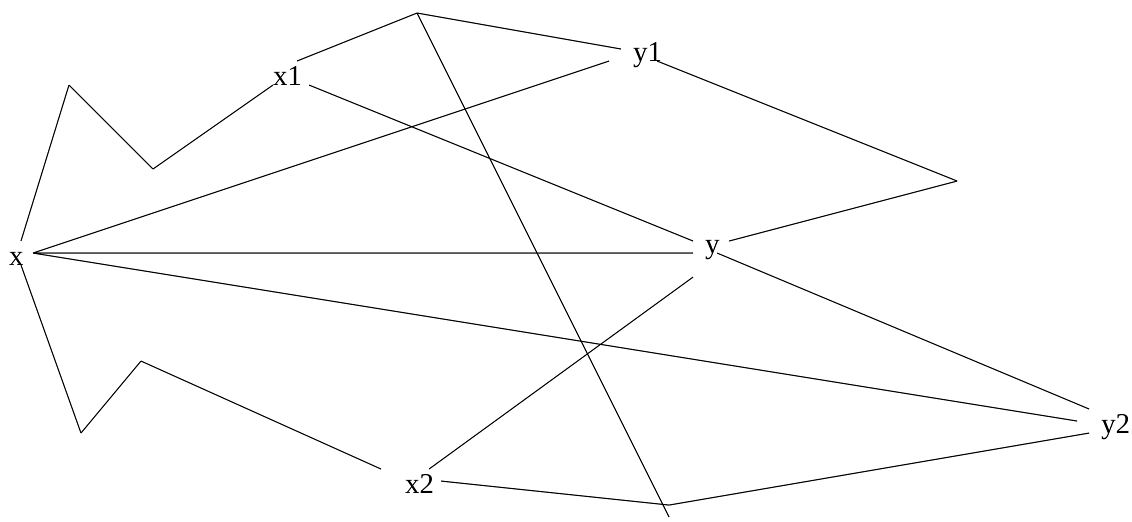


Figure 8: The Line from x to y is not on the Outer Envelope

Fortunately, we can use a powerful theorem due to Chazelle [31] to find a line which cuts the polygon. Given any polygon with weights assigned to the points of the polygon, there is a line which separates the polygon into two subpolygons, such that the sum of the weights of vertices in each subpolygon is at least 1/3 of the total weight of the points. For our problem, we give weights of 1 to vertices of G in P, and weight 0 to any other points of P; Chazelle's theorem tells us that we can choose l so that both P' and P'' contain at least $n/3$ vertices.

We can now make an inductive argument that the number of edges which touch the outer envelope of the visibility polygon is at most cnlog$n$. We choose $c = 8$ for simplicity; we could choose any $c \geq 2/$(log3-1). The argument is clearly true for graphs with at most three vertices. Suppose that this is true for all polygons which contain fewer than k vertices from a graph, and let P be a polygon which contains k vertices of a graph. Choose a line l as described above by using Chazelle's theorem. P' and P'' are smaller polygons, so induction can be used on them. Let n' be the number of vertices of G in P', and let n'' be the number of vertices of G in P''. The number of edges which touch the outer envelope is at most $2k$ + cn'logn' + cn''logn'', by previous argument and the inductive hypothesis. Since both n' and n'' are at least $k/3$, logn' and logn'' are at most log$k$ + 1 - log3. Since n' + n'' sum to k, the number of edges of the outer envelope is at most cklog$k$ + $2k$ + ck - cklog3); our choice of c guarantees that this is at most cklog$k$. (note slight technical glitch to fix, since n' + n'' really = n+2; easy to fix but want to avoid loss of clarity)

Although O($n$log$n$) lines touch the outer envelope, this does not imply that P has O($n$log$n$) points. A single line may have several segments which touch the outer envelope, and we add a point to P for each segment. There is a relationship which is surprising to me, at least, between the size of the outer envelope of a set of lines and the inverse of Ackermann's function, normally called $\alpha(n)$, which is a function invented for entirely different purposes. The number of segments of the outer envelope of a set of k lines is at most $k\alpha(k)$ [].

In summary, we know that any induced visibility graph on n vertices is an induced subgraph of a visibility graph on O($nlogn\alpha(n)$) vertices (note that $\alpha(nlogn)$ is O($\alpha(n)$)). Since there are $2^{\Theta(nlog^2 n\alpha(n))}$ visibility graphs on this number of vertices, it is easy to see that there are $2^{O(nlog^2 n\alpha(n))}$ induced visibility graphs on n vertices. However, it is by no means clear that this bound is tight.

Open Problem: What is the correct function f(n) such that the number of induced visibility graphs on n vertices is $2^{\Theta(f(n))}$?

Clearly, f(n) lies somewhere between $n$log$n$ and $nlog^2 n\alpha(n)$. If the correct answer is $n$log$n$, then the question of whether an implicit representation of visibility graphs can be found is particularly interesting. Resolving the question would allow us to know how much we could hope for in a space optimal representation. Of course, finding a generalized implicit representation is also open for this class of graphs.

7.4 Optimization on Visibility Graphs

The key result on visibility graph optimization is presented in chapter ?, which studies a polynomial time algorithm for the maximum clique problem. This is the most well known (to the authors) example of a problem on a graph class which can be solved efficiently if the model is given, but is open if the input was in the form of an adjacency matrix; such problems are discussed at length in chapter ?. The algorithm also can be used to solve the clique problem on induced visibility graphs, if the model is given as input; it is also solvable if only the vertex ordering of the polygon is given, rather than the entire model.

Of somewhat lesser interest is the fact that a Hamilton circuit can be found in a visibility graph if the representation is given; this is an example of a trivial problem, since the answer to the Hamilton circuit question is always yes. The question of finding a Hamilton circuit if a visibility graph is given in adjacency list form is still open.

We first note that the number of maximal cliques on a visibility graph is exponential, so that this problem cannot be solved by the simple algorithm of enumerating all maximal cliques. (hwk or here?) The usual procedure for showing this is to show that the graph composed of pairs of nonadjacent vertices which are adjacent to all other vertices in G is in the class. For visibility graphs, consider placing vertex pairs on a circle; each vertex pair is placed very close together on the circle, and at some distance from the next vertex pair. Between each vertex pair x, y, add a third vertex which is just far enough inside the circle to block visibility between x and y. Create a polygon by connecting these vertices along the order defined by the circle. Each set containing one vertex from each vertex pair, and all intermediate vertices, is a maximal clique, so there are $2^n/3$ maximal cliques in this visibility graph.

The obvious question is whether the efficient algorithms are due to the representation or the special properties of the graph class. The major open question, of finding a polynomial time algorithm for the clique problem on visibility graphs and induced visibility graphs if the input is in adjacency list form, is given elsewhere; we pose some related questions here.

Open Problem: Find a polynomial algorithm for the Hamilton Circuit problem on induced visibility graphs, or show that the problem is NP-complete on the class.

The problem above is open whether the model is given or not.

Lin and Skiena [114] study the complexity of a number of problems on visibility graphs. For example, they show that the independent set and dominating set problem are NP-complete on the class, and that isomorphism is isomorphism complete. However, the following problem is left as an open problem in that paper.

Open Problem: What is the complexity of the chromatic number problem on visibility graphs, both for the case when the model is given as input and for the problem when input is in the form of an adjacency matrix.

7.5 Line of Sight Graphs and Other Variants

Visibility representations are studied for other classes of objects besides polygons. In the more general problems, vertices map to geometric objects, and two vertices are adjacent if and only if there is some straight line (sometimes these lines are restricted to horizontal and/or vertical lines) between points on the corresponding objects which does not hit any object corresponding to other vertices.

In one interesting class, studied in [],[], vertices are mapped to horizontal segments on the plane, and visibility lines must be horizontal or vertical. For our purposes, this class is interesting because we no longer have the problem of representing endpoints efficiently for this class. Nevertheless, the natural representation is not implicit, since adjacency between vertices depends on which other lines can obstruct the visibility.

This class has an implicit representation for a different reason. Every line of sight graph of horizontal intervals is a planar graph (check, hwk?), and thus has an implicit representation.

Visibility graphs can also be generalized to higher dimension; the author knows little of work in this direction. (clique? recognition hard? all graphs?)

Errata

The paper [] claims that the number of lines on the outer envelope of any set of lines with n endpoints is O(n). If this were true, it would give a bound on the number of induced visibility graphs of at most $2^{O(nlogn\alpha(n))}$?. The proof in that paper is incorrect. Note that the proof also would work for pseudolines (lines which are not necessarily straight connecting points), and it is possible to construct an example on n points for which $\Theta(n\log n)$ pseudolines touch the outer envelope.

Exercises

1) Show that the number of maximal cliques in a visibility graph can be exponential.

2) Show that there is a set of $\Theta(n\log n)$ pseudolines such that each touches the outer envelope (point to the construction in exercise in optimization chapter?)

3) This problem deals with some well known necessary conditions due to Ghosh [76] for a Hamilton circuit $h_1\ h_2\ ...\ h_n$ to be the outer border of the polygon representing a visibility graph. (Q to self: which hold for IVGs?) For each of a,b,c, you may solve either 1) the example question; show a graph and ordering which violate the condition 2) the proof question; show that the condition is necessary, or 3) the algorithm question; design and analyze an efficient algorithm to test for the condition.

3a) An ordered cycle is a cycle in which vertices are in increasing order. Ghosh showed that every ordered cycle of length at least 4 contains at least k-3 diagonals.

3b) A blocking vertex for a nonadjacent pair of vertices (i,j) is a vertex b such that all vertices between i and b are nonadjacent to all vertices between b and j on the simple path from i to b to j along the Hamilton cycle. Ghosh showed that every invisible pair must have a blocking vertex.

3c) A nonadjacent pair of vertices x,y is called minimal if they have exactly one blocking vertex. Two pairs u,v and x,y are separable if one can travel from u to v on the Hamilton cycle and avoid both x and y. Ghosh showed that two separable nonadjacent pairs must contain distinct blocking vertices.

3d*) Find a graph and ordering which satisfy all of Ghosh's conditions, but does not correspond to a valid polygon for the corresponding visibility graph (give graph?).

4) A graph with an ordering is ordered chordal if every ordered cycle of length at least 4 has a chord. Give a polynomial time algorithm for determining whether an ordered graph is ordered chordal. The following problem seems to be open and very natural:

Open Problem: Can you determine in polynomial time whether a graph has an ordering so that every ordered cycle has a chord?

5) Coullard and Lubiw [35] showed that if G is a visibility graph, there is an ordering of the vertices so that the first 3 vertices form a clique, and every vertex added is adjacent to some 3-clique of old vertices. Such an ordering can be used to recognize whether there is a polygon P such that G is the visibility graph of P, and all distances between visible points are equal to those given in an input matrix M; fundamentally, adjacency to a 3- clique fixes a point in space.

a) Design an algorithm to find such an ordering in polynomial time, and analyze its complexity (you should be able to design an O(n^7) algorithm).

b) Use the above property to show that (some particular graph G) is not a visibility graph.

6) Show that the number of external visibility graphs is $2^{O(nlogn)}$

7) The following is an example of a type of problem called an art gallery problem, which is closely related to visibility graphs. For a deeper study of such problems, see [132], [148]. In these problems, a polygon is viewed as the walls of an art gallery, and

we consider placing guards at points so that some guard can observe every point of the gallery wall (where a picture may be hanging). There are many variants, depending on the restrictions on the shape of the gallery, and forms of movement allowed for the guards. Suppose that we have a simple polygon, and we want to place stationary "guards" immediately inside polygon endpoints so that every point on the wall of the gallery can be seen by at least one guard.

a) Show that for every n, there are polygons which require $\lfloor n/3 \rfloor$ guards, where n is the number of endpoints of the polygon.

b) Show that every polygon with n points can be guarded by $\lfloor n/3 \rfloor$ guards. This problem, first posed by Klee [], was originally solved by Chvatal []; for your assignment, you may assume that every triangulation of the plane can be colored with 3 colors.

8) Show that $K_{n,n}$ - matching is a subset of a visibility graph on O(n) vertices.

9) Show that $K_{n,n}$ - matching has a covering set containing a total of O($n$log$n$) vertices.

10) Show that some graph is not an induced visibility graph. (Show Grotsch graph has no ordered chordal ordering?)

11) A vertex is called convex if the polygon edges touching the corresponding point form an angle of less than 180 degrees. The authors conjectured that the neighbors of any convex vertex in a visibility graph would induce a weakly chordal graph; that is, a graph with no induced cycle of length greater than 4 in G or its complement.

a) Show that it is possible to have a convex vertex v in a visibility graph such that N(v) does not induce a weakly chordal graph.

b) The conjecture proved false above was made because there are polynomial time algorithms for solving the clique problem on weakly triangulated graphs. Show that if the conjecture had been true, then there would have been a polynomial algorithm for solving the maximum clique problem on visibility graphs even if the model was not given as input. We note that this is not trivial, since removing a vertex from a visibility graph does not necessarily give a smaller visibility graph.

12) Prove or disprove: if G is a visibility graph with at least four vertices, there is a vertex x in G such that G - x is a visibility graph.

8 Graph Classes Defined by Intersection

Many interesting classes of graphs have come from requiring graphs to have two separate properties. We will study representation of a number of graph classes defined in this way, and introduce open problems relating to some of these classes.

When one takes the intersection of two graph classes, one does not know whether the number of graphs will stay approximately the size of the smaller class, or whether we will get far fewer graphs than we have in either class individually. When we get a much smaller class, it can be challenging to find a good representation of the new class, since we cannot use the representations of the larger graph class. Therefore, classes defined by intersection are good challenges for the implicit graph question.

Many graph classes defined by intersection have been studied in the literature. There are papers on intersection of classes defined by forbidden subgraphs (such as square and trumpet free graphs), classes defined by global properties (such as chordal co-comparability graphs), intersections of global and local properties (cubic planar graphs), and many more. This book can only study a small fraction of classes which have been defined by intersection. We choose to cover a 'consistent' set of classes, which come from a variety of approaches for dealing with interval graphs. We will not repeat material which is well covered in Golumbic [79]

8.1 Some Fundamental Properties

One of the oldest and most satisfying techniques for working with a class C of graphs is to note a property or set of properties which the class obviously has, and to determine whether the properties completely characterize C. Euler took the obvious observations that Eulerian graphs (without isolated vertices) were connected, and had no vertex of odd degree, and showed that these two properties exactly characterized Eulerian graphs. Many other useful characterizations have been discovered in this way.

Suppose that we view interval graphs with this approach in mind. One simple property is that interval graphs cannot contain any chordless cycles of length greater than 3.

G is chordal if and only if every cycle of length greater than 3 has a chord. For more on chordal graphs, see Golumbic's book [79]; we discuss issues in representing chordal graphs in chapter ?.

Two other obvious properties of interval graphs are used as parts of characterizations. If you have three independent vertices in an interval graph, one must be placed in the middle of the other two, and every path from the first to the third must pass through a neighbor of the middle vertex. A trio of independent vertices is called an asteroidal triple if for each pair of vertices, there is a path between the two which avoids all neighbors of the third. Thus, all interval graphs are clearly asteroidal triple-free graphs.

It is also natural to assign an ordering between nonoverlapping intervals, saying that one interval comes before another interval. This relationship is clearly transitive; if x comes before y and y comes before z, then x comes before z. Since these assignments are made exactly between nonadjacent edges of the interval graphs, we are saying that the complement graph has a transitive orientation. Graphs which have transitive orientations are called comparability graphs, and every interval graph is a co-comparability graph.

It turns out that these properties of interval graphs, which are quite easy to see, can be used to completely characterize interval graphs. [112] showed that a graph G is an interval graph if and only if G is chordal, and G is asteroidal triple-free. [77] showed that G is an interval graph if and only if G is chordal, and G is a co-comparability graph. In general graphs, the class of asteroidal triple-free graphs properly contains the class of co-comparability graphs (exercise ?)

The number of comparability graphs (and thus the number of co-comparability graphs as well) is $2^{\Theta(n^2)}$, since every bipartite graph is a comparability graph. Therefore, the implicit graph question is not relevant for these classes. Recognition of comparability graphs is discussed in chapter ?; to summarize, a transitive orientation of a comparability graph can be found in linear time, but the fastest recognition algorithm for the class takes time proportional to matrix multiplication.

Asteroidal triple-free graphs were first studied as an independent class of graphs in [41], and a number of interesting properties of the class have been found [40], [106]. Let us look briefly at the problem of recognizing asteroidal triple-free graphs, before we deal with combinations of the properties studied here. The reader is asked to provide an $O(n^3)$ algorithm for recognizing asteroidal triple-free graphs in exercise ?. We will show that recognizing asteroidal triple-free graphs is as hard as recognizing graphs without an independent set of size three; thus, any algorithm which did better than matrix multiplication for recognizing graphs in this class would have far reaching implications.

Let G be a graph, and suppose that we want to test whether G has an independent set of size 3. Create graph G' by adding n vertices to G, where each v_i is adjacent to every vertex of G' with the exception of the ith vertex of G. If G has an independent set v, w, x of size 3, then G' has an asteroidal triple, since there are vertices of G' which are nonneighbors of v only, w only, and x only. No new vertices of G' can be in an independent set of size 3, so if G' has an asteroidal triple v, w, x, then v, w and x must form an independent set of size 3 in G. The problem of determining whether G has an independent set of size 3 has been reduced to the problem of determining whether a graph on $2n$ vertices is asteroidal triple-free, which implies that any algorithm of $O(n^k)$ where $k < \alpha$ (the exponent in matrix multiplication of 2 n by n matrices) would improve the time complexity of determining whether a graph has an independent set of size 3.

Thus, finding an $O(n^2)$ algorithm for recognizing asteroidal triple-free graphs may be a very difficult problem. As an open problem, the following question seems more tractable.

Open Problem: Can you find an algorithm which recognizes asteroidal triple-free graphs in time proportional to matrix multiplication?

A related open problem asks whether there is a good 'certificate' that an input graph is AT-free. At this point, there seems to be no certificate which can be verified in $o(n^3)$ time. On the other hand, there is no reason to assume that a very good certificate does not exist; note that even the apparently much harder Hamiltonian graph recognition problem has a certificate which can be verified in $O(n)$ time.

Recently, the paper [27] shows that the independent set problem can be solved in $O(n^4)$ time on asteroidal triple-free graphs, while the clique problem is NP-complete on the class.

In addition to these properties, we will consider an important generalization of chordality. G is weakly chordal if neither G nor the complement of G contains a chordless cycle of length greater than 4. Thus, chordless 4 cycles are allowed in weakly chordal graphs, but longer chordless cycles are not. Forbidding longer chordless cycles in the complement as well makes it possible to solve optimization problems on the graph efficiently [95, 150] and weakly chordal graphs are a subclass of the perfect graphs [94]. In exercise ?, the reader is asked to show that chordal graphs are a subclass of weakly chordal graphs. The fastest known recognition algorithm for weakly chordal graphs takes $O(n^4)$ time [150]; this algorithm is discussed in chapter ?.

8.2 Intersections of Fundamental Properties

Combinations of chordality, co-chordality, comparability and co-comparability are well covered in Golumbic, with the exception of chordal comparability graphs which we treat in chapter ?. Golumbic's books devotes chapters to chordal co-chordal (split) graphs, chordal co-comparability (interval) graphs, and comparability co-comparability (permutation) graphs. In this chapter we look at what happens as we generalize from chordality to weakly chordal, and from co-comparability to asteroidal triple-free graphs.

Before dealing with combinations of properties from the previous section, let us briefly consider classes obtained by taking the intersection of the fundamental properties and bipartite graphs. Intersecting chordal graphs and bipartite graphs gives us the class of forests, while comparability intersect bipartite leaves us with the bipartite graphs. Bipartite co-comparability graphs are equal to bipartite permutation graphs (exercise ?), and bipartite co-chordal graphs are discussed in exercise ?. Weakly chordal bipartite graphs are equal to the class called chordal bipartite (bipartite and no chordless cycles of length > 4) discussed in chapter ? (exercise ?). Bipartite graphs are always co-AT-free, since the complement has no independent set of size 3.

The hardest relationship to see is the equivalence between bipartite AT-free graphs and bipartite permutation graphs. Every permutation graph is a co-comparability graph, and co-comparability graphs are a subclass of AT-free graphs, so it is clear that bipartite permutation graphs are a subset of bipartite AT-free graphs. The

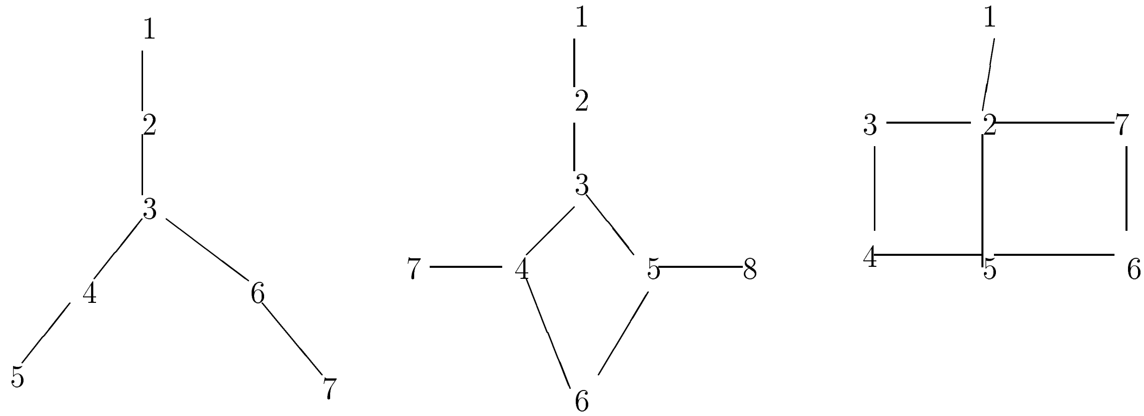


Figure 9: Forbidden Subgraphs

simplest way to show the equivalence is to use a forbidden subgraph characterization of comparability graphs due to Gallai [70]; the forbidden subgraphs can also be found in [167]. If G is not a co-comparability graph, then G is either not bipartite, G contains a cycle of length at least 6 or G contains one of the three graphs in figure ?. Since the graphs of figure ? and long cycles have asteroidal triples, every bipartite graph is either a co-comparability graph or has an asteroidal triple.

Our first generalization takes the characterization of interval graphs as chordal co-comparability graphs, and replaces the chordality condition with a requirement that the graph is weakly chordal. Since the complement of every weakly chordal graph is weakly chordal, the complement of this class is the class of weakly chordal comparability graphs. We will show that recognition, representation, and optimization problems on this class can be effectively transformed to the same problem on the subclass of chordal bipartite graphs.

Let G be a directed graph. Consider the following natural way of transforming a directed graph into a bipartite graph. Let $\mathrm{B}(G)$ be the graph obtained by making two copies, v_1 and v_2, for every vertex v of G, and adding an edge between w_1 and x_2 if and only if there is an edge from v to w in G. This transformation was used by Ford and Fulkerson for finding the maximum independent set in a partial order [67].

Theorem: Let G be a comparability graph, and let G' be a transitive orientation of G. Then $\mathrm{B}(G')$ is chordal bipartite if and only if G is weakly chordal. [58]

Proof: Suppose that $\mathrm{B}(G)$ has a chordless cycle C of length at least 6. We will show that C corresponds to a chordless cycle in G. To do this, we must rule out the possibility that C contains two vertices which map to a single vertex of G, and the possibility of edges of G between two vertices from the same color class in $\mathrm{B}(G)$.

Suppose that there is some vertex v such that both v_1 and v_2 are in C; there must be an edge from some c_1 in C to v_2, which implies by the transitivity of G' that $\mathrm{N}(c_1)$ contains $\mathrm{N}(v_1)$, so v_1 and c_1 could not both be in C.

Suppose that there is an edge between x and y in G, where x_1 and y_1 are in C.

Without loss of generality, assume that $x \to y$ in G'. By the transitivity of G', $\mathrm{N}(x_1)$ contains $\mathrm{N}(y_1)$, which cannot be true for a pair of vertices in a chordless cycle. A similar argument shows that there is no edge of G between two vertices x_2, y_2 of C.

Thus, if C has k vertices in $\mathrm{B}(G)$, these correspond to k distinct vertices of G, and the edges between the vertices in G are the same as the edges in $\mathrm{B}(G)$. The vertices of C form a chordless cycle of length at least 6 in G, so G is not weakly chordal.

The complement of a comparability graph cannot have any induced cycles of length greater than 4, and comparability graphs can only have induced cycles of even length, except for triangles [70]. Suppose that G has an induced cycle $C = c_1,c_2,...,c_2k,c_1$ of length at least 5.

Without loss of generality, assume that the edge (c_1,c_2) is directed from c_1 to c_2 in G'. This will force all edges of C to be directed from odd numbered vertices to even numbered vertices in G'. Then $\mathrm{B}(G)$ contains the cycle c_11,c_22,c_31, $c_42,...,c_2k2,c_11$, and $\mathrm{B}(G)$ is not chordal bipartite.

Therefore, $\mathrm{B}(G)$ contains an induced cycle of length at least 6 if and only if G is weakly chordal. QED

This theorem means that a weakly chordal comparability graph can be represented using any representation scheme for a chordal bipartite graph. If there is a generalized implicit representation of chordal bipartite graphs, we can take any weakly chordal comparability graph G, find a generalized implicit representation of $\mathrm{B}(G)$ as above, and store the bits of x_1 and x_2 with each vertex x of G. To test whether x and y are adjacent, we check whether x_1 and y_2 or x_2 and y_1 are adjacent in $\mathrm{B}(G)$. Since every chordal bipartite graph is weakly chordal, the questions of whether a generalized implicit representation can be found for the two classes are equivalent.

Theorem ? gives an immediate improvement in the time complexity of recognizing weakly chordal comparability graphs. The obvious algorithm has a bottleneck step of recognizing weakly chordal graphs, which takes $\mathrm{O}(n^4)$ time. Theorem ? reduces this to recognizing comparability graphs, and recognizing chordal bipartite graphs. The bottleneck becomes recognizing comparability graphs, which takes O(MM) time.

We will reduce this time complexity further by eliminating the need for verifying that G is a comparability graph. We cannot simply eliminate this step; there are many orientations of noncomparability graphs such that the bipartite transformation is chordal bipartite (exercise ?).

We can improve the time complexity by first testing whether the bipartite transformation $\mathrm{B}(G)$ is chordal bipartite, and if it is, using properties of chordal bipartite graphs to help check the transitivity of the orientation. The new ordering of steps is as follows. Given G, find the candidate transitive orientation G', and the bipartite transformation $\mathrm{B}(G)$ derived from this. If $\mathrm{B}(G)$ is not chordal bipartite, answer that G is not a weakly chordal comparability graph. If $\mathrm{B}(G)$ is chordal bipartite, we construct a Γ-free matrix representing $\mathrm{B}(G)$ using doubly lexical orderering (see chapter ?), and look for transitivity violations in G' as follows.

Consider each edge $x \to y$; we must verify that $\mathrm{N}(y)$ is a subset of $\mathrm{N}(x)$ or the orientation is not transitive. If x and y have the same outneighbors in G' (this can easily be checked via preprocessing) then there can be no transitivity violations of the form $x \to y \to z$. Doubly lexical ordering ensures the property that if $\mathrm{N}(x)$ is a proper subset of $\mathrm{N}(y)$, then x comes after y in the matrix ordering. Therefore, if x comes before y in the ordering and they do not have the same neighborhood, we answer that G is not a comparability graph. If x comes after y in the matrix ordering, let z be the first column which has a 1 entry in the row corresponding to y. If $\mathrm{M}[x,z] = 0$, then there are edges $x \to y \to z$, and we answer that G is not a comparability graph. If $\mathrm{M}[x,z] = 1$, then $\mathrm{N}(y)$ is a subset of $\mathrm{N}(x)$, since any u which has an edge from y and not from x comes after z in the ordering, meaning that the rows x,y and columns z,u would induce a Γ. If we do not find any violations after checking each edge, we answer that G is a weakly chordal comparability graph.

Each edge is checked in constant time for violations. Therefore, weakly chordal comparability graphs can be recognized in $\mathrm{O}(n^2)$ time. Other problems which can be solved efficiently on this class by using the fact that the bipartite transformation is chordal bipartite can be found in [58].

Other generalizations of fundamental classes have not been studied at this point. We introduce a number of natural new classes defined in this way; counting, representation, and efficient recognition algorithms are all worthy of study for the classes.

The classes which it seems most natural to generalize in this way are the split graphs (chordal co-chordal), interval graphs (chordal co-comparability or chordal AT-free), permutation graphs (comparability co-comparability) from Golumbic, and the chordal comparability graphs of chapter ?. Split graphs do not generalize to a new class by generalizing from chordality to weakly chordal, since weakly chordal graphs contain both the chordal and co-chordal graphs, and weakly chordal co-weakly chordal gives simply the weakly chordal graphs.

Suppose that we want to generalize the class of interval graphs. We have generalized the chordal co-comparability characterization to weakly chordal co-comparability (actually, we dealt with the complement of this class) above. Generalizing this characterization to chordal AT-free simply gives us the other characterization of interval graphs. However, we get a new graph class if we take the chordal AT-free characterization, generalizing from chordal to weakly chordal; thus, we can study the weakly chordal AT-free graphs.

We now look at generalizations of permutation graphs, using the fact that these are exactly equal to comparability co-comparability graphs. We can generalize either G and/or its complement from comparability graph to AT-free graph. If we ignore classes which are complements of each other, we get two new classes; AT-free comparability graphs and AT-free co-AT-free graphs.

Generalizing chordal comparability graphs gives us weakly chordal comparability graphs, which we studied earlier. We can also consider generalizing the complement class, and studying co-chordal AT-free graphs.

There are many open problems on these classes of graphs, which have not been studied at this point. Fundamentally, do they behave like the more restricted classes when we deal with problems of counting, representing, and optimizing, or do we get classes which are much larger and harder to deal with? We will make some observations about these classes, and pose open problems related to them.

We first discuss the class of AT-free co-AT-free graphs. It can be seen easily that this class of graphs is not perfect, since C_5 is in the class. However, there cannot be any longer cycles in either G or the complement of G, since every cycle of length at least 6 contains asteroidal triples.

We will show that if the number of graphs in the class is large, it is not because of cycles of length 5. We will use $\mathrm{O}(n\log n)$ bits, and derive a graph G' which has no cycles of length 5, such that G can be reconstructed from G' plus the $\mathrm{O}(n\log n)$ bits. Thus, the counting problem for AT-free co-AT-free graphs will be reduced to the counting problem for AT-free, co-AT-free, weakly chordal graphs, though representation questions on the two classes can be separate. Our fundamental approach is to find one vertex that can be eliminated from a C_5 so that adjacencies from this vertex can be determined completely from adjacencies to other vertices in the C_5.

We will present this material so that the algorithm for storing the graph is as simple as possible, leaving out the proof that the storage method is correct. We will follow the presentation of the storage method with statements of a series of lemmas which can be used to prove the validity of the method. The proofs themselves do not use any interesting techniques, in each case showing that if the statement of the lemma is false then there is an asteroidal triple in G or the complement of G. These proofs are therefore omitted; the interested reader should be able to construct each proof.

Let C be an induced $C_5 = 1,2,3,4,5,1$ in an AT-free co-AT-free graph. The clones of a vertex i, denoted clones(i), of C are the set of vertices which have the same adjacencies to other members of C as i. For example, clones of 1 are the set of vertices in G which are adjacent to 5 and 2, and nonadjacent to 3 and 4; they may be adjacent or nonadjacent to 1. Clones of different vertices i, j of C are adjacent if and only if i and j are adjacent (unproved lemma ?).

If G is AT-free and co-AT-free, a vertex v of G can have the following relationships to C and the clones of C (unproved lemma ?).

1) v is a clone of some vertex in C.

2) v is adjacent to no clone of any vertex of C.

3) v is adjacent to every clone of every vertex of C.

4) v is adjacent to some of the clones of exactly one vertex of of C.

5) v is nonadjacent to some of the clones of exactly one vertex of C.

A set S of clones is called a unique neighbor set if there is some type 4 vertex above which is adjacent to a vertex in S. Similarly, a set S of clones is called a unique nonneighbor set if some type 5 vertex is nonadjacent to a vertex in S. Any set of clones which is not a unique neighbor set or a unique nonneighbor set is a module, and can be reduced to a single vertex by preprocessing the graph using the substitution decomposition.

A vertex c of C is called reconstructible if it satisfies the following conditions.

1) The clones of c are not a unique neighbor set or a unique nonneighbor set.

2) If the clones of a vertex c_2 at distance 2 on the cycle from c are a unique neighbor set, then clones of the vertex c_1 between c and c_2 on C are neither a unique neighbor set nor a unique nonneighbor set.

3) If the clones of a vertex c_1 at distance 1 from c are a unique nonneighbor set, then the clones of the vertex c_3 of C which is nonadjacent to both c and c_1 are neither a unique neighbor set nor a unique nonneighbor set.

The fact that every induced 5-cycle of an AT-free co-AT-free graph has a reconstructible vertex is the key to our storage scheme; let us discuss how this apparently strange looking definition arises naturally. We want to be able to reconstruct adjacencies to c from adjacencies to other vertices in C. Suppose that we store the vertices of C along with a reconstructible vertex c, and look at what we can deduce for each type of vertex relationships to the cycle. We will also store whether clones of each vertex of C - c form a unique neighbor and/or unique nonneighbor set with respect to C. We assume that G has been preprocessed according to the substitution decomposition; thus we may assume that G has no nontrivial modules. When we discuss vertices of the cycle, we will assume that the cycle is ordered c,c_2,c_3,c_4,c_5.

First, assume that v is nonadjacent to every clone of every vertex of C. We can determine by examining relationships between v and C - c that v is either nonadjacent to every clone of every vertex of C, or that v is adjacent only to c. Since we know that the clones of c are not a unique neighbor set, we conclude that v is nonadjacent to c. Similar reasoning lets us conclude that if v is a neighbor of every vertex of every clone of C, then v is adjacent to c. Let v be a clone of some vertex of C.

Let us now assume that v is adjacent to clones of exactly one vertex c_i in C - c. If c_i is a neighbor (we will assume $i = 2$) of c, we can conclude that v and c are nonadjacent, since no vertex is adjacent to clones of exactly two consecutive vertices of C. If c_i is a nonneighbor (we will assume $i = 3$) of c, we seem to have two possibilities; v could be a neighbor of c and thus a clone of c_2, or v could be nonadjacent to c, with c_3 as a unique neighbor of v. This is the reason for adding a condition in our definition of reconstructible vertices; if clones of c_3 are a unique neighbor set at distance 2 from c, then the clones of c_2 form a module and are eliminated during a preprocessing step. If clones of c_3 form a unique neighbor set, then v cannot be a clone of c_2, and we deduce that v is nonadjacent to c. If clones of c_3 do not form a unique neighbor set, then v must be adjacent to c.

Suppose that v is adjacent to clones of exactly two vertices in C - c. Then v must be a clone vertex, and cannot be a clone of c. If v is adjacent to c_2 and c_3 or c_4 and c_5, then v is adjacent to c; otherwise, v is nonadjacent to c.

The case where v is adjacent to clones of exactly 3 vertices in C - c is analogous to the case in which v is adjacent to clones of exactly 1 vertex, so we will not present the arguments in detail. If the three vertices are not consecutive, then v must be adjacent to c. Otherwise, assume that v is adjacent to clones of c_2, c_3, and c_4. If clones of c_5 form a unique nonneighbor set, then c is adjacent to v (since c_3 cannot have clones); otherwise, c is nonadjacent to v (since c_5 cannot be a unique nonneighbor set).

Therefore, we can deduce the neighbors of c from the graph induced by V - c and the O(log$n$) bits stored with C. This gives a relatively complex elimination scheme for transforming an AT-free co-AT-free graph G into a weakly chordal AT-free co-AT-free graph G' using O($n$log$n$) bits, so that G can be reconstructed from this information. This implies that if the number of AT-free co-AT-free graphs is $2^{\Theta(f(n))}$, then the number of weakly chordal AT-free co-AT-free graphs is also $2^{\Theta(f(n))}$.

We give a series of lemmas which can be used to prove that this representation works. The proofs of these lemmas are the subjects of exercise ?.

Lemma ?: Let C be an induced cycle of an AT-free co-AT-free graph. Clones of different vertices i,j of C are adjacent if and only if i and j are adjacent.

Lemma ?: Let C be an induced 5-cycle in an AT-free co-AT-free graph. Every vertex v is either adjacent to no clones of any vertex in C, adjacent to all clones of every vertex of c, is a clone of some vertex in C, is adjacent to some clones of exactly one vertex in C, or is nonadjacent to some clones of exactly 1 vertex of C.

Lemma ?: If there are two unique neighborhood sets with respect to C, these must be clones of nonadjacent vertices of C. If there are two unique nonneighborhood sets with respect to C, these must be clones of adjacent vertices of C.

Lemma ?: If C is an induced 5-cycle of an AT-free co-AT-free graph, then C contains a reconstructible vertex.

The construction above does not resolve the counting question for AT-free co-AT-free graphs, but it may allow us to focus our attention on a possibly more tractable class, the weakly chordal AT-free co-AT-free graphs. It is also possible that the restricted types of relationships to 5-cycles can lead to efficient optimization algorithms for this class of graphs, which seem to be in some sense 'barely' not perfect, in the sense of having only restricted forms of 5-holes, and no longer holes in the graph or its complement. Counting, geometric characterization, generalized implicit representation, and recognition (faster than by recognizing AT-free graphs) is also open for this class.

The author will add one approach for dealing with this class which fits with the rest of this chapter. One of the major tools for dealing with asteroidal triple-free graphs is to find an ordering by performing a pair of lexicographic breadth first searches on G; the last vertex of the first lexicographic search is used as the start vertex for a second lexicographic breadth first search. This technique is used, for example, to find a dominating pair of vertices in an AT-free graph.

We can try to use what we know about weakly chordal comparability graphs in order to represent weakly chordal AT-free co-AT-free graphs efficiently. For weakly chordal comparability graphs, we take a transitive orientation G' of G, find the bipartite transformation B(G), and show that B(G) is chordal bipartite. As we show above, if G is a comparability graph, B(G) is chordal bipartite if and only if G is weakly chordal. However, if we have a nontransitive orientation of G, the bipartite transformation of other graphs may also be chordal bipartite. Suppose that we orient edges of G so that $x \to y$ if x comes before y in the ordering found by performing lexicographic breadth first search twice (note to self: or is the ordering I want on the complement graph?). Is the result always chordal bipartite? If it is, we can use the bounds on the number of chordal bipartite graphs from chapter ? to show that there are at most $2^{O(nlog^2 n)}$ AT-free co-AT-free graphs. This bound would not necessarily be tight; the AT-free co-AT-free graphs contain the permutation graphs, but there are only $2^{O(nlogn)}$ permutation graphs.

Open Problems: What is the number of AT-free co-AT-free graphs? Find a generalized implicit representation for the class. Can these be recognized in o(n^3) time? Find a representation which exactly characterizes the class.

The problems have been stated for AT-free co-AT-free graphs. Equivalent problems are open for the other classes mentioned as well; weakly chordal AT-free co-AT-free graphs, weakly chordal AT-free graphs, AT-free comparability graphs (note to myself: look at this class! may be easy NOTE: there is a conjecture that these are co-trapezoid or at least bounded interval dimension), co-chordal AT-free graphs.

The class of AT-free graphs has received a good deal of attention recently; this is in part because they are a large class of graphs which can be recognized in polynomial time, and maintain some of the nice properties of interval graphs. I believe that the following class deserves study as well.

Let L be an ordered list of vertices of a graph G. L is an AT-destroyer if for every independent set of 3 vertices $x < y < z$ in L, every path from x to z goes through a neighbor of y.

Although every graph with an AT-destroyer is AT-free, there are AT-free graphs which have no AT-destroyer (reference?). The class of AT-free graphs is sometimes justified in terms of an indication of a possible 'linear structure'; perhaps the existence of an AT-destroyer is a more clear indication of a linear structure.

Open Problem: Is there a polynomial time algorithm which determines whether an input graph has an AT-destroyer, and can such a destroyer be found efficiently?

8.3 Exercises

1) Show that every co-comparability graph is asteroidal triple-free.

2) Show that there are asteroidal triple-free graphs which are not co-comparability graphs.

3) Show that the comparability graphs are exactly equal to the containment graphs

of substars of a star.

4) Design an $O(n^3)$ algorithm for recognizing asteroidal triple-free graphs.

5) Show that if the complement of G contains a chordless cycle of length greater than 4, then G contains a chordless cycle of length at least 4.

6) Explain why problem 5 shows that chordal graphs are a subclass of weakly chordal graphs.

7) Show that the class of bipartite co-comparability graphs is equal to the class of bipartite permutation graphs.

8) Show that a bipartite graph $G = X,Y,E$ is co-chordal if and only if the neighborhoods of color class X are ordered by set containment.

9) Show that G is weakly chordal and bipartite if and only if G is a chordal bipartite graph; i.e., G is bipartite and has no induced cycle of length > 4.

10) Give an example of an orientation of a noncomparability graph such that the bipartite transformation is chordal bipartite.

11) Every AT-free comparability graph is weakly chordal, so the transformation must give a chordal bipartite graph. Some well known subclasses of chordal bipartite graphs are bipartite permutation graphs, biconvex graphs (you can order each color class such that neighbors of all vertices in the other class appear consecutively), and convex graphs (one color class can be ordered as above). Show that it is not true that G is an AT-free comparability graph iff the transformation produces a graph in class C, for $C =$ any of bipartite permutation graphs, biconvex graphs, convex graphs, chordal bipartite graphs.

For problems 12 and 13, you will need to use properties of chordal bipartite graphs, as discussed in chapter 10. In particular, you need to know that a Γ-free ordering of the bipartite adjacency matrix can be constructed in $O(n^2)$ time.

12) Design an $O(n^2)$ algorithm for solving the maximum independent set problem on weakly chordal comparability graphs.

13) Design an $O(n^2 \log n)$ algorithm for computing the transitive closure or reduction of a weakly chordal partial order.

14a-d) Prove each of the unproved lemmas leading to the existence of a reconstructible vertex from any 5-cycle in an AT-free co-AT-free graph.

15) Find a graph which is weakly chordal, AT-free, co-AT-free, and is not a permutation graph.

16) Show that chordal P_5-free graphs can be recognized in $O(n^3)$ time.

17) Show that the number of co-chordal triangle-free graphs is $2^{O(nlogn)}$

9 Graph Classes Defined by Forbidden Subgraphs

One common way to define classes of graphs is to specifically exclude certain subgraphs. Almost every small subgraph, and many combinations of several small subgraphs, have been studied in some context. Classes defined by forbidding a fixed of set induced subgraphs will all be hereditary, but there is nothing in the definition which will naturally lead to a good representation. Therefore, these classes are particularly interesting with respect to the implicit graph question.

In the interests of space, we look primarily at graph classes defined by single forbidden induced subgraphs. The key forbidden subgraph here is P_4. If a forbidden subgraph F is a subset of P_4, then the class defined by forbidding F as an induced subgraph has an implicit representation. If F is not a subset of P_4, then the class defined by forbidding F as an induced subgraph contains $2^{\Theta(n^2)}$ members. Those readers who are interested in more precise bounds on the constants in front of the Θ for individual graphs can find more information in [135]. We will discuss the class defined by forbidding P_4 in some detail, then move on to some interesting questions on graph classes which have too many members for the representation problem to be interesting. We will discuss counting, representation, construction, recognition, and optimization problems for various classes in this chapter.

It is interesting to note that the size of poset families defined by forbidding single induced posets has very different behavior than the corresponding questions on graphs. Forbidding each induced height 2 subposet gives a class of posets with $2^{o(n^2)}$ members, and there are many classes defined in this way which have more than 2^{cnlogn} members for any value of c. Results and many open counting problems of this form are contained in [26].

9.1 Cographs

One particularly tractable class of graphs is the class of graphs without any induced P_4. These have been characterized in a number of different ways, and there are efficient algorithms for a wide range of problems which are NP-complete on general graphs [].

Graphs without induced P_4s are commonly called cographs because of the following characterization; essentially, it says that P_4-free graphs are exactly equal to the set of graphs which can be built from single vertex graphs using only the union and complement operations. More formally, the class of cographs is defined as follows.

1) A single vertex is a cograph.

2) If G_1 and G_2 are cographs, then $G_1 \cup G_2$ is a cograph.

3) If G is a cograph, then the complement of G is a cograph.

The equivalence between cographs and graphs without induced P_4s is proved in [113].

It can be shown (see problem ?) that every cograph on n vertices can be constructed using O(n) of the above operations. Each operation can be described using O(log$n$) bits, and the graph can be reconstructed from the sequence of operations. Thus, the number of cographs is $2^{O(nlogn)}$. Representing a cograph by giving a list of these operations has serious drawbacks; adjacency testing would be relatively expensive. Therefore, the question of whether an implicit representation of cographs exists is natural.

Another representation of cographs is used frequently. A *cotree* for G is a tree T such that vertices correspond to leaves in the tree, internal nodes have degree at least 2 and are labeled 0 or 1, and vertices x and y of G are adjacent if and only if the least common ancestor of x and y is labeled 1. The proof that G is a cograph if and only if G has a cotree representation is given in [113]. Cotrees correspond to substitution decomposition trees for cographs, as studied in chapter ?, where series module nodes of the decomposition tree are labeled 1 and parallel nodes are labeled 0. Thus, cographs are also equivalent to graphs which have no neighborhood modules in their substitution decomposition tree.

Cotrees can be used to design efficient algorithms for many problems on cographs. The fundamental technique is to perform a postorder traversal, and to use the fact that relationships between vertices from different children of the same node are very easy to characterize. For example, suppose that we want to solve the maximum clique problem on cographs using the cotree representation. Mark all leaves as having maximum clique size 1. As we process a cotree in postorder, if we encounter a 1 node v, the maximum clique on descendants of v is equal to the sum of the weight of the maximum cliques of its children, and if v is a 0 node, then the maximum clique on descendants of v is the maximum value of the clique size of any children. Problems which can be solved using this strategy are given in [].

Representing a graph by giving its cotree uses O($n$log$n$) bits. Once again, testing whether x and y are adjacent is not necessarily efficient, as it requires determining the label of the least common ancestor of x and y. Of course, this is not an implicit representation, since determining whether x and y are adjacent involves looking at information which would not be stored at x or y.

However, it is easy to use the cotree representation of G to construct a representation of G as a permutation graph. The strategy involves using postorder traversal of the cotree as described for the clique problem above; the complete construction is left for exercise ?.

Since algorithms using the cotree for so many problems take O(n) time, it is clear that the bottleneck step for the construction and optimization problems is construction of the cotree. This may seem to be an easy problem, since the substitution decomposition tree can be constructed in linear time, and the cotree is a special case of the substitution decomposition tree. In fact, this argument is circular, since most known linear time substitution decomposition algorithms [123, 44], use the cotree construction algorithm as subroutines; an exception is the algorithm in [48]. We will describe below a linear time cograph recognition and cotree construction algorithm for cographs due to Corneil, Perl and Stewart [43]. The O(n^2) and O($m\log n$) algorithms for substitution decomposition given in chapter ? can also be used for cotree construction.

The algorithm is incremental, meaning that one vertex is added at each step, and the cotree representation of the current graph is found. Since cographs are hereditary, we stop if at any time we find the current graph has no cotree representation.

We simplify the discussion by assuming that G is connected (if not, the cotree can be constructed by joining all cotrees for connected components under a 0 node), and we add vertices so that the subgraph induced by vertices added at any step is always connected. This can be achieved by adding vertices in an order governed by an arbitrary spanning tree. This restriction allows us to assume that the root is always a 1 node. For some applications, it is desirable to add vertices on-line; the algorithm of Corneil, Perl and Stewart [43] can handle this case as well.

Start with the first u,v, and place them under a 1 node. We now consider the process of adding a new vertex x to a cotree T; we will call the resulting cotree T'.

We use a marking procedure on T as follows. Nodes can be unmarked, partially marked, or fully marked; a node is fully marked if it has been marked by all its children. Originally, leaves which correspond to neighbors of x are marked; since these have no children, they are considered to be fully marked.

Fully marked nodes are placed on a queue, and any fully marked node marks its parent. The result of the marking procedure can be seen in figure ? NOTE: delaying figures on cograph recognition, since I am considering an alternative approach to the problem using just modular decomposition. It is easy to do the marking in time proportional to the number of neighbors of x. Let α, β be the partially marked nodes of T which are at the lowest and second lowest possible levels respectively.

In [43], it is shown that the partially marked nodes, with the exception of α, must have the following structure if the new graph is a cograph. Let PM be the set of partially marked nodes other than α. If PM is empty, then G is a cograph. All vertices of PM must be 1 nodes marked by all but 1 child, and they must consist of every other node on the path from β to the root. α must be either a child or grandchild of β, unless β does not exist. An example is shown in figure ?.

If the conditions above are met, the new graph is a cograph, and x is added as follows. We note that once the reader understands the algorithm to this point, it may be more instructive to deduce the rules on ones own rather than to treat the rules as following a strange piece of pseudocode. If there are no partially marked nodes, x is added as a child of the root. Otherwise, there are 2 cases.

If α is a 0 node marked by at least 2 children, or a 1 node not marked by at least 2 children, split alpha by removing all nodes which marked α and place them under a new node y, giving y the same label as α. If α is a 0 node, place y and x under a new 1 node, and make this a child of α. If α is a 1 node, make a new 0 node with children α and x, and make the new node a child of y.

If α is a 0 node marked by only 1 child c, or a 1 node not marked by only 1 child c, we have two cases. If c is a leaf, replace c by a new node, which has the opposite label from α, and has c and x as children. If c is not a leaf, add x as a new child of c.

The steps of marking the tree and adding the new node to the tree take time proportional to the number of neighbors of x, if standard data structures are used. Data structures used would include a queue of fully marked nodes for the marking procedure, and for each internal node of the tree a variable which stores the number of children which have marked the node, and a list of these children. The time bound comes from the fact that work performed is proportional to the number of fully marked nodes of the tree, and the number of fully marked nodes is $\mathrm{O}(\mathrm{N}(x))$ (see exercise ?).

The only other step which could be a bottleneck for cotree construction is identification of the lowest level partially marked nodes α and β. These can be found by a separate labeling procedure, which takes place after partially marked nodes are identified. Each partially marked node labels its parent and grandparent. If G is a cograph, there will be exactly one partially marked node which is not labelled; this is the lowest level partially marked node α. β, if it exists, is the partially marked node which is the parent or grandparent of α. From this, it is easy to check that the partially marked nodes other than α are equal to the 1 nodes on the path from β to the root, and that these are marked by all but 1 child, by simply following the path from β to the root.

Many generalizations of cographs can be found in the literature, defined by severe restrictions on the number and types of P_4s in G. It is not immediately clear that all of these classes can be represented implicitly. [23] presents definitions and basic theorems for many graph classes defined in this way.

9.2 Classes Which are too Large to have Efficient Representations

In this section, we study classes of graphs defined by forbidden induced subgraphs which contain too many members to have efficient representations. We will point out a number of well known classes of this type, and discuss some natural problems which have not been asked about some of the classes. We will see that for any graph F which is not a subset of P_4, the graph class defined by forbidding F as an induced subgraph has $2^{\Theta(n^2)}$ members. Readers interested in a more precise evaluation of the growth rate of these classes of graphs can find a formula in [].

9.3 Triangle-free Graphs and Matrix Multiplication

We first discuss the class of triangle-free graphs. A graph is triangle-free if it does not contain the subgraph K_3. For this class, we do not need to make the distinction between containing subgraphs or containing induced subgraphs. Papers which deal with triangle-free graphs, and subclasses of triangle-free graphs, include [],[]. It is easy to see that the number of triangle-free graphs is $2^{\Theta(n^2)}$, since every bipartite graph is triangle-free. Therefore, the question of finding a generalized implicit representation for this class is not interesting.

Recognizing triangle-free graphs efficiently does raise some interesting questions, however. Before discussing the best known algorithm for this problem, we introduce a new piece of terminology.

The 0,1 multiplication of two 0/1 matrices A,B is what we get if we perform standard matrix multiplication of AxB, and replace all positive values with a 1. The author has sometimes called this form of multiplication Boolean multiplication in the past; however, Boolean multiplication has other meanings which could confuse the reader. Specifically, one might assume that Boolean arithmetic is being used, in which case $1 + 1 = 0$, so that the result of multiplying two 2 by 2 matrices which have all entries equal to 1 would produce an all 0 matrix, rather than the all 1 matrix desired here.

Triangle-free graphs can be recognized relatively easily in $\mathrm{O}(n^\alpha)$ time, where n^α is the time needed to perform 0,1 matrix multiplication of two n by n matrices. At the time of this writing, the best known upper bound for n^α is $\mathrm{O}(n^{2.376})$ [36] Let A be the adjacency matrix of G, and compute B = the result of the 0,1 multiplication of A x A. Then G has a triangle if and only if there is some pair of vertices x,y such that $\mathrm{A}[x, y] = 1$ and $\mathrm{B}[x, y] = 1$.

It is generally agreed that the asymptotically fastest matrix multiplication algorithms are not practical, since the constants in front of the n^α are astronomical []. There is also a less precise reason for objecting to using matrix multiplication on this problem. The steps performed in recognizing triangle-free graphs using matrix multiplication do not translate in any natural way to graph theoretic steps, and are completely divorced from any intuition about what is happening in the graph. The author, at least, feels that an algorithm which used (for example) matching as a subroutine to solve the triangle-free graph recognition in $\mathrm{O}(n^{2.5})$ time would be a very interesting result, even if it would not speed up the asymptotic time complexity of triangle-free graph recognition. Therefore, we pose two open problems.

Open Problem: Design a $\mathrm{o}(n^3)$ algorithm for recognizing triangle-free graphs which is completely graph theoretic; specifically, matrix multiplication routines should not be used.

Of course, an algorithm which used less time than matrix multiplication is an even more impressive than the result above! On the other hand, we are satisfied with time complexity proportional to matrix multiplication if the problem is actually tied to matrix multiplication. Transitive closure and transitive reduction of digraphs are graph theoretic properties, but it can be shown that they are essentially equivalent in time complexity to matrix multiplication [], and this is generally viewed as resolving the problem.

In the same way, we would be very interested in a result stating that triangle-free graph recognition is as hard as matrix multiplication. However, it is hard to imagine proving such a result, for the following reason. Triangle-free recognition is essentially a decision problem; there is one bit of output (yes or no). On the other hand, a matrix multiplication routine must produce n^2 bits of output. Since lower bound arguments are generally proved by reduction, it is hard to imagine that we could turn a matrix multiplication question into a question where the answer is yes or no; each answer yes and no must correspond to many matrix multiplication problems.

Therefore, we propose a potentially easier form of matrix multiplication, which we will call matrix multiplication verification. In this problem, we are given matrices A,B, and C, and are asked whether A x B = C.

Open Problem: Is triangle-free recognition as hard as 0,1 matrix multiplication verification?

Let us add a few notes on this problem. It is entirely possible that matrix multiplication verification is much easier than matrix multiplication. In fact, standard matrix multiplication can be verified in the randomized sense quite easily in O(n^2) time [], whereas a randomized O(n^2) matrix multiplication algorithm would be considered to be a great breakthrough. Since 0,1 multiplication cannot be harder than standard matrix multiplication, it may seem counterintuitive that 0,1 verification may well be harder than standard matrix multiplication verification.

To see how this might be true, let us look at a simple technique for randomly verifying standard matrix multiplication of two n by n matrices. Generate randomly a string of n 1s and -1, and call the resulting vector y. To check that A x B = C, we check whether (y x A) x B = y x C; the test uses O(n^2) scalar operations. If AB = C, the answer is always yes, by associativity of matrix multiplication. If AB is not equal to C, it will take a remarkable vector y to make the two results equal; formal arguments can be made that the likelihood of y being chosen to make this happen is extraordinarily small [].

To see why this is not as effective for verifying 0,1 matrix multiplication, look at the following example. Fundamentally, the problem is that all positive values in the result are compressed to a single value. Thus, while different positive values give errors with respect to standard matrix multiplication, they do not give errors in 0,1 multiplication. Suppose that you are multiplying 2 n by n matrices which contain all 1 entries, and you introduce an error by placing a 0 rather than a 1 in position 1,1 of the result matrix C. The result of y x C should be 1 everywhere if and only if there are more 1s than -1s in y; otherwise the result should be 0s everywhere. Introducing a 0 in position C[1,1] can only change the result of $y * C$ if the number of 1s differs from the number of -1s by at most 1; as n becomes large, this possibility becomes increasingly unlikely. The example above could be remedied by randomly generating strings with an equal number of 0s and 1s, but this is not enough to handle general 0,1 verification; see exercise ?. Verifying 0,1 matrix multiplication in randomized O(n^2) time would be a very useful and interesting result.

Many other graph theoretic problems have matrix multiplication verification as a bottleneck step. Some of the better known examples include recognizing graphs without simplicial vertices, comparability graphs, and diagrams of posets, and finding all shortest paths in an unweighted graph. We are making some attempts to organize these problems, in terms of which problems are reducible to each other, at the time of this writing, since these relationships have not been explored previously. note: Dieter suggests a good variant: is G diameter 2

9.4 Claw-Free Graphs

(include Kratsch et. al)

There is another well known class of graphs which has a more surprising gap in our knowledge with respect to the recognition problem. A graph is claw-free if it contains no induced subgraph isomorphic to $K_{1,3}$. Independent set and matching are among the set of problems which can be solved in polynomial time on claw-free graphs [],[]; there has also been work on subclasses of claw-free graphs [] and properties of claw-free graphs [].

Once again, it is easy to see that the number of claw-free graphs is $2^{\Theta(n^2)}$. A claw contains an independent set of size 3, so the complement of every bipartite graph is claw-free.

The author was not able to find any mention of the recognition problem for dense claw-free graphs in the literature, and the obvious bound for recognition in this case is so slow that it seems that it should be easy to improve. We will examine an obvious algorithm to give an upper bound, and give a lower bound on what one should hope to achieve without solving another well known open problem.

The obvious $\mathrm{O}(n^4)$ algorithm for this problem (test whether each subgraph of size 4 is a claw) can be improved easily if we allow the use of matrix multiplication. We can test whether G is claw-free by determining whether the complement of the graph induced by the neighborhood of each vertex is triangle-free. Each test can be done using matrix multiplication as above, so we can test whether a graph is claw-free using n matrix multiplications.

For a 'lower bound', we will show that recognizing claw-free graphs is at least as hard as recognizing triangle-free graphs, assuming that both problems require $\Omega(n^2)$ time. Suppose that we want to test whether G is triangle-free. Create a new graph G_2 by constructing the complement of G, and adding a new vertex which is adjacent to all vertices in the complement. G_2 has a claw if and only if G has a triangle.

The construction above is not, of course, a true lower bound; it is entirely possible that both of these problems can be solved in linear time. Nevertheless, even an ambitious graph algorithms specialist needs to know which improvements would cause improvements in algorithms for well known problems. Improving on n matrix multiplications for this problem seems quite feasible, whereas getting below 1 matrix multiplication may be quite difficult. Therefore, the open problem I will pose in this context is as follows.

Open Problem: Find an algorithm for recognizing claw-free graphs which takes

$o(n^{1+\alpha})$ time, where n^α is the time needed to perform 0/1 multiplication of two n by n matrices.

We will discuss optimization problems for claw-free graphs in chapter ?.

9.5 Forbidding Single Induced Subgraphs

One other class defined by small forbidden subgraphs is interesting, in order to prove that the only small classes of graphs defined by one forbidden subgraph are subclasses of cographs. G is square-free if and only if G contains no induced subgraph isomorphic to C_4.

The number of square-free graphs is $2^{\Theta(n^2)}$, since they contain the class of split graphs. Suppose that a split graph $G = (X,Y,E)$, where X is an independent set and Y is a clique, contains the C_4 u, v, w, x. At most two vertices of the C_4 are from X, since C_4 has no independent set of size 3; similarly, at most 2 vertices are from Y. The two vertices from Y must be neighbors on the cycle, but this would make the other two vertices neighbors, contradicting the fact that these vertices must be in X.

It is now easy to show that any graph F which is not an induced subgraph of P_4 defines a large class of graphs, since any graph is either an induced subgraph of P_4, or contains C_3, C_4, or their complements as induced subgraphs.

Theorem ?: Let F be a graph. If F is not an induced subgraph of P_4, then the graph class defined by forbidding F as an induced subgraph has $2^{\Theta(n^2)}$ members.

Proof: Consider any graph F such that F is not an induced subgraph of P_4, and the class defined by forbidding F as an induced subgraph does not have $2^{\Theta(n^2)}$ members. As shown above, F cannot contain K_3, or the class contains the bipartite graphs. F cannot contain C_4, or the class contains the split graphs. F cannot contain the complements of K_3 or C_4; the class would contain the complements of the corresponding classes (here the cobipartite graphs and the split graphs, since split graphs are self-complementary).

Every 3 vertex graph is an induced subgraph of P_4, C_3, or $3K_1$, so F contains at least 4 vertices u, v, w, x. Without loss of generality, since F does not contain 3 independent vertices, assume that u and v are adjacent. If neither w nor x is adjacent to u or v, then F either contains an independent set u,w,x of size 3, or $F = 2K_2$ which is the complement of C_4, and thus defines a class with $2^{\Theta(n^2)}$ members. Assume w is adjacent to v; then w is not adjacent to u, or F contains C_3. Vertex x is adjacent to either u or w, or u, w, x is an independent set of size 3; this implies that x is nonadjacent to v, or F would contain C_3. If x is adjacent to both u and w, then F contains C_4; thus we can assume that u,v,w,x are a P_4.

Since F is not an induced subgraph of P_4, F contains a fifth vertex y. Since u, x, y are not an independent set, we can assume without loss of generality that x and y are adjacent. This implies y is nonadjacent to x, or w, x, y induce C_3; this implies in turn that y is adjacent to u (or u,w,y are independent) and y is nonadjacent to v. The vertices induce a C_5, which means that the class defined by forbidding F as an induced subgraph contains the bipartite graphs. QED

9.6 Relation Between Recognition Problems

The following theorem, which seems to be stated first in [], is not very well known; basically, this says that the K_i-free graphs are at least as hard to recognize as the F-free graphs for any forbidden induced subgraph F on i vertices.

Theorem: Let F_i be any forbidden induced subgraph on i vertices. The problem of determining whether a graph G contains F_i as a forbidden induced subgraph can be reduced in O(im) time to the problem of determining whether a graph on O(in) vertices contains K_i.

Reduction: Order the vertices of F_i arbitrarily with numbers from 1 to i. Create a graph $G(F_i)$ as follows. Make i copies $v_1 ... v_i$ of each vertex v in G. Add an edge from v_j to w_k in $G(F_i)$ if j and k are distinct, v and w are distinct, and the relationship (i.e. adjacency or nonadjacency) between v and w in G is the same as the relationship between vertices j and k in F_i.

It is easy to see that a subgraph isomorphic to F_i in G maps to a clique of size i in $G(F_i)$, and that a clique of size i in $G(F_i)$ corresponds to a set of vertices isomorphic to F_i in G.

Other reductions of this type are also possible; exercise ? asks for a reduction showing that recognizing whether a graph contains C_k as an induced subgraph for $k > 4$ is at least as hard as recognizing whether a graph has a triangle. I know of no general theorem showing that with some specific exceptions (such as P_4) recognizing graphs without any fixed forbidden subgraph is at least as hard as triangle-free recognition; such a result seems intuitively very plausible.

9.7 Classes defined by Forbidding Sets of Induced Subgraphs

Many classes of graphs defined by sets of forbidden subgraphs have been defined; some examples are given in []. In general, these classes can form major challenges to the generalized implicit graph conjecture.

Scheinerman and Zito [145] raise the question of whether we can take as input a set of forbidden subgraphs, and solve the counting problem for the class; that is, we would want to find a function f(n) such that the number of graphs in the class is $2^{\Theta(f(n))}$. We do not even know how to solve this problem for sets of two forbidden subgraphs at this time (check; check also their statement of the problem). There are many reasonable approaches for progress on this problem; as well as attacking the problem for small number of forbidden subgraphs, it would be interesting to know whether the general problem is NP-hard, and if so how many different graphs are necessary to make this difficult. NP-hardness results would not end research in this direction, since it is reasonable to take exponential time in the size of your input (which may be simply two graphs with a total of a very small number of vertices) graphs for many classes of graphs we are interested in; a counting procedure which takes exponential time in the size of the input graphs could also be very interesting, and it is not obvious how to decide this problem in any time bound. We will present this as a single open problem.

Open Problem: Determine the difficulty of finding procedures which take a finite set of forbidden induced subgraphs as input, and solving the counting problem for the class of graphs defined.

Exercise ? deals with an attempt to generalize cographs in the same way that weakly triangulated graphs generalize chordal graphs; we allow P_4 but forbid P_5 in G or its complement, just as weakly triangulated graphs allow C_4 but forbid longer cycles in G or its complement. These graphs are studied in [].

There are many specific open classes defined by sets of forbidden subgraphs. For example, one can consider the class of graphs defined by forbidding $K_{3,3}$ and K_5 as induced subgraphs rather than as minors. Recognition, counting, and representation problems may be interesting on such a class; exercise ? shows that there are at least $2^{n^{1.5}}$ graphs in this class.

There are many other variants of forbidden subgraphs, involving issues such as forbidden minors, homeomorphs, and many others; [] gives a number of different notions and graph classes which fit these notions.

Errata: There is an excellent survey on graph classes in the Handbook of Algorithms and Theoretical Computer Science []. However, the time complexity given for recognition of claw-free graphs is linear, without any further comment. I have spoken with the author, and determined that this is a simple case of an exponent being dropped accidentally during the production of the book.

9.8 Exercises

1) Show that every cograph can be built using $O(n)$ operations.

2) Give an $O(n)$ algorithm for constructing a permutation graph representation of a cograph, if the cotree is given as input.

3) Design a linear time algorithm to determine whether a given vertex v is contained in a P_4. (not sure on this; check Habib)

4) Show that random = -1,1 doesnt work for 0,1 verification.

5) Recognize simplicial-free graphs in O(MM)

6) Show that the number of fully marked nodes in the cotree is proportional to the number of neighbors of the new vertex.

7) This problem deals with the class of graphs which contain no induced P_5 in the graph or its complement. It is clear that these graphs contain no holes or antiholes of length greater than 5. Let C be a 5-cycle 1,2,3,4,5 in such a graph.

a) Show that this class of graphs has $2^{\Theta(n^2)}$ members.

b) Show that the set of vertices adjacent to at least one and at most 4 vertices of C form a module M.

c) Show that M has exactly 5 maximal submodules, corresponding to 'clones' of each vertex in C.

d) Use the results from a and b and the fact that independent sets can be found in polynomial time on weakly chordal graphs to argue that independent set can be solved in polynomial time for graphs in the class.

8) One approach to working with classes defined by forbidden pairs of subgraphs is to take small graphs which do not rule out P_4s, and see what we get by forbidding pairs of these subgraphs. The smallest such graphs would be C_3 and its complement. Show that the class defined in this way has only a constant number of members.

9) Consider the class of graphs without any induced C_3 or C_4. Show that there are too many graphs in this class to have an implicit representation.

10) Show that triangle-free, claw-free graphs have an implicit representation.

11) Show that there are $2^{\Omega(n^2)}$ square-free claw-free graphs.

12) Show that there are $2^{\Omega(n^{1.5})}$ graphs which contain neither K_5 nor $K_{3,3}$ as induced subgraphs.

13) Show that recognizing whether a graph contains C_k for $k > 4$ is at least as hard as determining whether a graph has a triangle.

14a) Show that determining whether G has a triangle is at least as hard as recognizing comparability graphs.

14b) Show that recognizing comparability graphs is at least as hard as recognizing triangle-free graphs.

10 Chordal Bipartite Graphs

This chapter deals with chordal bipartite graphs and their generalizations. Chordal bipartite graphs are a useful and natural class of graphs with many interesting properties. We will see that these properties give rise to a form of representation which is fairly natural and is space optimal, but is not implicit. Chordal bipartite graphs are also interesting in that they are one of the only natural classes of graphs which are known to have more than $2^{\Omega(nlogn)}$ and fewer than $2^{o(n^2)}$ members on n vertices.

A graph G is chordal bipartite if G is bipartite, and every cycle of length at least 6 has a chord. The name comes from the idea that these are the bipartite analogue of chordal graph. G is chordal if every cycle which can have a chord (that is, every cycle of length at least 4) has a chord. In bipartite graphs, only cycles of length 6 can have chords, so requiring all cycles which can have chords to have them gives rise to the chordal bipartite graphs. They are analagous to chordal graphs in several other ways, involving separator properties [81] and elimination schemes (see exercise ?).

10.1 Γ-free Matrices

There is a single important idea which gives rise to interesting representations and efficient algorithms for chordal bipartite graphs. The exact origin of this idea is hard to pin down precisely; it has been attributed to [], [], and [], and I have been advised by several of these researchers that I should not try to unravel the mystery of whom to credit for the original discovery!

A Γ in a matrix M is a pair of rows i<j and columns k<l such that M[i,k] = M[i,l] = M[j,k] = 1 and M[j,l] = 0. A matrix is Γ-free if it has no Γ. A Γ-free ordering of a matrix is a permutation of the rows and columns of the original matrix such that the resulting matrix is Γ-free.

A graph G is chordal bipartite if and only if there is a Γ-free ordering of the bipartite adjacency matrix [],[],[]. One direction of this proof follows immediately from exercise ?, which shows that if the bipartite adjacency matrix is Γ-free, there is a vertex which is not in the center of a P_5. No such vertex can be in a hole of size $>$ 5, so that vertex can be removed without eliminating any chordless cycle of size $>$ 5; we can repeat the process until all vertices are eliminated.

Furthermore, a particular ordering scheme called doubly lexical ordering will always produce a Γ-free ordering for a chordal bipartite graph, which leads to efficient algorithms for recognizing chordal bipartite graphs, as will be discussed in a later section. Since the proof that chordal bipartite graphs have Γ-free orderings relies on the existence of a doubly lexical ordering algorithm, we defer this proof to a later section.

We should note that some of the work I cite here was done originally on the class of strongly chordal (also called sun-free chordal) graphs; these are graphs such that the augmented adjacency matrix has a Γ-free ordering, and are studied in [],[],[]. Since representations of each come from the Γ-free property, for the purpose of this book almost all theorems on one class translate directly to the other class. I will point out some places where the two classes differ slightly in a later section. I choose to work with chordal bipartite graphs exclusively, since we will introduce some natural generalizations which make sense in the bipartite context and give rise to interesting representation problems.

The Γ-free property is very useful in designing algorithms. It is easy to determine whether the neighborhood of one vertex contains the neighborhood of another vertex if G is chordal bipartite and we are given a doubly lexical ordering of the bipartite adjacency matrix. To simplify the description, we need to know this feature of a doubly lexically ordered matrix. If the neighborhood of vertex x properly contains the neighborhood of vertex y from the same color class, then the row or column number of x is greater than the row or column number of y.

Suppose that we want to know whether the neighborhood of vertex i is contained in the neighborhood of vertex j, where i comes before j in the doubly lexical ordering. We will assume that i and j are rows; there is an equivalent test if they both are columns. Let k be the first column such that $\mathrm{M}(i,k) = 1$. If $\mathrm{M}(j,k) = 0$, then clearly $\mathrm{N}(i)$ is not contained in $\mathrm{N}(j)$. The key point is that if M is Γ-free, then if M(j,k) = 1, then $\mathrm{N}(i)$ must be contained in $\mathrm{N}(j)$. This is true since if there is any later column l such that $\mathrm{M}(i,l) = 1$ and $\mathrm{M}(j,l) = 0$, then i, j, k, l induce a Γ. If we want to know whether $\mathrm{N}(j)$ is contained in $\mathrm{N}(i)$ where i comes before j, this is only possible if $\mathrm{N}(i) = \mathrm{N}(j)$, which can be tested by seeing whether $\mathrm{N}(i)$ is a subset of $\mathrm{N}(j)$ and their degrees are equal. Thus, neighborhood containmnent for a pair of vertices can be tested in constant time for a pair of vertices in a chordal bipartite graph, if we are given a doubly lexical ordering of the bipartite adjacency matrix.

The author has used this observation to speed up a number of algorithms on graphs which had neighborhood containment as a bottleneck step. Computing all neighborhood containment relationships on a general graph takes time proportional to matrix multiplication, currently $\mathrm{O}(n^{2.376})$ [36]. Computation of neighborhood containments was the key bottleneck step in recognizing circular- arc graphs [59]. We were able to transform the problem in this case to computing neighborhood containments on a chordal bipartite graph, achieving an $\mathrm{O}(n^2)$ algorithm for this problem. Cogis' algorithm for recognizing graphs with Ferrers dimension 2 [] had computation of containment relationships as a bottleneck steps, and we were able to show that this could be performed on a chordal bipartite graph, reducing the time complexity of Ferrers dimension 2 digraph recognition, interval dimension 2, or trapezoid graph recognition (these problems are all closely related [118]) to $\mathrm{O}(n^2)$. In chapter ?, we will see a close relationship between chordal bipartite graphs and weakly chordal comparability graphs, which allowed a number of problems including recognition, independent set, and transitive closure and reduction of the corresponding class of directed graphs to be solved efficiently on weakly triangulated comparability graphs [58]. This observation was also used in an algorithm to compute shortest paths efficiently in strongly chordal graphs [92].

10.2 Counting and Representation

In this section, we will see that Γ-free ordering is also crucial in designing a space optimal representation for chordal bipartite graphs. We will find that the number of chordal bipartite graphs is $2^{\Theta(nlog^2n)}$, and that there is a method of storing all the information necessary to reconstruct a Γ-free matrix using $O(nlog^2n)$ space.

We present the easier direction; we will show that the number of Γ-free n by n matrices (and hence the number of chordal bipartite graphs with n vertices on each side) is $2^{\Omega(nlog^2n)}$.

Theorem cbg.1: The number of chordal bipartite graphs is $2^{\Omega(nlog^2n)}$.

Let n be a power of 2. Consider how many n by n matrices we can construct using the following method. Divide the matrix into quadrants. Place 1s in every position of the bottom right quadrant. Choose any 'perfect matching' for the top left quadrant; that is, for each row we choose exactly 1 column in this quadrant to have a 1 entry, filling the rest with 0s, and making sure that no column has two 1 entries. The other two quadrants are constructed recursively using the same technique.

Each matrix constructed is Γ-free by induction. Clearly, a 1 by 1 matrix is Γ-free. Assume that all $k/2$ by $k/2$ matrices constructed using this technique are Γ-free, and let M be any k by k matrix built in this way. Suppose that M has a Γ; it should be clear that the Γ does not lie entirely within a single quadrant of M. Since there is only one 1 in each row and column of the upper left quadrant, the lower right entry of any Γ which crosses quadrants must be in the bottom right quadrant; this is impossible, since the bottom right quadrant contains no 0 entries.

There are $n!$ choices for the upper left quadrant. In the second stage of the recursion, there are 2 quadrants we are choosing, and we have $(n/2)!$ choices for the perfect matching. Continuing in this fashion, the number of matrices which can be constructed is $n! * (n/2)!^2 * (n/4)!^4 * ... * (n/n)!^n$.

Reversing the order and using the fact that $x! \geq x^{x/2}$, we see that this product is at least $1^n * (2^1)^{n/2} * 4^{2n/4} * 8^{4n/8} * ... * (n/2)^{n/4^2} * n^{n/2}$. The log of this product is $(n/2)*(0+1+2+3+...+\log n) = (n/2)(\log n(\log n+1))/2$. The logarithm of the number of matrices is at least a constant times $nlog^2n$, so the number of Γ-free n by n matrices is at least 2^{nlog^2n}. Since each matrix corresponds to a different bipartite adjacency matrix for a chordal bipartite graph, the number of chordal bipartite graphs on $2n$ vertices is $2^{\Omega(nlog^2n)}$.

The idea for a natural and space efficient representation for representing chordal bipartite graphs came from the idea of eliminating information from the Γ-free matrix which could be deduced in other ways. Consider storing the 2 by 2 matrix which has a 1 in all positions. We only need to know that the entries in positions (1,1), (1,2), and (2,1) have value 1; if we know that these 3 have value 1 and that the matrix is Γ-free, we can deduce that the entry in position (2,2) is Γ-free. Extending this idea, a 1 entry is redundant in our representation if it can be deduced from other 1s in the matrix. It can be shown [149] that all but $O(n\log n)$ 1 entries in the matrix are redundant, and that the entire Γ-free matrix can be reconstructed if we know the positions of these O($n$log$n$) 1 entries. Since the positions can be stored using O(log$n$) space per entry, we have a natural O($nlog^2n$) space representation. We will show how to find this set of nonredundant 1 entries efficiently, and how to reconstruct the graph from these entries. (skip the proof of number of nonredundant 1s entirely, or give a handwavy outline, or give full proof?)

Theorem: Every 1 entry such that replacing the value by a 0 causes a Γ is redundant.

Proof: Suppose that we are given the set of 1 entries in a Γ- free matrix M which do not create Γs if they are replaced by 0s. We reconstruct M as follows. Process the entries of M in lexicographic order; that is, the order (1,1),(1,2), ... ,(1,n),(2,1), ... (2,n), ..., (n,1),(n,n). If the entry is not known to be 0 or 1 at the time it is processed, we make it a 0. If the entry (i, j) is known to be a 1 (either from the original set or from earlier in the procedure), find the next known 1 in row i at position (i, k) and the next known 1 in column j at position (h, j), and add a 1 in position (h, k). The proof that every 1 entry we add is necessary for the result to be Γ-free is simple, and is left to the exercises.

Let (i, j) be the lexicographically first 1 entry which is given the value 0 by the algorithm above, when it should have the value 1. Then replacing (i, j) with a 0 must create a Γ in M. Consider a 'minimal' Γ caused by replacing (i, j) with 0. Formally, a lexicographically largest pair (g, h) such that rows g and i, columns h and j form a Γ if (i, j) is replaced by 0 will suffice. Informally, we want a Γ so that the rectangle defined by the Γ does not contain the rectangle defined by any other Γ. Clearly $g <$ i, $h < j$.

The algorithm will find a 1 in position (g, h), since (i, j) is the lexicographically first 1 value which is given a 0 by the algorithm. We will show that the algorithm fills in (i, j) as 1 when (g,h) is examined. There cannot be any 1 in row g between columns h and j. Suppose (g, x) has a 1 entry where column x is between h and j. Then (i, x) must be a 1, or rows g, i and columns h, x induce a Γ. Replacing (i, j) by 0 creates a Γ induced by rows g and i, columns x and j; this is a smaller Γ than the Γ induced by g, i, h, j, contradicting the assumption that the Γ g, h, i, j is minimal. A similar argument shows that there are no 1s in column h between rows g and i.

When (g, h) is examined, (g, j) must already be known to be 1, since no position of row g will fill in a 1 in position (g, j), and (g, j) is set to be 1 by the algorithm ((i, j) is the first entry incorrectly left 0). Thus, when (g, h) is examined, (g, j) will be found as the next 1 in row g. No entry examined after (g, h) will fill in a 1 in position (i, h) since there is no 1 in column h to be the upper right corner of any Γ filled in. Therefore, when (g, h) is examined, (i, h) is already known to be a 1, and will be the next 1 in column h when (g, h) is examined.

When (g, h) is examined, (g, j) will be the next known 1 entry in row j, and (i, h) will be the next 1 entry in column h. The algorithm will fill in (i, j) with a 1, contradicting the assumption that there is a first 1 entry of the matrix which is not filled in correctly by the algorithm. QED

$$\begin{array}{ccc} 1 & 1 & 0 \\ 1 & * & 1 \\ 0 & 1 & * \end{array}$$

Figure 10: Example Matrix: Redundant 1s Marked with *

The algorithm for filling in the matrix correctly if given the position of all nonredundant 1s is given in the proof above. As each 1 value, which corresponds to an edge is visited, we need to be able to find the next 1 in the corresponding row and column, and possibly insert a new 1 value in the matrix. If the 1 values for each row and column are kept in a balanced tree data structure, each edge can be processed in O(log$n$) time. Using the fact that entries in the set are in the range $1..n$, this can be reduced to O(loglog$n$) time per edge, giving an O($m$loglog$n$) algorithm for reconstructing the matrix, if we are given only the nonredundant 1s in the matrix.

Theorem ? shows that we can store a chordal bipartite graph by giving the position of all nonredundant 1s. In order to fully analyze the construction and use of this representation, we need to know the time taken in finding the set of nonredundant ones from the original matrix M. It is easy to see that nonredundant ones can be found in O(n^4) time; examine each possible 4 tuple corresponding to a pair of rows and a pair of columns; if these induce a Γ mark the bottom right entry as being redundant. In fact, we can do much better.

The algorithm is essentially the reversal of the process of reconstructing M from the list of nonredundants 1s discussed in theorem ?; in fact, these are all derived from the algorithm in [116] for determining whether a matrix is Γ-free, which is discussed in a later section. Given a matrix M, we look at each 1 value in the matrix. If the position of the 1 value is (i, j), we look for the next column k which has a 1 for row i, and the next row l which has a 1 in column j, and mark the 1 value in position (l, k) as a redundant 1.

Every 1 marked as redundant by this algorithm must be redundant, since we find a Γ which has each such element at the bottom right. If (k, l) is redundant, consider the smallest Γ in the sense of theorem ? if the entry at position (k, l) is replaced by a 0. Let (i, j) be the position of the upper left corner of this Γ. If there is a 1 entry in row i, column $j2$, where columns $j < j2 < l$, then the entry at position $(k, j2)$ must be 1 (or rows (i, k) columns $(j, j2)$ induce a Γ), which implies that (i, k) and columns $(j2, l)$ induce a smaller Γ. A similar argument can be used to show that there is no 1 in column j between rows i and k. Therefore, (i, j) will mark (k, l) as a redundant 1.

We should note that that 1s may be redundant as a result of other redundant 1s. For example, in figure ?, the nonredundant 1s are at positions (1,1), (1,2), (2,1), (2,3), and (3,2). However, there is no Γ if we replace the 1 at position (3,3) by 0 unless the redundant 1 at position (2,2) is included.

Representing a chordal bipartite graph by storing the position of all nonredundant ones may be space efficient, but it is not satisfactory for a number of reasons. It is inefficient for testing adjacency, since we must reconstruct the entire matrix in order to test adjacency between a single pair of elements.

Open Problem: Is there a generalized implicit representation of chordal bipartite graphs?

We note that Dahlhaus [] has shown that it is possible to add O($nlog^3n$) bits to this representation scheme, and test for adjacency between vertices of G in O(log$n$) time for chordal bipartite graphs; the data structure can be computed in O($nlog^3n$) time from the nonredundant one representation. This is not a local representation; to compute adjacency between x and y, one must use bits which are globally accessible.

The problem below deals with the notion of partial order dimension, as discussed in chapter ?. There are a number of candidate forms of generalized implicit representations of chordal bipartite graphs. I choose to focus on dimension because dimension is a well studied parameter, and determining that the dimension of a chordal bipartite graphs grows very quickly, which would be discouraging if we are only looking for an implicit representation, would have independent significance.

Open Problem: What is the maximum dimension as a function of n of a partial order which corresponds to a chordal bipartite graph?

Another perspective on representing chordal bipartite graphs comes from trying to represent the class of graphs used in theorem ? efficiently, since this is a large class of chordal bipartite graphs. It is not at all obvious that this class has small dimension. However, there are easy ways to represent this class. For example, let the rectangle number of a graph be the minimum number of isooriented rectangles per vertex so that G is the intersection graph of these rectangles. In exercise ?, the reader is asked to show that the rectangle number of the graphs in theorem ? is O(log$n$).

Open Problem: What is the maximum rectangle number of a chordal bipartite graphs as a function of n?

10.3 Characterizations

Even if we have an implicit representation for chordal bipartite graphs, that representation may not be a characterization. For example, even if the maximum dimension of a chordal bipartite graph is O(log$n$), there would be bipartite graphs which are not chordal bipartite and have dimension O(log$n$). Similarly, the nonredundant one representation is not a characterization of chordal bipartite graphs. Ideally, we would like an implicit representation which characterizes the class exactly; no such representation is known at this time. However, there are two tree models which are exact characterizations of the chordal bipartite graphs. Both of these characterizations use $\Omega(n^2)$ space, but they are interesting nevertheless, and may help some reader design a space efficient characterization.

The first of the two characterizations is from [63]; it is actually a characterization of the related class of strongly chordal graphs. The relationship between strongly chordal graphs and chordal bipartite graphs is discussed in more detail in chapter ?; in particular, a bipartite graph G is chordal bipartite if and only if the graph formed

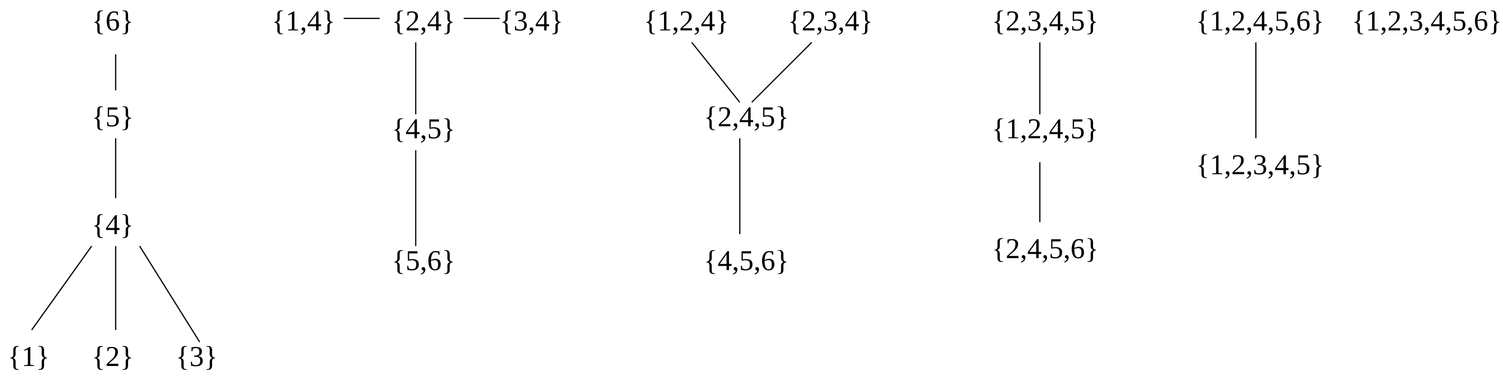


Figure 11: Tree Sequence in Characterization

by turning one color class into a clique is strongly chordal. This gives an intersection model for the class by restricting the intersection model of chordal graphs discussed in chapter ?.

Theorem [63]: G is strongly chordal if and only if G is the intersection graph of subtrees of a weighted, rooted tree such that for every pair of intersecting subtrees T_x, T_y in the model, vertices in the intersection of $T_x \cap T_y$ are exactly the closest vertices to the root in one of the trees T_x or T_y.

The second characterization comes from [111]. This is originally given as a characterization of totally balanced hypergraphs. Connections to graph classes come from the fact that the hypergraph defined by the adjacency matrix of G is totally balanced iff G is strongly chordal, and the hypergraph defined by the bipartite adjacency matrix of G is totally balanced iff G is chordal bipartite; see [23] for details.

It may be helpful to examine the sequence of trees in figure ? before looking at the definition. Given a hypergraph H, there is a sequence of trees T_0,T_1,...,$T_n - 1$. Each T_i is a tree on n - i nodes, and each node of T_i corresponds to a set of i+1 vertices of H. The sequence of trees obeys the following rules.

i) Nodes of T_0 correspond to vertices of H. For $i > 0$, each node of T_i comes from a distinct edge e of $T_i - 1$, and corresponds to the union of the vertices from either endpoint of e.

ii) Each edge of a tree connects two nodes x,y such that the set corresponding to x can be obtained from the set corresponding to y by replacement of exactly 1 vertex.

iii) Every hyperedge e of H corresponds to some node of one of the trees.

Lehel [111] shows that a hypergraph H is totally balanced if and only if such a sequence of trees exist. I find this to be an elegant characterization, but do not see how to make it space efficient. If we provide the entire sequence of trees, we have used $\Theta(n^2\log n)$ space. Perhaps some reader will find a way to show how to get from one tree to the next using less space. I am not optimistic about this leading to a generalized implicit representation, however; even if the sequence of trees can be encoded efficiently, testing adjacency between vertices seems to depend on much more than the index of nodes in the sequence of trees.

10.4 Recognition

There is a polynomial time recognition algorithm for chordal bipartite graphs as a consequence of their definition, since we can determine whether an arbitrary graph has an induced cycle of size at least k in O(n^k) time. However, we can do much better; the best known time bounds for this problem are O(mlogn) or O(n^2).

Just as we saw in chapter ? for chordal graphs, algorithms for chordal bipartite recognition are divided into two phases. A construction phase produces an ordering of the bipartite adjacency matrix, which may or may not be Γ-free, and a verification phase which checks whether the ordering is in fact Γ-free. In the first algorithm designed specifically for recognizing chordal bipartite graphs, both phases took O(n^3) time. Lubiw [116] showed that the verification phase could be solved easily in O(n+m) time as follows. The algorithm is very similar to the algorithm for removing nonredundant 1s from a representation described in section ?.

For each 1 entry (i, j) in the matrix, let k be the next row such that the entry at position (k, j) is 1, and let l be the next column such that the entry at position (i, l) is 1. We check that the entry at position (k, l) is a 1. If all these values are 1, then the matrix is Γ-free; the proof is similar to the proof above, showing that the smallest Γ in the matrix will be found in this way. If we are given the matrix in the form of an ordered adjacency list, the verification phase can be performed in O(n+m) time; this is easy given the result of exercise ? of chapter ?, which allows constant time adjacency testing with input in adjacency list form, if one is willing to use $\Theta(n^2)$ space.

We will describe an O(n^2) algorithm for taking a 0/1 matrix and producing a Γ-free matrix by permuting rows and columns, if such a permutation is possible. The technique used is called doubly lexical ordering. A doubly lexical ordering of an i by j matrix is an ordering of the rows and columns such that if the columns and rows are viewed as i or j digit numbers, both rows and columns occur in increasing order. Unfortunately, for historical reasons, we must read these numbers from right to left and bottom to top; that is, the most significant digit for rows is the rightmost column, and the most significant digit for columns occurs in the bottom row. An example of a matrix and a doubly lexical ordering of the matrix is shown in figure ?.

? showed that a matrix can be permuted to be Γ-free if and only if every doubly lexical ordering is Γ-free. This led to the development of a series of doubly lexical ordering algorithms. The first, due to ?, has time complexity O($i^2 j$), where i is the number of rows and j is the number of columns. This was improved to O($mlog^2n$) by Lubiw [116] where m is the number of nonzero entries and n is the total number of rows and columns. Paige and Tarjan improved the time bound to O($m$log$n$). All these algorithms are designed to create a doubly lexical ordering of an arbitrary

	1	2	3	4	5
1	1	0	1	1	0
2	0	1	1	0	0
3	1	1	1	0	0
4	0	1	0	1	1
5	1	0	0	1	0

	3	1	5	2	4
2	1	0	0	1	0
3	1	1	0	1	0
5	0	1	0	0	1
1	1	1	0	0	1
4	0	0	1	1	1

Figure 12: Doubly Lexical Ordering

matrix M, which is not constrained to be 0/1. We will describe an algorithm which works only for the case of 0/1 matrices, with time complexity $O(n^2)$. This algorithm is described in [153], and can be viewed as a particular implementation of Paige and Tarjan's algorithm for the 0/1 case. We will need the $O(n^2)$ time bound for a number of results in chapter ?, where chordal bipartite graphs are used to recognize circular-arc graphs and trapezoid graphs.

At a high level, algorithms for chordal bipartite recognition resemble algorithms for chordal graph recognition. Both algorithms divide into a phase for construction of an ordering, and a phase for verifying the ordering. The construction phases use partitioning techniques. To construct a perfect elimination ordering for a chordal graph, we choose any vertex x to go last in the ordering, divide all sets into neighbors and nonneighbors of x, placing neighbors after nonneighbors; any vertex from the last set is chosen as we repeat this procedure. The algorithm for doubly lexical ordering is similar, but it is no longer true that any vertex can be chosen to go last. It is easy to see that if $N(x)$ is a subset of $N(y)$, x must come before y in a doubly lexical ordering.

Hoffman, Kohlen, and Sakarovich dealt with this problem by choosing a vertex which could go last at each step. Their algorithm successively chooses rows to go last in the row set and partition the column set. For each row r, they keep a tuple corresponding to the number of vertices in each set of the partition which r is adjacent to, and the row with largest lexicographic tuple value (with numbers read from right to left) is chosen to go last and partition the columns.

Let us demonstrate the algorithm on the matrix of figure ?. Originally, the columns are in a single block (1,2,3,4,5); row values are 1:3 2:2 3:3 4:3 5:2. Row 1 may be chosen to go last; this divides the columns into neighbors and nonneighbors of row 1 (2,5)(1,3,4). The new values for the remaining rows are 2:1,1 3:1,2 4:2,1 5:0,2. Row 3 has the 'largest' value, and is chosen next; the new column partition is (5)(2)(4)(13). Row values become 2:0,1,0,1 4:1,1,1,0 5:0,0,1,1; thus row 5 is chosen next. The column partition becomes (5)(2)(4)(3)(1). Remaining row values are 2:0,1,0,1,0 4:1,1,1,0,0. Thus the resulting matrix is shown in figure ?

The reader may note a similarity to the description of lexicographic breadth first search in [79]. The drawback of this method is that we now have to use time in

	1	2	3	4	5
1	1	0	1	1	0
2	0	1	1	0	0
3	1	1	1	0	0
4	0	1	0	1	1
5	1	0	0	1	0

	5	2	4	3	1
4	1	1	1	0	0
2	0	1	0	1	0
5	0	0	1	0	1
3	0	1	0	1	1
1	0	0	1	1	1

Figure 13: Sample Matrix for HKS Algorithm

choosing a vertex which goes last; this increases the time complexity from linear for chordal graphs to n^3 for chordal bipartite graphs.

The algorithms of [],[], and [] are based on the same idea, but avoid the bottleneck step of searching for a vertex which can go last; we choose to describe the algorithm from [153]. We keep a current partition of both the rows and the columns. Any row r from the last set of rows is used to partition the last set of columns. However, r cannot be placed at the end of the row list; other rows may have edges to all of r's edges and more. If we call the neighbors of r from the last block of the column partition C_r, we will end up by placing r after all rows from the last block which have any nonneighbors in C_r, before any rows from the last block whose neighbors in the last block properly contain C_r, and together with any rows from the last block whose neighbors in the last block equal C_r.

Let R be the last current block of the row partition, and let C be the last block of the column partition which is not marked as having been split by R. If R has split every block, we remove R from further consideration, placing rows in R in any order on the output list. Rows of R are chosen for processing in any order. Let r be the current row. We step from back to front through subblocks of C which have not previously split by r until we reach a block B which contains a nonneighbor of r; B is split into N(r) and B - N(r), with N(r) placed after r in the partition list. Store the fact that r has split all blocks which do not precede B. After all rows of r have been processed, divide R on the basis of how many consecutive rows at the end of C are adjacent to the row; rows adjacent to more columns are placed later in the ordering. Each vertex placed in the same subblock of R' has its last nonneighbor from C in the same subblock C' of C; mark R' as having split all blocks which do not precede C'. If the vertices of R' are adjacent to all vertices of C, mark R' as having split all blocks which do not precede C.

Let us examine the algorithm with the input matrix of figure ?. Initially, the row partition is (1,2,3,4,5) and the column partition is (1,2,3,4,5). We might choose row 5 first, and obtain the column partition (2,3,5)(1,4). Row 4 might be chosen next, giving the column partition (2,3,5)(1)(4); note that row 4 does not reach the first block of the column partition, and therefore does not split this block. Rows 2 and 3 are nonadjacent to column 4, and do no further splitting. Row 1 is adjacent to both 1 and 4, and thus splits the block (2,3,5), leaving a column partition (2,5)(3)(1)(4). The new row partition becomes (2,3)(4)(5)(1). Row 1 has split all blocks, and is done; the same is true for row 5. Row 4 now splits earlier blocks; since it is adjacent to both 2 and 5, it does no further splitting. Rows 2 and 3 now compare themselves to the next block (1) of the column partition; the new row partition becomes (2)(3)(4)(5)(1). Row 3 splits earlier blocks of the column partition, and the final column partition becomes (5)(2)(3)(1)(4); the row and column orderings define a doubly lexical ordering of the matrix.

For any pair of vertices r, c, the block containing c is split by r only once, so the total time spent splitting blocks is $\mathrm{O}(n^2)$. Sometimes, r is contained in a block R which is used to refine a block C, even though r has already split C. When this happens, we must find the last subblock of C which contains nonneighbors of r, in order to place r in the correct row block R'. This does not cause any problems with respect to the time bound, since any block examined by r at this time will be marked as split by R', so vertices in these blocks will never be compared to r later in the algorithm.

It is interesting to note that none of the algorithms for recognizing chordal bipartite graphs use the fact that G must have a Γ-free ordering during the construction phase of the algorithm. We would be happy to have a linear time algorithm which performed doubly lexical ordering if G was chordal bipartite, and created any ordering if G was not chordal bipartite. The analogy here is with the comparability graph recognition algorithm, which produces an ordering in linear time which is transitive if and only if G is a comparability graph. I feel that there should be a linear time algorithm for chordal bipartite graph recognition based on this approach; note that the verification phase is linear for the class.

We now argue that if G is chordal bipartite, every doubly lexical ordering of the bipartite adjacency matrix is Γ-free. Let vertices 1,2 and a, b form a Γ in the doubly lexical ordering, with $1 < 2$ and $a < b$. Since row b comes after row a but 2 is adjacent to a and not b, there must be a vertex 3, $3 > 2$, such that 3 is adjacent to b but not a; similarly, some vertex $c > b$ must be adjacent to 2 but not to 1. We choose vertices 3 and c which are maximum in the ordering meeting these requirements; vertices 3 and c must be nonadjacent or the vertices induce a C_6.

This forms the basis of an inductive proof, in which we have a P_k with the last row and column being inconsistent with a doubly lexical ordering on the induced vertices, necessitating a new later row and column. The completion of the proof that every doubly lexical ordering of a chordal bipartite graph is Γ-free is left to exercise ?.

10.5 Optimization Problems on Chordal Bipartite Graphs

A number of problems can be solved efficiently on chordal bipartite graphs due to roperties of Γ-free matrices. These will have some consequences for other classes as well, since various problems on such classes as circular-arc graphs [59] and weakly chordal comparability graphs (see the intersection chapter) can be transformed to problems on chordal bipartite graphs.

The neighborhood containment problem for chordal bipartite graphs will be especially important for speeding up a number of algorithms. In the neighborhood containment problem, we want to determine for all pairs of vertices x, y whether $\mathrm{N}(x)$ is a subset of $\mathrm{N}(y)$. In general, the best known neighborhood containment algorithm uses matrix multiplication. Since $A^2[x,y]$ is equal to the number of paths of length 2 between x and y, $\mathrm{N}(x)$ is a subset of $\mathrm{N}(y)$ if and only if $A^2[x,y]$ is equal to the degree of x. This algorithm takes $\mathrm{O}(n^{2.376})$ time if very complex algorithms are used.

If we are given a doubly lexical ordering of the bipartite adjacency matrix, it is very simple to determine neighborhood containment for a chordal bipartite graph. Suppose that x comes before y in the ordering. Let b be the first neighbor of x in the ordering of the other color class. If y is nonadjacent to b, then $\mathrm{N}(x)$ cannot be a subset of $\mathrm{N}(y)$. If y is adjacent to b, then $\mathrm{N}(x)$ is a subset of $\mathrm{N}(y)$, since if there was any later vertex c adjacent to x but not y, then x,y,b,c induce a Γ. If x is after y in the doubly lexical ordering, $\mathrm{N}(x)$ is a subset of $\mathrm{N}(y)$ if and only if $\mathrm{N}(x) = \mathrm{N}(y)$. We can determine all 'twin' vertices in linear time for an arbitrary graph (exercise ? of chapter ?); alternatively, it is easy to compare successive rows to identify twins from the doubly lexically ordered matrix. Therefore, neighborhood containment in chordal bipartite graphs can be found in $\mathrm{O}(n^2)$ time, using graph theoretic algorithms.

Let x be the first vertex from one of the color classes in a Γ-free ordering. It is easy to show that the neighborhoods of neighbors of x are totally ordered by the subset relation. This gives a nice characterization of chordal bipartite graphs which is analagous to the perfect elimination scheme characterization of chordal graphs. A bipartite graph is chordal bipartite if and only if we can repeatedly remove a vertex which is not in the center of a P_5; note that such vertices can clearly not be part of a chordless cycle of length greater than 5, just as simplicial vertices cannot be part of a chordless cycle of length greater than 3.

The property above immediately gives efficient algorithms for some problems. Consider the matching problem on chordal bipartite graphs. The simple greedy algorithm of repeatedly matching the first remaining row in the ordering with the first adjacent column will always produce a maximum cardinality matching for a chordal bipartite graph. In chapter ?, we study the related class of strongly chordal graphs, in which the augmented adjacency matrix formed by replacing all diagonal elements by 1s has a Γ-free ordering. Domination is easy on this class, since the neighborhoods of neighbors of the first row are ordered by set containment; this element should clearly be dominated by its last neighbor in the ordering.

10.6 Generalizing Chordal Bipartite Graphs

The following generalization of chordal bipartite graphs seems natural, but has not been studied previously. For any fixed value k, we can define a class of bipartite graphs which allows no induced cycles of length greater than k. Counting, representation, recognition (beating n^k or $O(n^{k-3}MM)$), and optimization problems all seem to be open for this class, for all classes beyond chordal bipartite.

From the large set of problems defined in this way, I would like to choose one which seems particularly interesting.

Open Problem: Consider the classes of bipartite graphs defined by forbidding induced cycles of length greater than k. Is there a fixed i such that the number of graphs in the class becomes $2^{\Theta(n^2)}$ when k is greater than i, or is the number of graphs in the class $2^{o(n^2)}$ for all fixed k?

NOTE: see Brightwell et al, Forbidden Induced Partial Orders, seems to say $2^{o(n^2)}$ for all fixed

I do not even know whether or not the number of bipartite graphs without induced cycles of length greater than six is $2^{\Theta(n^2)}$.

In chapter ?, we discuss a different way to generalize chordal bipartite graphs, by looking at other forbidden submatrices. Bipartite graphs without induced paths of length greater than k, rather than induced cycles of length $> k$, are discussed in [].

(results, number of these? induced or not?)

10.7 Exercises

0) make sure O(c) adj test for adj lists is in previous exercise/chap

1) Show that every 1 added by nonredundant 1 algorithm is necessary; i.e. every Γ-free matrix containing the initial set of 1s must contain every 1 added by the algorithm.

2) Prove that every matrix has a doubly lexical ordering.

3) Show that chordal bipartite graphs are exactly those graphs which are bipartite, and have an elimination scheme based on always removing vertices which are not in the middle of a P_5.

4) Let x be a row corresponding to the first vertex in a Γ-free ordering of the bipartite adjacency matrix. Show that the neighborhoods of vertices in $N(x)$ are ordered by set containment.

5) A bisimplicial edge (x, y) in a bipartite graph is an edge such that every neighbor of x is adjacent to every neighbor of y. A bisimplicial edge elimination scheme successively removes bisimplicial edges (not the endpoints, as in the class called perfect elimination bipartite graphs discussed in the next question, , but just the edge itself) until no edges remain. Show that G is chordal bipartite if and only if G has a bisimplicial edge elimination scheme.

6) A bipartite graph is a perfect elimination bipartite graph if we can successively remove both endpoints of a bisimplicial edge until no edges remain. Show that there are $2^{\Theta(n^2)}$ perfect elimination bipartite graphs.

7) A bipartite graph is convex if we can order one color class so that the set of neighbors of every vertex in the other color class occurs consecutively in the ordering. Show that every convex graph is chordal bipartite. Show that the dimension of a convex graph (considered as a height 1 poset) is at most 3.

8) Show that the rectangle number of graphs in theorem ? is O(log$n$).

9) Complete the proof that every doubly lexical ordering of a chordal bipartite graph is Γ-free.

10a) Show that computing neighborhood containments in a chordal graph is as hard as for general graphs.

10b) Show that we can compute neighborhood containments for all pairs of neighbors in a chordal bipartite graph in $O(n^2)$ time.

11) If we subdivide every edge of a chordal graph, we must get a bipartite graph with no induced cycles of length > 6. Since there are 2^{cn^2} chordal graphs, why doesn't this show that there are 2^{cn^2} bipartite graphs with no induced cycles of length > 6?

11 Matrices

This chapter is the result of a different way of generalizing questions on chordal bipartite graphs, by generalizing the matrix characterization rather than generalizing the graph theoretic definition. The concept of implicit representation is extended from graphs to matrices, and we raise open questions regarding counting and representing matrices with various forbidden submatrices. It is possible to phrase these questions in terms of bipartite graphs, but to do so would be artificial, while the questions on matrices seem quite natural. We restrict our attention for the purposes of this book to 0/1 matrices, which are the most closely related to graphs. Matrices will be assumed to be n by n unless stated otherwise. Many of the classes of matrices discussed in this chapter are taken from [105].

Defining an implicit, generalized implicit, or space optimal representation of a class of matrices is analogous to the graph theoretic definition. A class of matrices has an implicit representation if we can store O(log$n$) bits of information with each row and column, such that the entry in row x, column y can be deduced entirely from the bits stored at x and y. A class of matrices with $2^{f(n)}$ members has a generalized implicit representation if we can store O(f(n)/n) bits of information at each row and column so that the entry at row x, column y can be deduced from the bits stored at x and y, while a space optimal representation uses O(f(n)) bits.

A forbidden submatrix F for a class of matrices is a matrix which is not an induced submatrix (i.e. the result of deleting rows and/or columns) of any matrix in the class. A matrix M is in the class PM(F) if we can permute the rows and columns of M so that the result does not contain F as an induced submatrix.

Thus, PM(Γ) corresponds to bipartite adjacency matrices of chordal bipartite graphs. This class has $2^{\Theta(nlog^2n)}$ members, and we know of a way to construct space efficient representations efficiently, while no implicit representation is known for the class. Similar questions can be asked about other classes defined by forbidden submatrices; we will discuss some of the most obvious specific matrices to consider, and describe what is known about counting the size of the classes in the more general case. In contrast with forbidding induced subgraphs, which is discussed in chapter ?, we will find that forbidding any single induced submatrix reduces the number of matrices in the class to $2^{o(n^2)}$, and thus each forbidden matrix has the potential to have its own interesting form of generalized implicit representation.

11.1 Small Forbidden Classes of Matrices

For a more complete description of what is known about recognizing classes defined by forbidding small induced submatrices, the reader is invited to see [105].

Of course, forbidding 1 by 1 matrices is uninteresting, leaving only 1 matrix (all 0s or all 1s) for each value of n. Therefore, the first forbidden matrices we consider are the 2 by 1 forbidden submatrices; because of the obvious symmetries we only need to consider the classes of matrices PM(1 1) and PM(1 0). PM(1 1) is a trivial class of matrices, exactly equal to the class of matrices which have at most one 1 entry in each row.

PM(1 0) is not a particularly interesting class of matrices. Clearly the ordering of rows does not matter, and for any row we must have all 0s before all 1s. There is clearly an implicit representation of such matrices; store with each row the number of 0s in the row, and store with each column the position of the column in the permuted ordering. Recognizing matrices in the class is easy, and will be left for the exercises.

We then come to two by two forbidden submatrices. We have already dealt with one such class, since chordal bipartite graphs correspond to forbidding Γ in the permuted matrix. Symmetries would map any other forbidden two by two matrix with exactly one 0 to chordal bipartite graphs, and exactly three 0s in the matrix could be handled in representations by interchanging the roles of 0 and 1.

11.2 Linear Matrices

Forbidding all 1 and all 0 matrices are equivalent for our purposes; assume that our forbidden submatrix is a 2 by 2 matrix with every entry equal to 1. These matrices are called linear matrices by Berge [16]. Since row and column permutations do not affect the forbidden matrix, we can interpret this completely as a graph theoretic problem on bipartite adjacency matrices; these matrices correspond to bipartite graphs without cycles of length four. For this class, we can come close to counting the number of matrices, and finding an implicit representation.

The key idea for counting this class of matrices comes from the theory of projective planes. It is possible to construct for certain n an n by n matrix M such that each row has $\Omega(n^{.5})$ 1 entries, and no pair of rows has a 1 in the same pair of columns. M cannot contain $1_{2,2}$, and replacing any 1 entries by 0s will not create an induced $1_{2,2}$. Therefore, the number of matrices in the class is at least $2^{n^{1.5}}$, since we can include or not include each one value from M in the matrix.

We can also show that every linear matrix has $\mathrm{O}(n^{1.5})$ 1 entries. The total number of 1s in rows with at most $2n^{.5}$ 1 entries is $\mathrm{O}(n^{1.5})$, so these can be ignored. We will show that there are fewer than $n^{.5}$ rows with more than $2n^{.5}$ 1 entries, which shows that the total number of 1 entries is $\mathrm{O}(n^{1.5})$. Consider a set of $n^{.5}$ rows, each of which has more than $2n^{.5}$ 1 entries. For any pair of rows, there is at most 1 column which has a 1 in both rows, or the submatrix induced by two columns and this pair of rows is the forbidden submatrix. Thus, each row has a 1 in at least $n^{.5}+1$ columns for which no other row in this set has a 1. There are only n columns altogether, so this cannot occur, which implies that the number of 1 entries is $\mathrm{O}(n^{1.5})$.

The fact that only $\mathrm{O}(n^{.5})$ rows have more than $\mathrm{O}(n^{.5})$ 1 entries also allows us to store the matrix locally. Since row and column permutations do not affect this class, let us assume that the rows are ordered in decreasing order of number of 1 entries. For all but the first $n^{.5}$ rows, store the positions of the row which contain a 1. For each column, store the first $n^{.5}$ bits of the matrix. This uses $\mathrm{O}(n^{.5}\log n)$ bits per row, and $O(n^{.5})$ bits per column. This leaves the question open as to whether there are $2^{n^{.5} logn}$ matrices of this form, or whether a representation using $O(n^{.5})$ bits per row and column is possible; this gap is relatively small when compared to some other classes defined by forbidden submatrices.

We now consider the recognition problem for this class of matrices. Let m be the number of nonzero entries in the matrix; we may assume that m is $O(n^{1.5})$. Since the forbidden matrix is impervious to row and column transformations, we get an immediate $O(n^4)$ algorithm by checking each pair of rows and each pair of columns. This can easily be improved using a number of different strategies. It is easy to determine whether a pair of rows contains a forbidden induced submatrix in $O(n)$ time, reducing the time complexity to $O(n^3)$. This can be reduced to $O(nm)$ by noting that we can test whether there is any forbidden submatrix including row r in $O(m)$ time. Matrix multiplication can be used to get an algorithm with time complexity $O(n^{2.376})$ (exercise ?). We can bring this down to $O(n^2)$ by keeping an n by n array which has a 1 at position i,j if and only if we have seen a 1 in the same column for rows i and j; when a new column is considered, every unit of work can be charged to a 0 entry of the array, since any attempt to mark a 1 entry of the array causes us to answer that the forbidden matrix is contained in M. We will refine this method below to give an $O(mn^{.5})$ algorithm for recognizing this class of matrices; an $O(m^{1.5})$ algorithm is presented in [].

(Kratsch et. al.!)

We will show how to recognize these matrices in $O(mn^{.5})$ time. We know that there are at most $n^{.5}$ row which contain more than $2n^{.5}$ 1 entries. For each such row r, we can use $O(m)$ time to check whether there is a violation involving r and some other row. After checking for forbidding submatrices in these rows, we may assume that every remaining row has at most $2n^{.5}$ 1 entries.

We can now use the array based method on the remaining rows to solve the recognition problem, but we get a better time bound. When a new row is considered, we do work proportional to the square of the number of 1 entries in the row, and never look at entries in the row again. Since each remaining row has $O(n^{.5})$ 1 entries, we mark $O(n^{.5})$ array positions for each 1 entry in the matrix; thus the total work is $O(mn^{.5})$. This method uses $\Theta(n^2)$ space; exercise ? asks to solve the same problem in $O(mn^{.5})$ space.

We note that the problem of recognizing these matrices is also equivalent in time complexity to the problem of determining whether an arbitrary graph contains a C_4 as a subgraph, where we do not require the subgraph to be an induced subgraph. Clearly, an algorithm for finding C_4s in an arbitrary graph can be used to find C_4s in a bipartite graph, which is the same as finding the forbidden submatrix if M is interpreted as a bipartite adjacency matrix. For any graph G, we can create a bipartite graph $B(G)$ by creating two vertices v_1, v_2 for each vertex v in G, and adding an edge (v_1,w_2) to $B(G)$ if and only if v and w are adjacent in G. Exercise ? shows that G contains a C_4 if and only if $B(G)$ contains a C_4, so the problem of finding C_4s in an arbitrary graph can be transformed to a bipartite problem using linear time and space. An algorithm which takes $O(m\ a(G))$ for finding C_4s, where $a(G)$ is the arboricity of G (as defined in the exercises), is presented in []. The arboricity of a C_4-free graph cannot be more than $n^{.5}$ (exercise ?), so the algorithm in [] can be modified to be always as good as the algorithm above if we include a simple stopping condition.

11.3 Forbidden 2 by 2 Identity Matrices

One class of forbidden 2 by 2 matrices is particularly interesting, and not very well understood. This is the class of 0/1 matrices such that rows and columns can be permuted so that there is no induced 2 by 2 identity matrix. There are a number of interesting questions with respect to the class.

$\mathrm{PM}(I_2)$ has been studied under the name of connected matrices. This terminology comes from the following definition of matrix connectivity [14]. For any pair of rows i, j, consider the sequence which comes from subtracting the entry at row j from the entry at row i, as we read the rows from left to right; the number of times this sequence changes signs (here 0s count as both negative and positive) is called $\mathrm{signchange}(i, j)$. A matrix is k-connected if the maximum value of $\mathrm{signchange}(i, j)$ is at most k; a 1-connected matrix is called a connected matrix.

It is not hard to show, as given in [105, 14] that forbidding I_2 is the same as forbidding the pair of matrices

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

Clearly, if I_2 is forbidden then neither of these matrices can occur, so every matrix in $\mathrm{PM}(I_2)$ is ?-connected. On the other hand, I_2 is a valid ordering for this class, so $\mathrm{PM}(I_2)$ might be a proper subset of connected matrices.

Suppose that we can permute the matrix so that neither of the above submatrices are induced submatrices; let M be the permuted matrix. Changing the order of rows, while leaving columns in the same order, cannot create either of the forbidden submatrices. We will choose an ordering of the rows so that the result matrix is in $\mathrm{PM}(I_2)$.

Consider each row as a binary number, read from right to left. Order the rows in increasing numerical order. Suppose that the result matrix contains an I_2 involving rows a and b, and columns y and z. There must be a column after z which has a 1 in row a and a 0 in row b, or row a would be larger than row b. This creates the forbidden matrix

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

This implies that M contains one of the forbidden submatrices. Thus, if M is connected, we can permute the rows of M to be in $\mathrm{PM}(I_2)$, and the connected matrices are exactly equal to $\mathrm{PM}(I_2)$.

One of the major advantages of the characterization in terms of forbidding the two matrices is that row permutation does not affect these forbidden submatrices; this makes it easier to rule out possible arrangements, as is required in problem ???.

The major papers dealing with this class of matrices appear only in Russian. The paper [14] claims to give a recognition algorithm for the class which works in $O(i^2j^2)$ time, where i and j are the number of rows and columns in M. Although I was able to get this paper translated, there are some crucial steps of the algorithm which I fail to understand; the algorithm as I understand it seems to be clearly incorrect in some parts in a manner that is inconsistent with the generally careful proofs in the rest of the paper. Thus, it is not clear whether the error is in the original paper, the translation, or my understanding. It would be a service for some reader who is capable of working in both English and Russian to present an English version of this paper.

Thus, the counting, representation, and perhaps recognition problems remain open on this class of matrices.

(I have a $2^{n^{1.5}}$ + epsilon proof for these?)

(N-containment on these? One direc easy. Is N-cont 2d poset?)

11.4 Forbidding $\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$

There is one class of forbidden 2 by 2 matrices which we have not covered, defined by forbidding a matrix with 1s in column 1, and 0s in column 2; we will call this matrix F in this section. Very little work has been done on this class of matrices.

[105] points out that matrices in this class can be recognized in polynomial time. The matrices are affected by column transformations, but not row transformations. We make a graph G with columns as vertices, which reflects required orderings of the columns. For each ordered pair of columns c_1,c_2, we can check whether these contain the forbidden matrix if c_1 is placed before c_2; if they do, add an edge from c_2 to c_1 in G. M can be permuted to avoid F if and only if G is acyclic, and any topological sort of an acyclic G gives a valid ordering of the columns. A straightforward implementation of this algorithm takes $\Theta(n^3)$ time.

We will present a linear time algorithm for recognizing this class of matrices. First, let us note that it is trivial to find a correct ordering of the columns, if such an ordering exists. If there is a valid column ordering, then ordering columns in increasing order of the number of 1s in the column is a valid ordering. Exercise ? asks to show that if the 1s in column c_1 are a proper subset of the 1s in column c_2, any ordering with c_1 after c_2 is still valid if c_1 and c_2 are swapped. Consider any columns c_1, c_2 such that the number of 1s in c_1 is smaller than the number of 1s in c_2. If there is a row r such that r has a 1 in column c_1 and a 0 in column c_2, then there must be two rows which have a 1 in column c_2 and a 0 in column c_1, and c_1 must precede c_2 in any ordering. Thus, we may assume that columns are ordered by

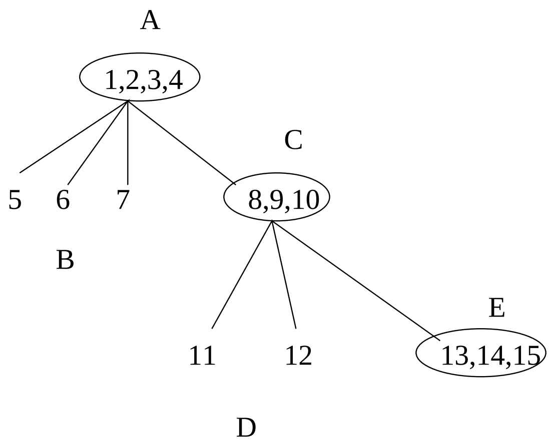


Figure 14: Set Labels for Tree

increasing number of 1 entries. If 2 columns have the same number of 1 entries, they either differ on at most two rows, in which case any ordering of the columns is valid, or they differ on at least three rows, in which case no ordering is valid. Therefore, order of columns which have the same number of 1 values does not affect whether the matrix is $\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$ free.

Our recognition algorithm reads one column at a time, and maintains a very restricted form of tree. Each node of the tree is a set of rows, and each row occurs in at most one node of the tree. The tree has height at most 2; at each level, at most one node is not a leaf corresponding to a single row of the matrix. The tree is made to have the property that two rows have a 1 value in a common column if and only if they are on a single root to leaf path in the tree. When a new column c is read, the ordering of columns seen up to this point is still valid if c has at most one 0 entry on every path from the root of the tree to a leaf. We first show how to check this condition on a tree, and then show how to update the tree based on the values for column c.

For convenience in the description of the algorithm, let us give names to some of the node types. Call the set of rows contained in the root A, and the set of leaf children of A which are single rows of the matrix B. The single nonleaf consisting of more than one row of M child of A will be called C; the one row children of C are called D, and lone child of C with more than one row is called E. A picture of this general tree is given in figure ?; we note that some of these sets may be empty.

Given the restricted form of the tree, it is easy to check whether the tree has at most one 0 entry on every root to leaf path. If A contains a row with a 0 in column c, verify that all other rows in the tree have a 1 in column c. If A contains no such row, check whether there is a row in C which has a 0 in column c. If there is a 0 row in C, check that all other rows in C, D, and E have a 1 in column c. If there is no 0 row in A or C, verify that E has at most 1 row with a 0 in column c. These steps clearly take time proportional to the number of 1 entries in c.

The key step is modifying the tree to reflect changes made by the addition of a column. Call F the set of rows which have a 1 in column c, and no 1 in any columns seen previously.

If c has a 0 in row a of A, create a height 1 tree with root containing all rows which have had 1 entries except a and F, and make the two children of the root a and F. Otherwise, let B_1 be the rows of B which have a 0 in column c, and let B_2 be the rows of B which have a 1 in column c. If c has a 0 in column x of C, leave A as the root, leave each row of B_1 as a child of A, make the rows of B_2,C-x,D,E a child C' of A, and make c and F the two children of C'. If c has no 0s in A or C, let D_1 (D_2) be the rows of d which have a 0 (1) in column c, and let e be the row of E which has a 0 in column c. Leave A as the root, and make each row of B_1 a child of A, and leave C as the other child of A. Make each row of D_1 and e children of C, and make E-e,D_2,B_2,F a single child of C.

All tree operations take time proportional to the number of 1 entries in c, so these matrices can be recognized in linear time.

The algorithm above also can be used to solve the counting problem for this class of matrices, though it does not solve the representation problem. We want to describe the sequence of tree operations using O($n$log$n$) bits. At first, this may seem difficult to do, since it is not possible to describe the changes in the tree caused by a single column c in O(log$n$) bits. First, the set of new rows F for a single column can be arbitrarily large. This is not a major problem, since each row is new only once; thus, the total number of bits needed to describe additions of new rows is O($n$log$n$).

Our other apparent difficulty is to deal with changes to B and D, since any subset of rows from B and D may be removed from its location, and placed in C or E. However, at most 2 rows are added to B and to D in a single step; this implies that the total number of removals from B and D is O(n).

This allows us to construct G from the following information. For each column c, store all row numbers which have their first 1 in column c, any row from A, C, or E which has a 0 in c. If there are no 0s in any row in A, store all rows of B which have a 1 in column c. If there are no 0s in any row of A or C, store all rows of D which have a 1 in column c. Each successive tree can be built from the previous tree and the information stored with c, so all trees can be constructed, and we can reconstruct column c from the information stored at c and the previous tree. Thus, M can be reconstructed from the information stored at each column.

Each row is stored as a 'new' row for one column, so the total space used storing new rows is O($n$log$n$). At most 1 row value is stored with c from each of A, C, and E, so the bits used for storing rows from these sets is O($n$log$n$). Each row of B or D stored with c is removed from B or D during this step; as argued above, this shows that the total number of bits stored for rows of B and D is O($n$log$n$).

Thus, we have showed that there are at most $2^{O(nlogn)}$ matrices which do not contain this forbidden matrix. The question of finding an implicit representation is left open at this time. I believe that an implicit representation should not be too difficult to find for this class; perhaps finding one could help us understand how to produce good representations of classes of matrices defined by other forbidden submatrices. A simpler proof of the upper bound on the number of matrices which does not immediately give a recognition algorithm is given in problem ?; we present the version in this chapter since it goes together with the recognition algorithm.

11.5 Other Classes of Interest

We mention in this section a number of other classes of matrices which have appeared in the literature. The counting and representation problem does not seem to have been studied for many of these classes.

We mention above that forbidding I_2 is equivalent to forbidding the two matrices

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

The latter characterization is part of a more general definition; a matrix has been called k-connected if it avoids the pair of k+2 by 2 matrices

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & .. \\ 0 & 1 & 0 & 1 & 0 & .. \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & .. \\ 1 & 0 & 1 & 0 & 1 & .. \end{pmatrix}$$

There are polynomial time algorithms for solving a number of NP- complete problems on 1 and 2-connected matrices (note: given rep or not?). Recognition of k-connected matrices is open for all $k > 1$, as are the counting and representation problems.

(binary Monge and bottleneck Monge small!)

A class of graphs called X-graphs is studied in [] (look up this paper). These are graphs in which there is an ordering of the vertices (Γ and I2-free : X graphs)

11.6 The Problems of Counting and Representation

As we saw in chapter ?, forbidding individual graphs, with a few exceptions, gives graph classes which are too large to have an efficient representation. The situation for forbidden submatrices is quite different; any forbidden submatrix of fixed size will reduce the number of matrices in the class sufficiently that we can hope for space efficient representations. Thus, every forbidden submatrix gives an open question for the generalized implicit graph question, while the same is not true for forbidden subgraphs.

To see why forbidding a fixed submatrix automatically reduces the number of possible matrices substantially, we start by examining a single class. Let us look at the number of n by n matrices which can be permuted to avoid I_2.

Suppose that we look at a submatrix which contains more columns than rows, and every column is distinct. There must be some pair of columns c_1, c_2 such that the one entries in each column are not a subset of the one entries in the other column. This means that there must be some pair of rows r_1, r_2 such that one row has the values 1 and 0 on c_1 and c_2 respectively, while the other has the values 0 and 1 on c_1 and c_2.

We will use this fact to show that each set of $n^{1/4}$ rows in an I_2-free matrix can be stored using $\mathrm{O}(n\log n)$ bits, rather than the $n^{5/4}$ bits which are required in general. By storing each set of $n^{1/4}$ rows in this way, we use $\mathrm{O}(n^{7/4}\log n)$ bits to store an I_2-free matrix.

If there are fewer than $n^{3/4}$ distinct columns on these rows, then we can store all rows in $\mathrm{O}(n\log n)$ bits. All distinct column values are stored, and given an index; this clearly uses $\mathrm{O}(n)$ bits. Each column can be represented by the corresponding index value for that column; this uses $\mathrm{O}(\log n)$ bits for each column, and therefore $\mathrm{O}(n\log n)$ bits to store all columns on these rows.

If there are more then $n^{3/4}$ distinct column values, we will show that the matrix contains an induced I_2. Group the distinct columns into consecutive sets of $n^{1/4}$ columns. Within each set of columns, there must be some pair of rows and columns so that one row has the values 1 0 on these columns, while the other row has the values 0 1. Since there are $n^{1/2}$ sets of columns and only $n^{1/4}$ rows, some pair of column sets C_1, C_2 must use exactly the same pair of rows; assume that all columns in C_1 are to the left of all columns in C_2. Call these rows r_1 and r_2; assume without loss of generality that r_1 is above r_2. From C_1, choose the column c_1 which has a 1 in r_1 and a 0 in r_2, and from C_2 choose column c_2 which has a 0 in r_1 and a 1 in r_2. These induce an I_2, which implies that there are at most $n^{3/4}$ distinct columns with respect to these rows; thus, all I_2-free matrices can be stored using $\mathrm{O}(n^{7/4}\log n)$ bits.

Although this argument may seem to be very dependent on the fact that we have a 2 by 2 identity matrix as our forbidden subgraph, it can be extended to any forbidden matrix.

In general, consider a forbidden matrix F with dimensions k x l. If r is the number of distinct rows for a matrix with c columns and no subgraph induces F, then r is $\mathrm{O}(c^{2k-1})$ [6](check; where does l go?). We will use this to show that for any fixed F, the number of n by n matrices with no induced F is $2^{o(n^2)}$.

Partition the columns of the matrix into sets of size $\mathrm{f}(n)$. For each set of columns, the number of different rows with respect to these columns is $\mathrm{O}(f(n)^{2k-1})$. The entries in each set of columns can be reconstructed if we write down each row value with respect to these columns, and for each row of the matrix we give the index of the appropriate row value on these columns. We use $\mathrm{f}(n)$ * $\mathrm{O}(f(n)^{2k-1})$ bits writing down the row values, and $n\log\mathrm{O}(f(n)^2k-1)$ bits giving the appropriate index for each row of M on these columns. Since there are $\mathrm{O}(n/\mathrm{f}(n))$ sets of columns, the total number of bits used to store the matrix is $\mathrm{O}((n/\mathrm{f}(n)(f^{2k}(n)+n\log\mathrm{f}(n))$.

To minimize the number of bits used, we balance the terms $nf^{2k-1}(n)$ and $n^2\log n/\mathrm{f}(n)$

by setting these to be equal, getting the equation $f^{2k}(n) = n\log n$, or f$(n) = n^{1/2k} log n^{1/2k}$. The number of bits used to represent M in this way is $\Theta(n^{2-1/2k} log^{1-1/2k} n)$.

The bounds above show that forbidding any fixed induced submatrix F reduces the number of matrices to $2^{o(n^2)}$. Note that [6] and others were working on bounds for a much different problem, bounding the number of different rows with respect to the number of columns in a matrix without F as an induced submatrix. By chopping the matrix into groups of columns, we completely ignore copies of F which are not contained in a single block of the column partition, and thus are dramatically overcounting the number of possible matrices. If we want to find generalized implicit representations of matrices, we will need to be able to get more accurate estimations of the number of matrices which do not contain F. For example, this argument shows that the number of Γ-free matrices is at most $2^{n^{7/4} log^{3/4} n}$, while we know from chapter ? that the actual number of Γ-free matrices is $2^{\Theta(n log^2 n)}$.

We note that it is conjectured [6] that the true bounds on the number of distinct columns for a k by l forbidden submatrix is O(c^k) rather than O(c^{2k-1}), and this bound has been proven for many specific submatrices, including all 2 by 2 forbidden submatrices. However, we do not want to focus on the exact bounds known for the specific classes of matrices discussed above, since the bounds derived from this form of argument are almost certainly very far from optimal. The bounds on the number of distinct columns were not intended to be used to place bounds on the number of matrices, and I believe that better, as well as more intuitively appealing, bounds will come from trying to count the number of matrices in the classes directly. Readers interested in bounds on column size for other specific classes of matrices are referred to [6].

One interesting aspect of the bounds on the number of matrices in a class which are based on number of distinct columns is that these give natural representations using o(n^2) space which are fast for testing adjacency, but do not have any form of local structure. For each set of rows, we store all distinct columns in the matrix on that set of rows, and for each column we give the index of the corresponding column. Thus, to test the entry at position (i,j) of M, we might find that j corresponds to column number 243 on this set of rows, and then look up the value of row i in column number 243. However, this is not local; in order to make the space of the representation small, we cannot store all values of column 243 with row i.

Although the general desired result would be to take an arbitrary forbidden matrix F, and describe an implicit representation for matrices which do not contain F, we will state a number of relevant problems which are potentially easier.

Open Problem: If we know only that a forbidden submatrix F has i rows and j columns, improve the upper bound on the number of possible n by n matrices which do not contain F as an induced submatrix.

We are also interested in counting the number of matrices possible for specific values of F, such as I_2. Representation is open for these classes as well as counting, as we will discuss briefly below.

Open Problem: Find a way to take a forbidden matrix F, and design a storage scheme using $o(n)$ bits per row and column such that the entry at position i,j of any matrix without F as an induced submatrix can be determined only from the bits stored at i and j.

Of course, finding an implicit representation may be even harder. Both the counting and representation problem remain open for almost all matrices; exceptions are generally matrices with particularly simple or regular structure, and Γ.

Extending these bounds to higher dimensional matrices may be interesting, but will not be discussed here since it has relatively little connection to graph theory.

11.7 Other Matrix Properties

This section discusses classes of matrices which are not defined by specific forbidden submatrixes. A number of these classes are directly related to classes of intersection graphs.

A matrix has the consecutive 1s property for rows if the columns can be permuted so that the 1 values in each row occur consecutively. These matrices are closely related to interval graphs, since G is an interval graph if and only if the vertex/maximal clique incidence matrix has the consecutive 1s property. In fact, this is another way of describing Booth and Luecker's recognition algorithm for interval graphs, which uses PQ- trees to enforce maximal cliques containing a vertex to occur consecutively.

A bipartite graph is convex if the vertices of one color class can be ordered so that for each vertex x of the other color class, neighbors of x occur consecutively; a bipartite graph is doubly convex if both color classes can be ordered in this fashion. Bipartite permutation graphs $\subset$ doubly convex graphs $\subset$ convex graphs $\subset$ chordal bipartite graphs (exercise ?). Clearly G is doubly convex if and only if the bipartite adjacency matrix has the consecutive 1s property for both rows and columns, while G is convex if the matrix has the consecutive 1s property for either rows or columns. PQ-trees can obviously be used to recognize convex or doubly convex in linear time.

Matrices with the consecutive 1s property for rows have an implicit representation. Order columns by their position in their permuted matrix, and keep for each row r the value of the first and last columns which have a 1 in position r.

It seems natural to consider the class of matrices which can be permuted so that no pair of 1s occur consecutively in any column as well, though I have never seen any work on this class. There are $2^{\Theta(n^2)}$ n by n matrices in this class (exercise ?), so the representation problem is not interesting. The recognition problem might be of more interest for the class.

11.8 Exercises

1) Design and analyze an efficient recognition algorithm for matrices in PM(1 0)

2) Show that not every I_2-free matrix can be permuted to avoid Γ.

3*) Show that not every Γ-free matrix can be permuted to avoid I_2.

4) Prove that every bipartite permutation graph is biconvex.

5) Show that $2^{\Theta(n^2)}$ matrices have the nonconsecutive 1 property.

6) Design a simple $O(nm)$ algorithm for recognizing linear matrices by testing whether each 1 entry is part of a forbidden matrix in $O(m)$ time.

7) Show that matrix multiplication can be used to recognize linear matrices.

8) Design an algorithm which uses $O(mn^{.5})$ time and space for recognizing linear matrices.

9) Show that one cannot recognize linear matrices by performing a doubly lexical ordering, and looking for the forbidden matrix by looking for the next 1 in the row and column of each entry.

10) If $N(c_1)$ is a subset of $N(c_2)$ and c_1 comes after c_2 in a $\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$ free ordering, show that we can swap c_1 and c_2.

11) A simpler method for counting the number of $\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$ free matrices works as follows. For each column, store the changes from column i to column $i+1$ (i.e. which 1 values become 0, and which 0 values become 1). Show that for any matrix in this class, the total number of changes is at most $2n$.

12*) Show that for linear matrix recognition, can test any row for a violation in $O(n)$ time.

13) Show that an arbitrary graph G contains an induced C_4 if and only if $B(G)$ contains a C_4.

14) The arboricity of a graph is the minimum number of forests necessary to cover all edges of G (check). Show that if a bipartite graph has no C_4, it has arboricity at most $n^{.5}$.

15) Show that there are $2^{\Omega(nlogn)}$ I_2-free matrices, even if no permutation of rows and columns is allowed.

12 Miscellaneous Classes

In this chapter, we deal with classes of graphs which have interesting forms of representation which are not covered in other chapters. The two broad groupings we will look at are classes defined by elimination schemes, and classes defined in terms of graph decompositions.

12.1 Elimination Schemes

We have already seen one important elimination scheme which characterized a graph class, in the correspondence between chordal graphs and graphs with perfect elimination schemes in chapter ?. Elimination schemes and characterizations are often desirable. Typically, we have some local property (for perfect elimination schemes, N(v) is a clique) which allows us to remove a vertex simply, and this can sometimes be sufficient to recognize a graph class defined by a global property (G has no induced cycles of length greater than 3). Elimination schemes also are useful in optimization problems.

Here, we deal with classes which are defined by elimination schemes. In several of these cases, the scheme itself gives rise to a representation which uses o(n^2) space, but is not generalized implicit.

12.1.1 Distance Hereditary Graphs

Distance hereditary graphs are usually defined in terms of a global property; G is distance hereditary if and only if for every pair of vertices x, y, all induced paths between x and y have the same length. If we think in terms of elimination schemes, we want to find vertices which can be removed via local testing which will not affect whether the graph is distance hereditary.

Vertices u, v are called twins if N(v) - u = N(u) - v; twins correspond to modules of size 2. It is not hard to see that removing one vertex from a pair of twins cannot affect whether a graph is distance hereditary. It is also not difficult to see that if w is a degree 1 vertex, G is distance hereditary if and only if G - w is distance hereditary (exercise ?). These simple observations give rise to an elimination scheme characterization of distance hereditary graphs. A sequence v_1 .. v_n is a pruning sequence if each v_i is either a degree 1 vertex in the graph induced by $v_{i+1}..v_n$, or a twin of some vertex in the graph induced by $v_{i+1}..v_n$.

Theorem [12]: G is distance hereditary if and only if G has a pruning sequence.

The characterization above shows that the number of distance hereditary graphs is $2^{O(nlogn)}$, since we can reconstruct G from the pruning sequence if we know the neighbor of each degree 1 vertex in the sequence, the twin of each twinned vertex, and whether the twins are adjacent. This does not give us an immediate implicit representation, however; if we know that u was created as the twin of x and v was created as the twin of y, we do not know whether u and v are adjacent. However, it is not hard to see from this characterization that every distance hereditary graph is a circle graph, since twins and degree 1 vertices can easily be added to a circle model (exercise ?). Pruning sequences are also useful for optimization problems on distance-hereditary graphs. We will present a linear time recognition algorithm for distance hereditary graphs below. Proofs of the correctness of the algorithm can be found in [91].

First, let us note that pruning sequences for cographs can be found in linear time; in fact, cographs are exactly equal to graphs which have pruning sequences without the pendant vertex operation (exercise ?). As discussed in chapter ?, we can construct a cotree T for any cograph in linear time. Any pair of leaves of the cotree with a common parent p in T are twins; all leaf siblings can be compressed to a single vertex, which replaces p in T; this process is repeated until a single vertex remains.

Perform a breadth first search from any vertex v; vertices are now divided into levels. The pruning sequence is composed by working backwards from the last level; let j be the current last level. Each connected component of level j is a P_4-free module [91], and is compressed to a single vertex as in the previous paragraph. Order the remaining vertices w_1, w_2,...,w_k of level j by increasing degree in the current graph, and remove any degree 1 vertices. The neighbors of w_1 form a P_4-free module M; M is shrunk to a single vertex, after which w is removed as a degree 1 vertex. After this, the neighbors of w_2 are a P_4-free module, and the same process is repeated for w_2,...,w_k.

Every vertex is in at most one call to a cotree construction algorithm on an induced subgraph of G, so the total time taken on cotree construction is $O(m+n)$. Other steps clearly can be performed in linear time.

The relationship between join decomposition and distance hereditary graphs is discussed later in this chapter. This led Dahlhaus [47] to modify the algorithm above to find the join decomposition of an undirected graph in $O(n+m)$ time, using a modular decomposition algorithm as a subroutine.

12.1.2 Strongly Chordal Graphs

For our purposes, strongly chordal graphs, which were first studied in [63], are defined by the following elimination scheme. A vertex v is simple if $N(v)$ is a clique, and the neighborhoods of vertices in $N(v)$ are ordered by set containment.

A simple elimination ordering is an ordering of the vertices $v_1..v_n$ so that each v_i is simple in the subgraph induced by $v_i + 1..v_n$. G is strongly chordal if and only if G has a simple elimination ordering.

It is easy to see that the graphs of figure ? are not strongly chordal, since they have no simple vertex. It can be shown that G is strongly chordal if and only if G is chordal and has no induced 'sun' of size at least 3, where a sun is a clique of size k connected cyclically to an independent set of size k as in figure ? [].

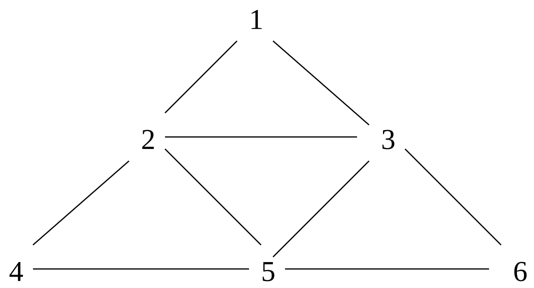


Figure 15: The 3-sun: Chordal, but not Strongly Chordal

First, let us note that the definition of strongly chordal graphs immediately gives a representation with $o(n^2)$ bits. Let v be a simple vertex. If $|N(v)|$ is at most $n^{.5}$, we can certainly eliminate v, using at most $n^{.5}\log n$ bits. Suppose that $|N(v)|$ is at least $n^{.5}$. Then we can eliminate v and all neighbors of v by ordering $N(v)$ by degree, and maintaining for each vertex w in V - $N(v)$ the number of neighbors of w which are adjacent to v; this information can be stored with $O(n\log n)$ bits. If we repeatedly remove low degree simple vertices or high degree simple vertices and their neighborhoods, we obtain a set of $O(n^{1.5} \log n)$ bits from which G can be reconstructed. Thus, the problem of finding a generalized implicit representation for the class is potentially interesting.

In fact, the number of graphs in the class is much smaller than $2^{n^{1.5}logn}$, as can be seen from the close correspondence between strongly chordal graphs and chordal bipartite graphs studied in chapter ?. This correspondence can be shown via direct reduction, or by the following characterization of strongly chordal graphs. The augmented adjacency matrix of the graph is the matrix formed by replacing 0s along diagonal entries with 1s.

Theorem []: G is strongly chordal if and only if the vertices can be ordered so that the augmented adjacency matrix is Γ-free [].

Theorem ? shows that the number of strongly chordal graphs is $2^{O(nlog^2n)}$, since we showed that the number of Γ-free matrices is $2^{\Theta(nlog^2n)}$ in chapter ?. However, it may not be clear that this bound is tight, since not every Γ-free matrix is the augmented adjacency matrix of a strongly chordal graph; for example, augmented adjacency matrices are always symmetric. Often, however, chordal bipartite graphs and strongly chordal graphs can be viewed as different ways of looking at an underlying Γ-free matrix. To show that this bound is tight, we will show how to transform these classes into each other.

Suppose that G is a graph. Consider the graph $B(G)$ formed by making two copies v_1, v_2 of each vertex v, and making x_1 and y_2 adjacent if $x = y$ or x and y are adjacent in G. If G is strongly chordal, the Γ-free ordering of the augmented adjacency matrix of G immediately gives a Γ-free ordering of $B(G)$ (the matrices are the same), so $B(G)$ is chordal bipartite. If G is not strongly chordal, G is either not chordal or G has a k-sun. In exercise ?, it is shown that either case implies a chordless cycle of length at least 6 in $B(G)$, so G is strongly chordal if and only if $B(G)$ is chordal bipartite.

Consider a bipartite graph $G = (X,Y,E)$. Let G' be the graph formed by making Y into a clique. Then G is chordal bipartite if and only if G' is strongly chordal.

Suppose that G is chordal bipartite, and assume that we have a Γ-free ordering of the bipartite adjacency matrix of G. The ordering of Y as in the Γ-free ordering, followed by any ordering of X, must be a simple elimination scheme for G'. Otherwise, let x_1 be the first vertex of X which is not simple in this scheme. There must be neighbors $y_1 < y_2$ of x_1 such that $\mathrm{N}(y_1)$ is not a subset of $\mathrm{N}(y_2)$ in the graph induced by the remaining vertices; let x_2 be in $\mathrm{N}(y_1)$ - $\mathrm{N}(y_2)$. Then rows x_1, x_2 and columns y_1, y_2 would induce a Γ in the ordering of G. If G is not chordal bipartite, then G' is not strongly chordal, since no vertex from a chordless cycle of length at least 6 in G can be simple.

I should note that the simplicity of the reductions between the two classes of graphs had always led me to believe that it was not useful to solve problems separately on the two classes, but there is one important issue which this viewpoint made me overlook. I have looked only at recognition of chordal bipartite graphs; I feel that there should be a linear time algorithm for this, since (as stated in chapter ?) current algorithms do more than is necessary, finding a doubly lexical ordering of an arbitrary matrix without using any special properties which must hold if the matrix has a Γ-free ordering. Therefore, I was very interested when a paper appeared which claimed a linear time algorithm for recognizing strongly chordal graphs. Although the paper had serious, unfixable flaws, it made me realize that the reductions above are not symmetric with respect to cost. The reduction from strongly chordal recognition to chordal bipartite recognition is linear; this implies that any linear algorithm for recognizing chordal bipartite graphs would give a linear algorithm for recognizing strongly chordal graphs. However, the reduction from chordal bipartite recognition to strongly chordal recognition involves adding $\Theta(n^2)$ edges. Thus, it is possible to have a linear time algorithm for strongly chordal graph recognition which does not imply a linear time algorithm for chordal bipartite graph recognition, and I might advise the reader interested in solving an open problem to try the former problem first.

12.2 k-Simplicial Elimination Schemes

I find the following elimination scheme to be a natural extension of perfect elimination schemes for chordal graphs. The class has not been studied previously, and I have decided to include it for the reader despite the fact that it is not closely related to notions of representability of graph classes.

A perfect elimination scheme is a very satisfying characterization of chordal graphs, since it allows testing by local properties (are the neighbors of v a clique) whether the graph has a global property (does G contain a hole of size > 3). When studying graphs without holes of size $> k$ for other values of k, it is natural to extend the definition of a simplicial vertex. Clearly, a vertex which is not in the center of an induced path on k vertices cannot be part of a chordless cycle of length greater than k. A vertex v is k-simplicial if v is not in the center of a path of length k; simplicial vertices correspond to 3-simplicial vertices. A graph has a k-simplicial elimination scheme if we can successively remove k-simplicial vertices until no vertex remains.

In general, k-simplicially reducible graphs are not equivalent to graphs without holes of size $> k$ (exercise ?). Chordal bipartite graphs are equivalent to bipartite graphs with 5-simplicial elimination schemes (exercise ? chapter ?). It is not clear whether this equivalence holds for other classes of interest. Given that the property is local, it may be easier to test for k-simplicially reducible graphs than for graphs without holes of size $> k$, though I know of no work on these classes.

In general, there are too many k-simplicially reducible graphs to have an interesting form of representation, since every split graph is 4-simplicially reducible. For bipartite graphs, these classes may become smaller; this is related to the open question discussed in chapter ?.

(West question? other bipartite?)

12.3 Doubly and Dually Chordal Graphs

Two other classes which come up in chapter ? are defined by elimination orderings. In order to see why these classes are defined, let us consider a use of a strong elimination ordering. Suppose that you want to solve the dominating set problem, and you are given a strong elimination ordering for G. It is clearly best to choose the maximum degree neighbor of the first vertex to cover this vertex; we continue in a similar fashion stepping through the ordering, and for each undominated vertex we encounter we choose its maximum neighbor in the remaining graph.

For this problem, it is not important that all neighborhoods of neighbors of v are ordered by set containment, as long as we know there is some neighbor w of v such that $\mathrm{N}[w]$ contains the neighborhood of all other neighbors of v; such a neighbor is called a maximum neighbor of v. A graph is doubly chordal if there is an elimination scheme such that each vertex removed is simplicial and has a maximum neighbor in the remaining subgraph. A graph is dually chordal if there is an elimination scheme such that each vertex contains a maximum neighbor in the remaining subgraph.

Although there are too many graphs in each class to have an efficient representation (exercise ?), there are linear time algorithms to construct each of the elimination schemes on the class which allows optimization algorithms to be solved in linear time. In fact, these algorithms are considerably simpler than finding the doubly lexical ordering of the adjacency matrix, which is used to find the elimination ordering of a strongly chordal graph.

The algorithms for constructing maximum neighborhood orderings for dually chordal graphs and perfect maximum neighborhood orderings for doubly chordal graphs are due to Brandstädt, Chepoi, and Dragan [24]; that paper presents correctness proofs for the algorithms as well.

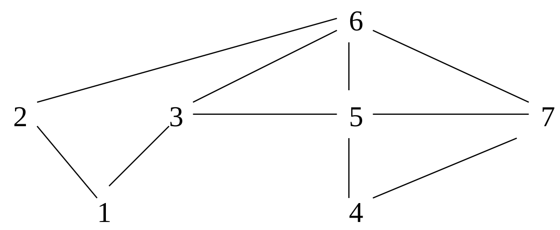


Figure 16: Sample Graph for Maximum Neighborhood Ordering

Vertices are placed in 3 categories; unnumbered, numbered, and marked; note that marked vertices are also considered to be numbered. Initially, all vertices are unnumbered. Repeatedly select and mark a numbered (but not marked) vertex u adjacent to the maximum possible number of numbered vertices; at the first step, any vertex is chosen as u. Assign all unnumbered neighbors of u maximum remaining numbers (in any order); u will be the maximum neighbor of these vertices (this is denoted by $\mathrm{mn}(x) = u$).

Consider the graph of figure ?; note that this graph is not dually chordal. Initially, we may choose vertex 7 to go last; at this point we place 4,6,5 in any order before 7, and make 7 the maximum neighbor of these vertices. Vertex 5 must be the next vertex marked, since it is adjacent to three numbered vertices while 4 and 6 are adjacent to only two. Vertex 3 is numbered and placed before 4,6,5; $\mathrm{mn}(3) = 5$. Vertex 6 is marked next; vertex 2 is numbered at this point, with $\mathrm{mn}(2) = 6$. Vertex 3 or 4 can be chosen at the next step; 4 numbers no new vertices while 3 numbers 2. The final ordering may be as follows: 1(3) 2(6) 3(5) 5(7) 6(7) 4(7) 7, with maximum neighbors in parentheses. 3 is not a maximum neighbor of 1, since 3 is nonadjacent to 2; thus, this graph is not dually chordal.

One can use standard data structures to implement this algorithm to run in linear time (exercise ?). Of course, this algorithm will construct an ordering even if the graph is not dually chordal, as shown above, so we must verify that the result is a maximum neighborhood ordering. The paper [24] tests this indirectly, verifying that the input is dually chordal by testing a corresponding hypergraph property.

For our verification algorithm, let us modify the ordering found above slightly; vertices which are numbered at the same time are reordered on the basis of when they were marked in the algorithm; vertices marked earlier are placed later in the ordering. Thus, in the example above, we change the ordering from 1 2 3 5 6 4 7 to 1 2 3 4 5 6 7. This reordering can be done in linear time (exercise ?). Suppose that we want to test whether the modified algorithm above produces a maximum neighborhood ordering. We say a vertex x has been verified if we have checked that $\mathrm{N}(x)$ was a subset of the neighborhood of some other vertex. Step through the elimination scheme in order; let v be the next vertex encountered. If v has a neighbor which comes after $\mathrm{mn}(v)$, the ordering is not a maximum neighborhood ordering. For each neighbor w of v in the remaining subgraph, if w is unverified and w is not $\mathrm{mn}(v)$, verify that $\mathrm{N}(w) \subseteq \mathrm{N}(\mathrm{mn}(v))$ in the remaining subgraph. After all neighbors are verified, we verify that $\mathrm{N}(v) \subseteq \mathrm{N}(\mathrm{mn}(v))$ and remove v from the graph. Testing for neighborhood containment can be done in O(N(w) time, either by using the technique from problem ? (AHU 5.8) or by adding neighbors of w to a list which is checked by mn(v) immediately before mn(v) is removed. Since each w is checked once for neighborhood containment, the total time taken is O(n+m).

If the elimination scheme is correct, we will never find a violation in the verification phase. If v has a neighbor z which comes after mn(v), then mn(z) must be adjacent to z but not mn(v); this was the point of the reordering step, since if mn(v) and z share the same mn value, z will be chosen as the maximum neighbor of v. Thus, the algorithm will answer that this is a maximum neighborhood ordering.

Suppose that our elimination scheme is not correct, i.e. for some v there is a neighbor w after v such that N(w) is not a subset of N(mn(v)) in the subgraph induced on the remaining vertices. Let v be the first vertex of the scheme for which mn(v) is not a maximum neighbor, and let w be the last vertex of N(v) in the scheme such that N(w) is not a subset of N(mn(v)). If w comes after mn(v), or w is unverified, we find the violation at this step. Suppose that N(w) was verified to be a subset of N(v_i); since mn(x) is always the last vertex of the scheme adjacent to x, v_i comes after w in the ordering. The neighborhood of w in the remaining subgraph is a subset of N(v_i), so v_i must also be a neighbor of v and not satisfy N(v_i) $\subseteq$ N(mn(v)), contradicting the assumption that w was the last neighbor of v in the ordering causing a violation.

If we want to create an ordering for a doubly chordal graph which is both a perfect elimination scheme and a maximum neighborhood ordering, we make a simple modification to the maximum neighborhood ordering above. When u is chosen as a maximum neighbor, instead of placing neighbors of u in any order before u, we repeatedly choose neighbors of u which are adjacent to the maximum number of vertices already in the output list, placing each in the output list after all remaining vertices, until all neighbors of u are exhausted. For example, in the graph of figure ? above with 7 chosen to go last, we could not choose 4 and then 6; if 4 is chosen, 5 must be the next vertex chosen in the output list. G is doubly chordal if and only if G is both chordal and dually chordal [24], so the linear algorithms for recognizing these classes allow us to determine whether the above ordering is correct in linear time.

12.4 Decomposition

Another method of describing a graph class which gives a space efficient representation which is not implicit is to define the class in terms of a decomposition tree.

12.4.1 Substitution

The starting point for many graph classes based on decomposition is the substitution decomposition. G can be reconstructed from the decomposition tree, plus representative graphs for each module, where a representative graph is the graph induced by taking one vertex from each child in the decomposition tree.

The ultimate restriction of substitution decomposition is to allow no prime modules in the decomposition tree. This gives exactly the class of cographs, discussed in chapter ?. As stated in that chapter, cographs have an implicit representation by virtue of being contained in the permutation graphs.

We can extend the above graph class and still have a graph class with $2^{O(nlogn)}$ members if we allow prime modules to have at most k children; this class is called the class of graphs with bounded decomposition diameter. It is not clear whether graphs with bounded decomposition diameter have an implicit representation.

We can also allow other operations which reduce a graph G to G' in such a way that G can be reconstructed from G' plus O(log$n$) bits to be combined with bounded diameter decomposition. For example, [] allows removal of degree 1 vertices to precede decomposition into at most k submodules. This can allow more graphs to be represented, as shown in exercise ?. For space efficient representation, this could trivially be extended to removing degree k vertices for any fixed k. Many other operations which can be represented efficiently could also be considered.

12.4.2 Join Decomposition

The join decomposition is discussed in chapter ? as part of a circle graph recognition algorithm. In join decomposition, vertices of a graph G are partitioned into V_1, V_2 such that both V_1 and V_2 have at least two vertices, and edges which go between the two sets induce a complete bipartite graph. The graph is then decomposed into subgraphs induced by $V_1 \cup m$ and $V_2 \cup m$, where m is a new "marker" vertex adjacent to those vertices which have a neighbor in the other set; each of these subgraphs is decomposed recursively.

Since the join decomposition can be viewed as an extension of the substitution decomposition, it seems natural to try to restrict join decomposition in the same types of ways that substitution decomposition has been restricted. However, there is relatively little work dealing with restricted cases of join decomposition. This may be because the decomposition is less well known, or because join decomposition is not as helpful when solving optimization problems (compare [] with []).

The one major class defined by restricting join decomposition is the class of completely separable graphs; these are graphs which are decomposed by the join decomposition into three vertex subgraphs. This class is exactly equal to the class of distance hereditary graphs discussed above [], which explains why distance-hereditary graphs have also been called completely separable graphs.

A natural generalization which has not been studied, but which leads to a class of graphs with few members, is to restrict the number of vertices in prime components to at most k for larger fixed values of k; in completely separable graphs, k is at most 3.

12.4.3 Generalized Join Decomposition

We will describe an extension of join decomposition due to Hsu [100] which raises a number of interesting open problems, and try to create restrictions which can have efficient representation schemes.

In the join decomposition, all vertices which have edges to the other side of the partition must have the same neighbors in the other side of the partition. In the generalized join decomposition, vertices can have different neighbors on the other side, but these neighborhoods must be ordered by set containment. More formally, vertices are partitioned into V_1, V_2 so that the edges which cross the split are a chain graph; that is, there would be no induced $2K_2$ on the subgraph formed by these edges. We then add a marker vertex for each different neighborhood. In order to make the subgraphs of the partition smaller than G, Hsu requires the graphs induced by V_1 and V_2 to each contain at least 1 edge; other conditions, such as requiring some vertex in each subgraph to have no neighbors in the other subgraph, or requiring some pair of vertices to have the same neighborhood in the other subgraph, could also be used. Hsu was able to show that G is perfect if and only if all subgraphs found by the generalized join decomposition are perfect; the idea comes from the fact that any long hole or antihole will continue to exist in one of the components after the decomposition.

This decomposition raises a number of interesting questions.

Can you find a split with respect to the generalized join decomposition in polynomial time? More generally, we can define a graph decomposition by forbidding any fixed forbidden bipartite subgraph (or set of subgraphs) in the graph formed by taking all edges which cross a split.

It can be shown [] that it is NP-complete to determine whether a graph is decomposable with respect to the following forbidden subgraphs: $2K_2$ (i.e. generalized join) and $K_{2,2}$. Another natural decomposition which is NP-complete to find requires that the edges connecting the two sets form a matching; this is called a symmetric $K_{1,2}$-free decomposition in []. On the other hand, it is possible to find a $K_{1,i}$ free decomposition for any fixed i if this means only that verteices in color class 1 must have at most i neighbors on the other side for any fixed i in polynomial time (exercise ?). Note that forbidding a graph with 1 vertex in A, 2 vertices in B, and one edge from an a vertex to a B vertex is equivalent to requiring that B is a module.

Open Problem: Can you determine which forbidden subgraphs define a decomposition which can be found in polynomial time?

Note that if we have a split V_1, V_2 defined by the generalized join decomposition, the number of bits needed to represent G is $n \log n$ + the number of bits to represent the subgraphs induced by V_1 and V_2 (exercise ?); if V_1 and V_2 are both large this is a vast improvement over the $\Theta(n^2)$ that we would need in general. For the purposes of representation, however, we may be interested in restricting the generalized join decomposition rather than extending it. The following restrictions seem natural, and may lead to graph classes which can be represented efficiently through the decomposition tree, but do not obviously have an implicit representation. Recognizing any of these restricted classes in polynomial time would be interesting.

A first restriction is to define the class of completely decomposable graphs with respect to the generalized join decomposition. If we change the definition slightly, as in exercise ?, all permutation graphs are completely decomposable.

Another restriction which reduces the amount of information which must be shared between subgraphs of a split is to allow only a constant number of marker vertices on each side of the split; i.e. a constant number of different neighborhoods when restricted to the other subgraph. This notion arises again during our discussion of algorithms for dealing with graphs having bounded cliquewidth below.

12.4.4 Recursively Decomposable Graphs

This section deals with an attempt by Borie, Parker, and Tovey to find a unifying idea behind graph classes defined in terms of decomposition schemes. By focusing on a unifying approach, the authors raised some interesting issues dealing with the distinction between having an efficient representation and finding an efficient representation. The results in this section are taken from [20].

A number of graph classes are defined using the following type of construction, which can generally be viewed either as a composition or as a decomposition. Graphs are composed or decomposed in such a way that interactions between the graphs is localized to a small subset of vertices, which are called terminals. A k-terminal graph is a graph in which at most k vertices are designated as terminals.

Viewing the operation as a composition, we combine k-terminal graphs according to one of a fixed set of rules. The composition rules may combine the graphs by identifying terminals of one graph with terminals of the other graph, and choosing a new set of terminals from the set of terminals of the two subgraphs. Edges in each of the original subgraphs are preserved in the new graph. A graph class is defined by allowing all graphs which can be built up using these rules from a finite set of base graphs.

We can view this process as a graph decomposition as well, by allowing a graph G to be decomposed into G_1, G_2 if G_1 and G_2 can be combined to get G using a composition rule. This gives rise naturally to a decomposition tree; an internal node is added for the decomposition step (typically labeled by the type of rule used), with recursive decompositions of G_1 and G_2 as the children of the node. A graph G is in the class if G has a decomposition tree such that all leaves are base graphs. Thus, classes which can be defined in this way are called recursively decomposable classes of graphs.

Examples of recursively decomposable graph classes are trees, series parallel graphs (see problem ? of chapter ?), Halin graphs [], and partial k-trees []. Many optimization problems can be solved easily if we are given a decomposition tree/composition sequence. This can be done since interactions between graphs are restricted to the terminals. For example, let us assume that we want to solve the maximum clique problem, and we are given a composition sequence. For each graph in the sequence, we compute and store 2^k values, where k is the maximum number of terminals; each value corresponds to the maximum clique if we are restricted to use a particular subset of the terminals. It is easy to compute the values for the combined graph if we know the values of the subgraphs, and we know which composition rule was used to get the new graph.

Note that the decomposition tree of a recursively decomposable graph gives a space efficient representation, which is not necessarily implicit. In fact, graph classes defined using this technique must be hereditarily sparse (every subgraph contains a vertex of degree at most $2?k$), and have an implicit representation for this reason. Although this may seem to resemble the notion of bounded substitution decomposition diameter, neither notion is stronger than the other (exercise ?).

By focusing on a general technique rather than on a particular class, Borie, Parker, and Tovey noted an important distinction between different decomposition rules. Since many problems are efficiently solvable on recursively decomposable graphs if given a decomposition sequence, we want to be able to take an input graph G and find a decomposition if it exists. The natural approach, which works on many classes, is to find any composition rule which breaks G into G_1 and G_2, and to attempt to decompose G_1 and G_2 recursively. Since there is a fixed number of rules, and a fixed number of ways to choose terminals for each subgraph, it is not difficult to make the argument that we can find a decomposition into G_1, G_2 in polynomial time if such a decomposition exists (exercise ?).

However, finding a single legal decomposition step efficiently does not necessarily imply that an entire decomposition sequence can be found efficiently. It is possible that if one rule is used at the first step, the graph decomposes into G_1 and G_2, where both G_1 and G_2 are recursively decomposable, but that if another rule is used, G may be decomposed into G_1' and G_2', where one or both of G_1',G_2' is not recursively decomposable using these rules. Readers familiar with the Church-Rosser property will be comfortable with this distinction. [20] calls a class of graphs defined by a set of decomposition rules deterministic if membership can be decided by arbitrary application of any valid rule to decompose a graph. Recognition and optimization problems can be solved in polynomial time for all deterministic classes, while this is not necessarily true for recursively defined classes which are not deterministic.

Borie, Parker, and Tovey classify decomposition rules as follows. A (k,u,r) operation works by decomposing a k-terminal graph G into two k-terminal graphs so that at most r vertices of G were terminals in either subgraph, with at most u of these becoming nonterminals in G. The class [k,u,r] is defined by allowing all (k,u,r) operations. They then show that the class [k,u,r] is deterministic if and only if it has one of the following forms; [2,r-2,r], [3,r-3,r], [k,0,k], or [k,1,k+1].

The immediate consequence is that if all rules are of one of the forms given above, you know that the class deterministic; this holds for such classes as Halin graphs and biconnected series-parallel graphs.

12.4.5 Cliquewidth and NLC-width

The following interesting notions of graph decomposition were first studied by Courcelle []. Certain types of logical queries can be answered in polynomial time if we are given this form of decomposition for graphs of bounded cliquewidth or NLC-width; this leads to open problems which will be discussed in the optimization chapter.

The cliquewidth of G is the minimum number of labels we need to construct G from a set of isolated vertices, if the following constructions are allowed. We give a method for building up graphs with cliquewidth at most k.

Start with any graph on at most k vertices; give each vertex its own label. At any time, we may make a label change as follows: change all vertices with label i in a cliquewidth k graph to have label j. At any time, we may add edges as follows: for some pair of labels i, j, add undirected edges from all vertices with label i to all vertices with label j. We may also combine two graphs with cliquewidth k using the disjoint union operation. The graph can be completely described by a composition tree, in which internal nodes have a description of the operation performed on the subtree of descendants of that node.

Cliquewidth is closely related to a parameter called NLC-width. In building a graph with NLC-width k, we again are allowed to use k labels on vertices, and can at any time perform a relabeling procedure exactly as in the definition of cliquewidth. The primary difference between the classes comes in the way in which edges are introduced; for NLC-width k graphs, edges are added between subgraphs when the union operation is performed. To combine two NLC-width k graphs, we give a k by k bipartite graph, and add edge between vertices with label i from graph 1 to vertices with label j from graph 2 if and only if there is an edge from i to j in the given bipartite graph. We are also allowed to perform a relabeling operation, telling us how labels should be assigned in the new merged graph.

In exercise ?, you are asked to show that the cliquewidth and NLC-width can differ by at most a factor of 2. Since our concern is primarily with the fact that the number of labels is constant, we will be able to speak interchangeably of graphs with bounded cliquewidth and graphs with bounded NLC-width. Graphs with cliquewidth 2 (or NLC-width 1) are exactly the cographs (exercise ?).

Since two graphs are combined using operations which can be described using $O(\log n)$ bits for graphs of fixed cliquewidth, the total number of bits needed to reconstruct a graph of fixed cliquewidth is $O(n \log n)$; however, this representation is not implicit. We will describe an implicit representation for cliquewidth k graphs below; we note that this is the first time we will create an implicit representation that does not immediately give a constant time adjacency testing algorithm.

Both the recognition and construction problems are solved for cliquewidth 3 [][], but are open for graphs of cliquewidth k for all k ¿ 3. Since constructing the decomposition allows you to solve optimization problems for graphs of fixed cliquewidth, even an approximation (for example, giving a construction sequence which never uses more than $2k$ labels for graphs of cliquewidth k) would be very useful.

Open Problems: Recognize graphs with cliquewidth k, for $k > 3$.

Open Problem: Find the appropriate construction sequence (or a good approximation) for graphs of fixed cliquewidth $k > 3$.

We now discuss a new approach which gives an implicit representation of graphs with bounded cliquewidth, and allows certain optimization problems to be solved in polynomial time on the class even when the graph is given in adjacency matrix form.

A set S of vertices is a k-module if S can be partiitioned into $S_1, S_2, ..., S_k$ such that for each S_i, S_i is a module in the graph induced by V - $S \cup S_i$; i.e., the vertices of S_i are indistinguishable to vertices of V - S. This is a generalization of the concept of homogeneous pair, as discussed in [].

A balanced k-module is a k-module of size at least $|V|/3$, and at most $2|V|/3$. In exercises ? and ?, we ask the reader to show that every graph with cliquewidth k has a balanced k-module, and that the k-module can be found in polynomial time given only the adjacency matrix for G.

We can use balanced k-modules to get an implicit representation of cliquewidth k graphs as follows. Construct a tree, which we will call a balanced k-module decomposition tree, from G by recursively finding a balanced k-module in the current set, and placing the k-module in the left subtree and other vertices in the right subtree. Since the size of each set is at most 2/3 the size of the set at the previous level, the height of such a tree is O(log$n$).

For a vertex v, we store a constant number of bits for each level of the balanced k-module decomposition tree. We store one bit which tells us whether v is placed in the right or left child at this level. For a vertex in the left child (i.e. part of the k-module) we store the value i such that S_i contains v. For a vertex v in the right child, we store all subset numbers corresponding to subsets adjacent to v; since k is constant, this is a constant nmumber of bits.

We can determine whether vertices x, y are adjacent by finding the first level at which x and y are placed in different children, and then determining whether the vertex placed in the right child is adjacent to the subset S_i containing the vertex placed in the left child. It is easy to determine this from the bits stored at x and y, so this is an implicit representation. Note, however, that the time needed to determine adjacency in this way is O(log$n$), leaving the following open problem.

Open Problem: Find an implicit representation of cliquewidth k graphs such that adjacency can be determined in constant time.

We can use the fact that every cliquewidth k graph has a balanced k-module to solve many optimization problems on the class in polynomial time, if the decomposition is not given as input. A very nice general scheme for solving optimization problems if the decomposition is given was presented in []; we note that this is essentially a dynamic programming approach, keeping track of all "relevant" solutions for a portion of the decomposition tree.

The balanced k-module approach uses recursion rather than dynamic programming. We will illustrate the technique on the vertex cover problem; this is generalized to a wide variety of problems in [].

Let S be a balanced k-module, and let $S_1, S_2, ..., S_k$ be the subsets of S. We can determine whether a set is a vertex cover in G if we know which vertices of V - S are in the set, and which sets S_i are completely contained in the vertex cover. This leads to a recursive technique in which we consider all of the 2^k possibilities as to which sets S_i are contained in the cover. For each possibility, we solve two problems recursively; find the optimal cover of the sets S_i not contained in the cover, and find the optimal cover of vertices of V - S which are not adjacent to any sets S_i which are not fully contained in the cover. The cost of the possible choice of sets S_i is then the sum of the costs of these two subproblems, plus the number of vertices in the sets S_i contained in the cover and the number of vertices of V - S which are adjacent to some set S_i which is not fully contained in the cover. This leads to a recursion of the form $\mathrm{T}(n) \leq 2^{k+1}\mathrm{T}(2n/3) + \mathrm{O}(n^2)$, which gives a polynomial solution for any fixed value of k.

Many problems are solved directly from this recursive approach; in [], we show that Courcelle's formalization of problems expressible in a form of second order monadic logic if the cliquewidth k decomposition tree is given can be solved in polynomial time even if the tree is not given; the time bound goes from linear to $n^{f(k)}$.

In [], we study the extension of second order monadic logic by a new operator. If we use this operator, more problems are expressible (examples include partial order dimension k, isomorphism, and perfection), but there is a cost. If the cliquewidth k decomposition tree is given, these problems have polynomial solutions, but if the tree is not given, then the time complexity is governed by the a recurrence of the form $\mathrm{T}(n) = n^{f(k)}\mathrm{T}(2n/3) + \mathrm{f}(n)$, which leads to a superpolynomial but subexponential algorithm. Interestingly, cliquewidth is itself expressible using the new operator, which implies a subexponential algorithm for the cliquewidth k problem; this makes it seem unlikely that the clqiuewidth k problem is NP-complete.

12.4.6 Clique Separator Decomposition

This form of graph decomposition is useful for solving optimization problems on some graph classes defined in this book, such as EPT graphs, as will be shown in chapter ?. We give an efficient algorithm for finding the decomposition, and define a new restriction which could give rise to classes which have efficient representations.

A clique separator is a clique C such that the subgraph induced by G - C is disconnected. The clique separator decomposition takes any clique separator C of G splits G into subgraphs consisting of C plus each connected component of G - C, and decomposes each subgraph recursively. The first polynomial algorithm for finding a clique separator was due to Whitesides [177] and found a single separator in $\mathrm{O}(nm)$ time; this implies an $\mathrm{O}(n^3m)$ clique separator decomposition algorithm (exercise ?). The following algorithm is due to Tarjan, and although a single clique separator still is found in $\mathrm{O}(nm)$ time, the entire clique separator decomposition is computed in

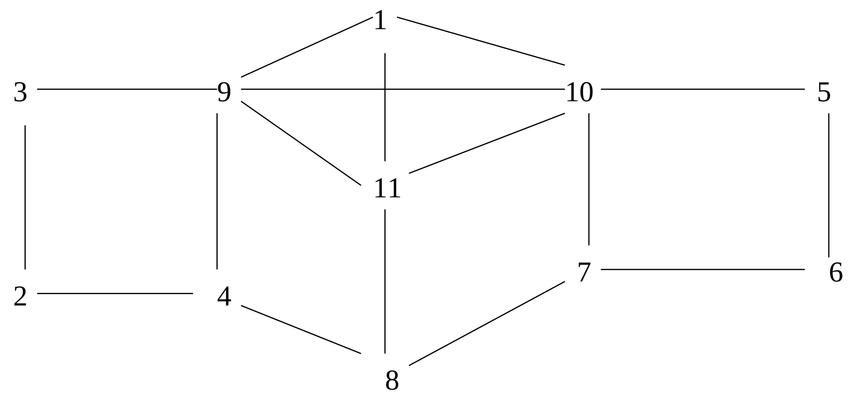


Figure 17: Sample Graph for Fill Edges

O(nm) time. Proofs of the correctness of the algorithm are contained in [163].

Clique separators are closely related to the idea of fill added to a matrix during Gaussian elimination. Given an ordering of the vertices, one computes fill by stepping through the vertices in order, and adding edges between all neighbors of the current vertex which occur after the vertex in the ordering.

Consider the graph of figure ?, taken from [163]. Fill edges for the ordering 1,2,3,...,11 are (3,4) (from 2), (8,9) (from 4), (6,10) (from 5), (8,10) (from 7). Tarjan's algorithm first computes an ordering which gives minimal fill, and uses this to find clique separators. Note the difference between minimal and minimum fill; finding an ordering with minimum fill is NP-complete [182].

The following algorithm, due to Rose, Tarjan and Lueker, finds a minimal fill ordering in O(nm) time; the proof of correcteness of this algorithm is in [140]. An alternative O(nm) algorithm for finding a minimal fill ordering is presented in [131]. The algorithm is related to the lexicographic breadth first search algorithm described in chapter ?; in that algorithm, each vertex v chosen divides sets into neighbors and nonnegighbors of the currently chosen vertex, placing neighbors after nonneighbors. The additional work in this algorithm comes from the fact that the algorithm is trying to divide sets into neighbors and nonneighbors of v not in the original graph, but in the filled graph; thus it computes the known fill involving v before dividing the set using a characterization of fill given in exercise ?.

Start with all vertices in a single set. Repeatedly remove a vertex v from the last set, and place it in the output list (the element chosen at step i is placed at position n-i+1 of the output list, as in lexicographic breadth first search). Let knownfilled(v) be the set of vertices w such that there is a path from v to w in G using only intermediate vertices which come from sets which precede w in the current ordering; note that all neighbors of v are included in this group, since no intermediate vertices are used. Divide every current set S into S - knownfilled(v), knownfilled(v) $\cap$ S, and place the former set before the latter. it is not difficult to compute this ordering in O(nm) time.

Consider running this procedure on the graph of figure ?. We can start by placing vertex 11 last; this creates the ordering (2,3,4,5,6,7)(1,8,9,10) 11. Suppose 10 is picked next; the path 10,5,6,7,8 uses intermediate vertices only in sets before 8, so the new ordering is (2,3,4,6)(5,7)(1,8,9) 10 11. If 9 is chosen next, we get (2,6)(3,4)(5,7)(1,8) 9 10 11; note that the path 9,3,2,4,8 causes 8 to be treated as a neighbor of 9. If 8 is chosen next, we get (2,6)(3)(4)(5)(7)(1) 8 9 10 11. The first vertex which distinguishes 2 from 6 is 7; we thus get the ordering 2 6 3 4 5 7 1 8 9 10 11. Edges of the fill from this ordering are (3,4) (caused by 2), (5,7) (from 6), (8,9) (from 4), (8,10) (from 7).

We now assume that L is a minimal fill ordering. For each v, let C(v) be the set of vertices w such that w comes after v in L, and (v,w) is either an edge of G or part of the fill caused by this ordering.

Let A be the connected component of the graph induced by V - C(v) which contains v, and let $B = V$ - (C(v) $\cup$ A). If C(v) is a clique and B is nonempty, then we have found a successful decomposition into $A \cup$ C(v) and $B \cup$ C(v).

Tarjan shows that if there is a clique separator of G, some clique separator is of the form C(v); this gives an O(nm) algorithm for finding a single clique cutset. He then makes several observations regarding further decomposition of the subgraphs which reduce the running time of finding the entire clique separator decomposition, even though finding a single separator has the same time bound as Whiteside's algorithm. These all come from the fact that the relative ordering of vertices produced by Tarjan's algorithm in each subgraph can be the same as the ordering in L, and the sets C(v_i) in the subgraphs are the same as the sets C(v_i) in G if this ordering is used.

The first observation which improves the time bound of clique separator decomposition using this algorithm is that after a successful decomposition step, the subgraph $A \cup$ C(v) is indecomposable [163]; in other words, the algorithm always chooses to separate a minimal set containing v. This implies that the number of atoms of a subgraph decomposed in this fashion is O(n); each decomposition step removes at least one vertex from the subgraph which needs to be decomposed further.

The second observation is that if the decomposition step defined by C(v) is unsuccessful in G, it cannot be successful in either subgraph formed by a successful decomposition step. Therefore, each vertex v is examined once to determine whether C(v) forms a clique separator, which means the entire algorithm runs in O(nm) time.

Thus, for the graph of figure ? and the ordering 1,2,3,..,11, we find that 9,10,11 separates 1 from the rest of the graph and is a clique, decomposing the graph into 1,9,10,112,3,4,5,6,7,8,9,10,11. C(2) = 3,4 is not a clique, but C(3) = 4,9 separates 3,2 from the rest of its component. The next clique C(v) is C(6) = 7,10, which splits 5 and 6 from its component. The remaining clique C(v) is C(8) = 9,10,11; this is not a clique separator since the component remains connected with the separator removed. Thus, the clique separator decomposition tree is shown in figure ?.

For the coloring, clique, independent set, and minimum fill problem, if there are efficient algorithms for atoms found by the clique separator decomposition, the problem can be solved efficiently for the entire graph. This decomposition is used in

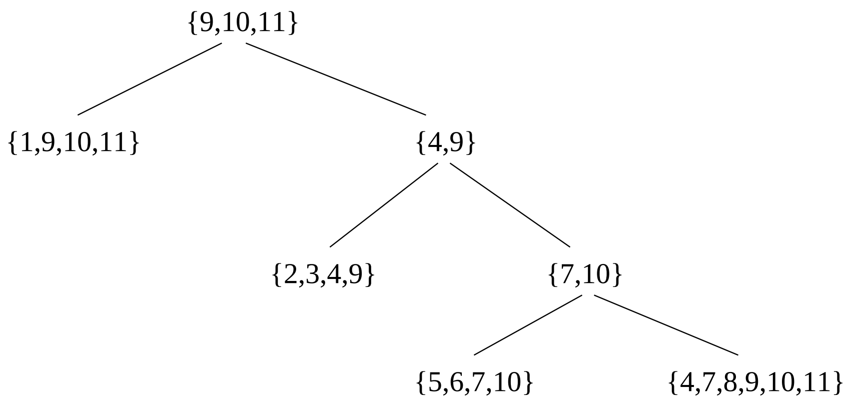


Figure 18: Decomposition Tree for the Graph of Figure ?

chapter ? for dealing with EPT graphs, and in chapter ? for recognizing path graphs.

Although even chordal graphs, which are completely decomposable with respect to clique separator decomposition, are too large to have an efficient representation, we can define a variant of this decomposition which could be interesting with respect to counting and representation. Define a ? clique separator C to be a clique separator such that the neighborhoods of vertices in C are ordered by set containment within each component. The fundamental idea is that while $\Theta(n^2)$ information must be maintained to reconstruct G from a standard clique separator and the graphs induced by the corresponding connected components, we need only O($n$log$n$) bits to reconstruct G from a ? clique separator.

amalgam, Sabiducci?

12.5 Exercises

1) Show that G is distance hereditary iff G - w is distance hereditary, for any degree 1 vertex w.

2) Show that every distance-hereditary graph is a circle graph.

3) Hammer and Maffray [] show that the house, the gem, the domino, and a chordless cycle of length greater than 4 give a forbidden subgraph characterization of distance-hereditary graphs. Show that each is a minimal non distance hereditary graph.

3a) Show that chordless cycles in G map to chordless cycles of length at least 6 in B(G).

3b) Show that k-suns map to $2k$ chordless cycles in B(G).

4) Show every distance hereditary graph is completely separable with respect to join decomposition.

5*) Show that there are graphs without any holes of size $> k$ which are not k-simplicially reducible.

6) Show that every permutation graph is completely decomposable with respect to the generalized join.

7) Let Bits(G) be the number of bits used to represent a graph G. Show that if G can be split into G_1, G_2 by the generalized join decomposition, Bits(G) is at most Bits(G_1) + Bits(G_2) + O($n$log$n$).

8) Consider a graph decomposition formed by successively removing vertices of degree at most k. Show that graphs which are completely decomposable using this decomposition neither contain nor are contained in the class of graphs with substitution decomposition diameter at most k.

9) Show that G is a cograph if and only if there is a pruning sequence with no pendant vertices.

10) Show that the elimination schemes for dually and doubly chordal graphs can be found in O(n+m) time. Note that the key point is that one can find the vertex adjacent to the maximum number of previously chosen vertices efficiently.

11) Show that neither LBFS nor MCS necessarily gives a maximum neighborhood ordering of a doubly chordal or dually chordal graph.

12) Show that O(n^2) clique separators are found if you recursively decompose a graph using the clique separator decomposition.

13) Show that there are $2^{\Theta(n^2)}$ doubly chordal graphs. Since every doubly chordal graph is dually chordal, the same result clearly holds for dually chordal graphs.

14) A graph is brittle if there is an elimination scheme such that each vertex removed is either not in the middle of a P_4 in the remaining graph, or not at the end of a P_4 in the remaining graph. Show that the number of brittle graphs is $2^{\Omega(n^2)}$, and that this remains true if we are only allowed to eliminate vertices which are not at the end of a P_4.

15) Show that trees are a recursively decomposable class.

16a) Show that there are classes of graphs which are recursively decomposable, and do not have bounded substitution decomposition diameter

16b) Show that there are classes of graphs which have bounded substitution decomposition diameter, but are not recursively decomposable.

17) Prove or disprove: that the class of graphs with substitution decomposition diameter k is hereditary.

18) Show that a single step of a k-terminal decomp can be found in polynomial time.

19) Show that for any i, you can determinine whether there is a decomposition of G into A, B, $A > 0$, $B > i-1$ such that each vertex in A has at most i neighbors in B in polynomial time.

20a) Show that if G has NLC-width k, G has cliquewidth at most $2k$.

20b) Show that if G has cliquewidth k, G has NLC-width at most k.

21) Show that graphs with NLC-width 1 are exactly equal to cographs (this result is also true for graphs with cliquewidth 2).

22) Show that every graph with cliquewidth k has a balanced k-module.

23) Show that we can find a balance k-module (if one exists) in a graph in polynomial time for any fixed k.

24) This problem gives the fundamental idea of Whitesides' algorithm for finding a clique separator in a graph.

Let S be a chordless cycle of length at least 4. Repeatedly find a connected component C of V - S; let R be the neighbors of C in S. If R is a clique, R is a clique cutset, since it separates S from C. Otherwise, take any nonadjacent x, y in R; let P be a chordless path from x to y which uses only vertices in C. Add vertices of P to S. Repeat unil S contains all vertices.

a) Show that if K is a clique cutset, then S is always in a single connected component of the graph induced by V - K.

b) Show that this gives an O(nm) algorithm for finding a clique cutset in G.

25) Show that a nonadjacent pair v_i,v_j is part of the fill caused by an ordering iff there is a path from v_i to v_j which uses only vertices which precede v_i and v_j in the ordering.

26) Show that you can determine whether there is a decomposition such that $K_{1,2}$ does not cross the partition in polynomial time.

27) Show that the modification of the maximum neighborhood ordering to reorder vertices which are numbered at the same time so that vertices which are marked earlier are placed later in the ordering can be accomplished in linear time.

13 Recognition Algorithms

This chapter covers efficient recognition algorithms for a number of classes of intersection graphs. Particular attention will be given to algorithms which have been considered too complex to cover in other texts. We will try to present the ideas of the algorithms, without covering details which might be necessary to implement the recognition algorithms completely within the best possible time bounds. We also present a collection of relevant classes which are NP-complete to recognize, although the NP-completeness proofs are not given here.

For a variety of reasons, recognition of some classes of graphs are covered in other chapters. Specifically, recognition algorithms for permutation graphs, comparability graphs, and two dimensional partial orders are presented in chapter ?, recognition of weakly triangulated graphs appears in chapter ?, algorithms for claw-free graphs, triangle-free graphs, and cographs are treated in chapter ?, distance hereditary graphs and strongly chordal graphs in chapter ?, and doubly and dually chordal graphs in chapter ?.

13.1 Chordal Graphs

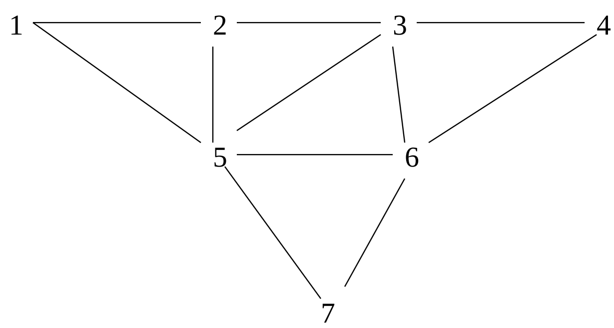
Recognition of chordal graphs is covered nicely in Golumbic [79]. We review this recognition algorithm briefly, since it is simple, and variants will be used throughout the book. The definition of chordal graphs implies a polynomial time recognition algorithm, since for any k we can determine whether G has a chordless cycle of length at least k in $\mathrm{O}(n^k)$ time. This would give an $\mathrm{O}(n^4)$ algorithm for chordal graph recognition; we will review algorithms which reduce the time bound to $\mathrm{O}(n+m)$.

Efficient chordal graph recognition algorithms rely on the following characterization of chordal graphs. A vertex v is simplicial if the neighbors of v form a clique. Note that a simplicial vertex can clearly not be part of any chordless cycle of length greater than 3. Therefore, if we want to determine whether a graph is chordal, we can remove any simplicial vertex we find; removal of some simplicial vertices may cause other vertices to become simplicial. It can be shown that if the graph is chordal, then this process of successively removing simplicial vertices will remove all vertices from the graph. This simple idea for testing chordality is formalized as follows.

A perfect elimination scheme $v_1v_2v_3...v_n$ is an ordering of the vertex set if and only if for all i, v_i is simplicial in the graph induced by v_{i+1} through v_n.

Theorem [51]: G is chordal if and only if there is a perfect elimination scheme for G.

See [79] for a proof of this theorem. It is not difficult (though not trivial either) to use this theorem to reduce the time needed for chordal graph recognition to $\mathrm{O}(n^3)$, but we can do much better by making more sophisticated use of the theorem. The algorithm is like the comparability graph recognition algorithm discussed in chapter ? in being usefully divided into two phases; a construction phase which creates an



(1234567) choose 7
(1234)(56)(7) choose 6
(12)(34)(5)(6)(7) choose 5
(12)(4)(3)(5)(6)(7) choose 3
(1)(2)(4)(3)(5)(6)(7)

Figure 19: Lexicographic Breadth First Search

ordering which is a perfect elimination scheme if and only if G is chordal, and a verification phase which checks whether the ordering which was constructed is in fact a perfect elimination scheme.

The construction phase is relatively simple, and quite a number of algorithms have been designed to construct a perfect elimination scheme for a chordal graph in linear time. We choose to describe lexicographic breadth first search (LBFS) for constructing the ordering, since this is also used in other algorithms of the book. The perfect elimination scheme is generated in reverse order.

Vertices are partitioned into sets; initially, all vertices are in the same set. At each step, an arbitrary vertex x from the last set is chosen. We remove x from the graph, and place it in the output list. Note that x comes before all previously selected vertices in the elimination scheme, and after all vertices which remain in the graph. All sets are subdivided into neighbors and nonneighbors of x, with neighbors of x in S placed immediately after the set of nonneighbors of x in S. An example is shown in figure ?. Standard data structures can be used to make this algorithm run in $O(n+m)$ time; for details, see [79, 140].

We now need to verify that the list generated is a perfect elimination scheme for G, since the algorithm clearly produces an ordering whether or not G is chordal. We now step through the perfect elimination scheme from first vertex to last; let v be the next vertex in the order. Find the first neighbor of v which comes after v in the order, and call this vertex w. Verify that w has edges to all neighbors of v which come after w in the ordering; if not, answer that this is not a perfect elimination scheme, since v is not simplicial. If the verification step works for all vertices, G is chordal and this is a perfect elimination scheme. Since the verification takes time proportional to $|N(v)|$ for each vertex v, this runs in $O(n+m)$ time. The correctness proof for this algorithm is left for exercise ?.

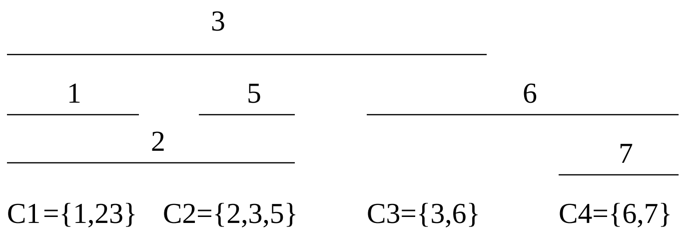


Figure 20: Building an Interval Model From a Consecutive Clique Arrangement

13.2 Interval Graphs

We will describe the standard algorithm for interval graph recognition due to Booth and Leuker [19] using PQ-trees. We note that this algorithm has been simplified in several ways, by using modified PQ-trees [108] and by dispensing with PQ-trees and complex data structures altogether. However, we will need to introduce PQ-trees in any case, since they will be used below as part of an efficient recognition algorithm for circular-arc graphs. We then describe an algorithm due to Corneil, Olariu, and Stewart [42], which recognizes interval graphs without using PQ-trees.

Booth and Leuker (independently???) designed an algorithm for recognizing interval graphs in $O(n+m)$ time as follows. As a first step, verify that G is chordal; if not, halt and answer that G is not an interval graph. If G is chordal, enumerate all maximal cliques of G; a discussion of this step is deferred to chapter ?, where maximal cliques are used as part of a chordal graph representation.

The number of maximal cliques in a chordal graph is at most n; this number can be exponential in general graphs. The total maximal clique size of G, defined by adding k to the sum for a maximal clique of size k, is $O(n+m)$. This allows processing vertex/clique incidence relationships in linear time. The interval representation is constructed by creating an ordering of the maximal cliques such that maximal cliques containing each vertex appear consecutively; the interval for v is created by simply spanning these cliques, as shown in figure ?.

For this reason, Booth and Leuker created a data structure called a PQ-tree, which is designed to enforce constraints of the type 'this set of objects must occur consecutively'. We will not give details of PQ-tree construction here; they are discussed in a number of texts such as [79] and [61]; the original paper [19] is also quite well written. The overall bound for enforcing a set of n constraints, where the total size of all sets constrained is m, is $O(n+m)$. The standard notion involves forcing all sets to be consecutive in a linear order, and from the discussion above gives a linear time algorithm for interval graph recognition. However, the data structure can also be used to force sets to be consecutive within a circular order; this will be used in the next section as part of a recognition algorithm for circular-arc graphs.

We now describe an interval graph recognition algorithm from [42]. Although the algorithm is simple, the correctness proof is nontrivial; the interested reader can consult that paper. I should mention that I use the terms earlier and later in the

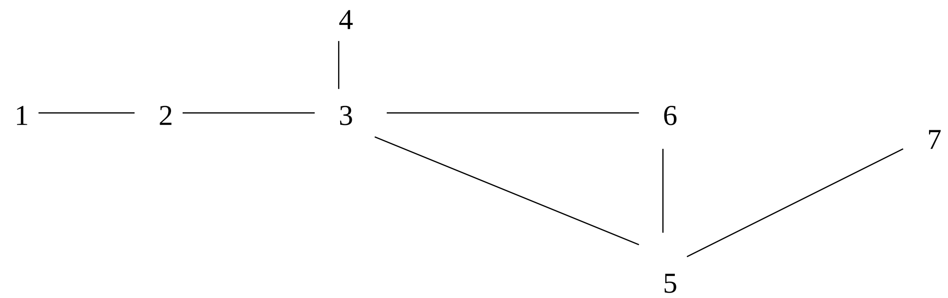


Figure 21: Sample Graph for Interval Graph Recognition

search differently from [42]; I say vertices occur later in the lexicographic search ordering if they were chosen for removal earlier in the algorithm; this usage comes from the idea that lexicographic breadth first search should also be producing a perfect elimination scheme.

The algorithm makes four passes of lexicographic breadth first search using certain tiebreaking rules to choose the appropriate vertex from the last set; the result is a valid ordering of the right endpoints in an interval model if G is an interval graph. To construct the intervals, we simply create an interval from each vertex v to the first neighbor of v in the ordering, and verify that each intermediate endpoint is a neighbor of v.

The algorithm uses different tiebreaking rules in the four passes of lexicographic breadth first search. Although most of the rules are simple to understand, the tiebreaking rule for the final pass is somewhat tricky. Therefore, we will describe the tiebreaking rule for the fourth pass at two levels. The first description is easy to understand, but hard to see how to implement in linear time, while the second description gives a linear time implementation of the search procedure.

Pass 1: Perform standard LBFS.

Pass 2: Perform LBFS, breaking ties by always choosing the vertex which occurs earliest in the ordering produced by Pass 1.

Pass 3: Perform LBFS, breaking ties by always choosing the vertex which occurs earliest in the ordering produced by pass 2.

Pass 4: Perform LBFS, breaking ties as follows. We always choose either the vertex a which occurs earliest in Pass 2 from the set, or the vertex b which occurs earliest in Pass 3 from the set. We choose b unless one of the following two conditions holds:
1) b is adjacent to some vertex which is in an earlier set of Pass 4 while a is adjacent to no such vertex.
2) Neither a nor b is adjacent to a vertex from an earlier set of Pass 4, b has a neighbor in its own set which is adjacent to a vertex in an earlier set of Pass 4, while a has no such neighbor.

Consider the graph of figure ?. The first pass, a standard LBFS, may produce the order 1 2 7 4 5 6 3. The second pass produces 7 6 5 4 3 2 1, and the third produces 1

2 4 3 6 5 7. Pass 4 works as follows; our initial set is (1,2,3,4,5,6,7) our two candidate are 7 and 1; the tiebreaking rule chooses 1 since there are no other sets. This gives us (3,4,5,6,7)(2) 1 and 2 must be chosen next. We now get (4,5,6,7) (3) 2 1. 3 is chosen, leaving (7)(4,5,6) 3 2 1. The next candidates are 6 and 4; neither has a neighbor in an earlier set, 6 has a neighbor in this set (5) which is adjacent to a vertex in an earlier set (7), while 4 has no such neighbor, so 4 is chosen, giving (7)(5,6) 4 3 2 1. Since 6 is before 5 in both pass 2 and pass 3, the order produced is 7 5 6 4 3 2 1.

It is easy to see how to implement the tiebreaking rules for passes 2 and 3 so that lexicographic breadth first search still runs in linear time. Sort all adjacency lists on the basis of order of vertices in the previous pass (see exercise ?), and the standard method of dividing sets into neighbors and nonneighbors will guarantee that the first vertex in the set will always be the earliest vertex from the previous pass. The same method is used to identify the vertices a and b in pass 4; we can keep two copies of the list, one in which vertices of a set are ordered according to position in Pass 2, and in the other vertices are ordered by their position in Pass 3. It is harder to see how to choose between a and b in Pass 4 in linear time. Therefore, we present an alternative description of pass 4, which is harder to visualize but simpler to implement in linear time.

Pass 4: Create lists A(v) (B(v)), where these are linked lists of neighbors of v which come after v in the orderings of pass 2 (pass 3), are adjacent to some vertex which comes before v in the ordering of pass 2 (pass 3), and have not yet been chosen in Pass 4. Perform LBFS, breaking ties as follows. Let a be the earliest vertex from the set in Pass 2, while b is the earliest vertex from the set in Pass 3. To choose between a and b, use the following rules in order until one succeeds.
1) If a precedes all its neighbors in pass 2, choose b
2) If b precedes all its neighbors in pass 3, choose a
3) If B(b) is empty or A(a) is nonempty, choose b
4) Let y be an arbitrary element of B(b). If the earliest neighbor of y in Pass 2 precedes the earliest neighbor of a in Pass 2, then choose b; otherwise, choose a.

It is not hard to see that pass 4 can be implemented to run in linear time. Calculating initial sets A and B for each vertex is easy; first calculate for every vertex w the earliest neighbor of w in the pass, and it is easy to check whether a neighbor w of v belongs on A(v) in constant time. When a vertex x is chosen in Pass 4, standard methods allow us to delete x from all A and B lists in O(N(x)) time; other steps are also easy to implement.

Therefore, interval graphs can be recognized in linear time, using standard data structures.

A number of graph classes are defined by whether the graph or its complement is a chordal and/or comparability graph; for example, interval graphs are equal to chordal cocomparability graphs. For some of these classes, it is useful to know whether algorithms are efficient in the complement as well. It is relatively easy to recognize co-chordal graphs in linear time; the construction and verification of a perfect elimination scheme in the complement can be done using edges of G (exercise

?). Co-interval graphs can also be recognized in linear time, though this result is less obvious. The first linear time recognition algorithm for cointerval graphs appears in ?.

An interval order is a partial order such that each element corresponds to an interval, with $x < y$ if the interval for x is completely before the interval for y. It is not hard to see that G is a cointerval graph if and only if G has a transitive orientation which is an interval order; in fact, G is a cointerval graph if and only if every transitive orientation of G is an interval order. A partial order is an interval order if and only if the neighborhoods of vertices are totally ordered by set containment [134] (exercise ?).

In chapter ?, we show that you can find a transitive orientation in linear time; we can thus test whether G is a cointerval graph in linear time by whether neighborhoods in the transitive orientation are ordered by set containment. (simpler trans or for this special case in ?)

13.3 Circular-Arc Recognition

Recognition of circular-arc graphs was a long standing open problem, first posed in 1964 [88]. The first polynomial algorithm for circular-arc graph recognition is due to Tucker in 1980 [168]. This algorithm has a reputation for being difficult to understand. In part, this is because the algorithm is complex, but the reputation of the algorithm has been exaggerated by a number of issues. Most importantly, Tucker himself talks about the need for simplification in the paper. [3] Some of the proofs are quite complex, even in portions where the algorithm is not, but it is hard for a reader to separate these issues since they are presented together in the paper. Since the time it was written, efficient algorithms for a number of other problems have led to modifications of Tucker's algorithm which are conceptually simpler, and improve the running time. The algorithm described here is due to Eschen and Spinrad [59] though large parts of it are due directly to Tucker.

Before going into circular-arc graph recognition, let us briefly discuss why we cannot use PQ-trees to solve the recognition problem for circular-arc graphs simply. For interval graphs, we enforced constraints for each maximal clique of G, using the fact that the total size of all constraints is $O(n+m)$. The clique structure of a circular-arc graph is much more complex than that of an interval graph; in fact, circular-arc graphs may have an exponential number of maximal cliques (exercise ?). Fundamentally, this is because cliques in circular-arc graphs do not necessarily share a common point on the circle (formally, arcs on the circle do not have the 'Helly property'); the reader can see this easily by laying out a triangle so that there is no

[3] Tucker's statements from the paper include 'The real difficulty in our algorithm is not with speed but its length' and 'As mentioned earlier, simplification rather than greater speed is what is really needed, since the constant in $O(n^3)$ is likely to be horrendous'. The author disagrees slightly with the latter statement; I do not believe the constant in the running time is such a problem, but it is very difficult for someone who has not studied the paper very carefully to produce a correct implementation of the algorithm.

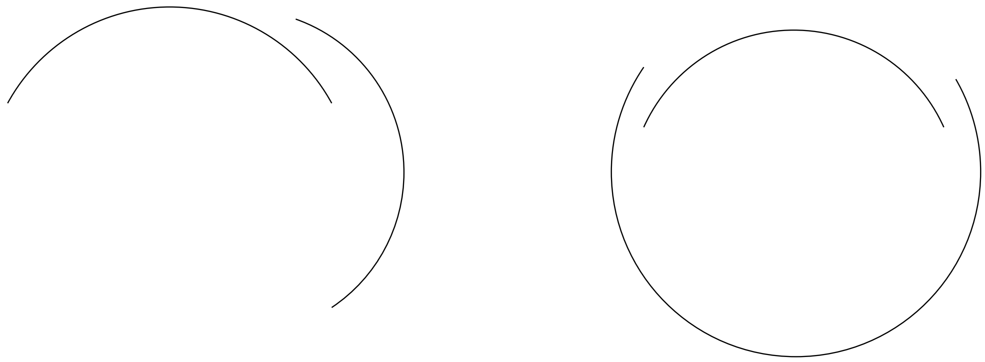


Figure 22: Overlap vs. Circle Covering

point in common. The tempting assumption that cliques of a circular-arc graph meet at a point has led to quite a number of incorrect submissions to technical journals; if it were true, many algorithms on the class would be much simpler.

The recognition algorithm has much of the structure of a partitioning algorithm, as discussed in chapter ?. The algorithm first finds an approximate layout of arcs on the circle, and uses partitioning to get a better approximation of the final layout. Finally, a reduction is made to permutation graph recognition.

A pair of intersecting arcs in a circular-arc graph can intersect in a number of different ways. We divide the neighbors of x, enforcing different types of arc intersections for each. Assume that G is preprocessed to eliminate neighbors with the same closed neighborhood; this preprocessing step can be performed in $\mathrm{O}(n+m)$ time. We will enforce the arc corresponding to x to contain the arc corresponding to y if $\mathrm{N}([y])$ is a subset of $\mathrm{N}([x])$. If neither neighborhood contains the other, we have two possibilities; the arcs may have an overlap relationship, as in figure ?-a, or may cover the circle, as in figure ?-b. We will enforce a circle-covering relationship between x and y if and only if every nonneighbor of x is contained in y. In exercise ?, it is shown that if G is a circular-arc graph, there is a model where all relationships are of the type we require. These relationships can easily be computed in $\mathrm{O}(n^3)$ time. A statement in Tucker's paper that this might be the key bottleneck step of the recognition algorithm inspired the design of our $\mathrm{O}(n^2)$ recognition algorithm.

If the graph has clique cover number 2 (case I as presented by Tucker [168]) we will solve the problem completely by reduction to permutation graph recognition, as we will see later. We note that the clique cover 2 case has been called the difficult portion of Tucker's original algorithm [181], and is one of the major reasons Tucker's algorithm is viewed as being very complex. By contrast, we will solve the clique cover 2 case very easily. To test whether G has clique cover 2, we see whether the complement is bipartite. If it is not, we either find a triangle in the complement or a longer odd hole; one of these can easily be found in linear time.

The recognition algorithm is then divided into several stages. In stage 1, an initial set of vertices is placed on the circle, dividing the circle into regions. In stage 2, other vertices have their endpoints placed in the appropriate region (or, more often, pair of regions). Partitioning is used in stage 3 to get the final refinement of the layout.

We will not present the case in which we find a longer odd hole in this book. It is not difficult to show that there is a unique method for placing the odd hole itself on the circle (stage 1). We then need to place other vertices with respect to the odd hole (stage 2). Placing other vertices with respect to the odd hole is similar to placement in the case when we find a triangle; for a complete description of the placement procedure in this case, see [59, 60]. The procedure for stage 3, finding the exact placement of endpoints, is the same whether we find a triangle or an odd hole.

Suppose that we find a triangle in the complement of G; this corresponds to an independent set of size 3 in G. We extend this to a maximal independent set I_1. We then slightly augment the independent set to get an independent set I which has the following properties: 1) there is no vertex v in I such that $\mathrm{N}(w)$ is a proper subset of $\mathrm{N}(v)$ for some w 2) We cannot get a bigger independent set by replacing some v in I with a pair of neighbors of v. Details of this augmentation step are the subject of problem ?.

The technique for finding a layout for the maximal independent set I was given by Tucker, though the description here is substantially different. The only data structure needed for solving this efficiently is a PQ-tree. A PQ-tree takes a set of items, and enforces constraints (either on the line or the circle) of the type 'these objects must occur consecutively'; each new constraint takes $\mathrm{O}(n)$ time. PQ-trees can be viewed as special cases of modular decomposition trees (see [101]); for details of an amortized linear time analysis in the case of interval graph recognition, see [19].

The idea for this step comes from finding sets of constraints which obviously must be enforced in a layout, and showing that this set of constraints are sufficient. Showing the sufficiency of these constraints is a difficult proof, and is given in Tucker [168]. However, we should not confuse the difficulty of the proof with the difficulty of the algorithm; the algorithm for this step is easy to understand.

One obvious set of constraints is that for each vertex v, the neighbors of v in I must occur consecutively on the circle. This is not quite enough to force an ordering on I, as shown in example ?. Clearly, i1 and i2 must be consecutive in the example, but there is no vertex adjacent to both of these. Thus, we add the following constraints, which it is easy to see are necessary. It is also not difficult to see that for any pair of adjacent vertices v and w, the neighbors of v and w in I must occur consecutively on the circle. Tucker shows that these simple necessary conditions are sufficient to enforce a correct ordering on the set I.

There are clearly at most n^2 such orderings to enforce; since PQ-trees enforce a constraint in $\mathrm{O}(n)$ time, stage 1 can be performed simply in $\mathrm{O}(n^3)$ time. This number of constraints can be reduced to $\mathrm{O}(n)$ by making the following observation. If v is a vertex, we only need to enforce the neighboring pair constraints that neighbors of v and w_1, v and w_2 must occur consecutively, where w_1 and w_2 vertices which have overlap relationship with v's arc, and whose arcs extend least far from v in the clockwise and counterclockwise directions respectively. These vertices can be found for v in $\mathrm{O}(n)$ time [59, 60]; the techniques are not difficult, but are at a lower level

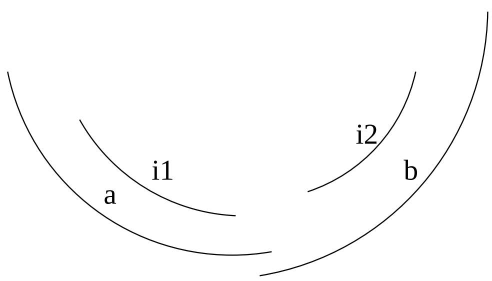


Figure 23: Constraints Imposed by Single Arcs are Insufficient

of detail than we wish to treat here.

We now enter stage 2, in which vertices are placed with respect to the placement of arcs in I. For many types of vertices, this placement is simple. Each v must be adjacent to a consecutive sequence of vertices in I; v's arc will properly contain all arcs that are not at the outer ends of this sequence. Suppose that v is adjacent to at least two vertices of I, and is nonadjacent to at least one vertex of I. We can determine whether v has an overlap or containment relationship with the two vertices i_c and i_{cc} on the clockwise and counterclockwise endpoints of the range by determining whether $\mathrm{N}(v)$ contains $\mathrm{N}(i_c)$ and $\mathrm{N}(i_{cc})$. These placements clearly take $\mathrm{O}(n)$ time per vertex.

The more complex cases involve placement of vertices which are adjacent to only one vertex of I, or are adjacent to every vertex of I. Consider a vertex v which is adjacent to a single vertex i of I; note that v must be adjacent to some vertex of I since I is a maximal independent set. We choose our independent set such that v's arc is not contained in i's arc. If arc v contains arc i, placement of v is easy. Suppose that v and i have an overlap relationship. One endpoint of v will be in the region inside of arc i. We cannot tell just by looking at relationships between v and vertices of I whether the other endpoint of v should be in the region between arc i-1 and i, or in the region between i and i+1. A similar difficult case arises for a vertex v which is adjacent to all vertices in I, and has an overlap relationship with exactly one vertex i; one endpoint is within i, and one either between i and i-1 or between i and i+1.

Let i be a member of I. We will give a procedure for placing the endpoints of vertices adjacent only to i from I, and endpoints of vertices which are adjacent to all of I and have an overlap relationship only with i. The only relevant vertices are those which can have endpoints within i and/or in one of the regions next to i. Our goal is to turn the placement problem on these vertices into a clique cover two problem.

One clique is formed by the set of vertices S adjacent only to i; there cannot be two nonadjacent vertices in this set, since we found a maximal independent set which could not be augmented by replacing a single vertex with two. However, the remaining vertices do not necessarily form a clique; vertices which approach i from the counterclockwise direction may be nonadjacent to vertices which approach i from the clockwise direction. We make all other vertices into a clique, but we add extra vertices to make sure that vertices approach from the correct direction. Think of

i on the bottom of the circle; we add vertices b,c,d, which can be thought of as clockwise, above, and counterclockwise from i respectively. Make a 4-cycle i, b, c, d; add adjacencies from S to b and d, from c to all vertices not in $S \cup i$ and from b and d to all vertices which approach i from the counterclockwise and clockwise direction respectively. The new graph has clique cover number two, and any placement of this graph can be used as a placement of endpoints in G. We solve the clique cover two problem using a reduction to two dimensional partial order recognition which will be given later.

Every vertex appears in at most four of the graphs used in the placement of endpoints near i, and each such problem will be solved in $O(n^2)$ time, yielding an $O(n^2)$ algorithm for the placement phase.

Stage 3 of the recognition algorithm involves taking our initial placement of arcs, and performing a partitioning procedure as described in chapter ? until we get no further refinements. Vertices are placed in the same initial set if their endpoints are placed in the same pair of regions at the end of stage 1. Note that this will place endpoints in the same set if they span the circle in opposite directions; a vertex which goes clockwise from region 4 to region 6 will be placed in the same initial set as a vertex which goes clockwise from region 6 to region 4 of the circle.

When vertices are placed or split into different sets, the relationship between at least one endpoint of each corresponding arc is fixed, and the necessary relationship between the other pair of endpoints can be determined. For example, if the clockwise endpoint of x is placed counterclockwise with respect to the counterclockwise endpoint of y on one region of the circle, then the other endpoint of x must be placed clockwise with respect to the other endpoint of y if and only if x and y are nonadjacent. Thus, whenever sets are split, we can determine and store in a matrix the necessary conditions on the other endpoints so that these can be accessed in constant time.

Suppose that we have more than one set of endpoints in a region. Take each vertex s from some set S; each one divides the region into a subregion clockwise of this endpoint of s, and counterclockwise of this endpoint of s. Vertices from other sets can be placed in the appropriate subregion in constant time, since we have stored relationships to this endpoint of s; each other set is split into a subset which has one endpoint in the region clockwise of S, and a set which has one endpoint in the region counterclockwise of s. Vertices of S are then grouped into subsets, depending on how many endpoints of the vertex lie clockwise in the region.

This procedure continues until all regions have exactly one set. Using the partitioning algorithm from chapter ?, we can use the fact that the relationship between each pair of endpoints is examined at most once to perform this partitioning in $O(n^2)$ time. Any set S with greater than one vertex after this step must have clique cover number two, and can be treated as a separate graph, since relationships between these endpoints and the rest of the graph has been completely determined.

We are now left with the possibility that the graph has clique cover number 2. We may assume that there is no pair of neighbors with identical closed neighborhoods, since these can use equivalent arcs and can be found easily by a preprocessing step.

We reduce this problem to a two dimensional partial order recognition problem as follows.

Let X, Y be any pair of cliques which covers the vertex set of G. Create a graph G' as follows. Vertices of G' correspond to vertices of G. For 2 vertices x_1, x_2 in X, add an edge from x_1 to x_2 if and only if the neighborhood of x_1 is contained in the neighborhood of x_2. For a pair of vertices x in X, y in Y, add an edge from x to y if and only if x and y are nonadjacent in G, and an edge from y to x if and only if x and y cover the circle. For a pair of vertices y_1, y_2 in Y, add an edge from y_1 to y_2 if and only if the neighborhood of y_1 contains the neighborhood of y_2. G is a circular-arc graph if and only if G' is a two dimensional partial order.

The reduction above is not difficult to understand if viewed in the following way. Tucker [168] showed that for any clique cover 2 circular-arc graph, there is a model in which some pair of points on the circle such that every arc passes through one of these points. The theorem is necessary because in general, cliques of a circular-arc graph do not have to meet at a point. Tucker's theorem can be modified easily to show that for any pair of cliques X, Y which partition the vertex set, there is a model in which all arcs corresponding to vertices in X pass through a point p of the circle, and all arcs corresponding to vertices in y pass through a point p'.

If we split the circle at points p and p', we get two orderings of the vertex set, since each arc appears on both semicircles. We interpret this as a representation of a two dimensional partial order by 'reading' each semicircle from p to p'. Thus, for example, if two vertices from X have an overlap relationship, one will come first in each list, and the vertices are nonadjacent in the two dimensional partial order. If arc x contains arc x', then x' will be encountered before x in both lists; on the other hand, if y in Y contains y' then the endpoints of y' will be encountered first on both semicircles. In the graph of figure ?, reading clockwise from p to p' we get the order 2 1 5 3 6 4, and reading counterclockwise from p to p' gives 3 6 1 4 2 5.

The reduction can easily be performed in $O(n^3)$ time. In order to reduce the time complexity, we need to be able to find the arc relationships (independent, overlap, containment, or circle-cover) between all pairs of neighbors in $O(n^2)$ time. We will describe how this step is performed; recall that this is only necessary if we want the fastest possible time bound.

Since independence in arcs corresponds to nonadjacency between vertices, our task is to distinguish whether pairs of neighbors have containment, overlap, or circle-covering relationships. We first find containment relationships; these correspond to pairs of neighbors x, y such that $N(x)$ - y is a subset of $N(y)$. Let v be a minimum degree vertex of G; v is chosen since no arc is contained in v, which means that the subgraph induced by $N(v)$ has clique cover number 2 (exercise ?). Clearly, the subgraph induced by V - $N(v)$ must be an interval graph; we call this subgraph I.

We compute neighborhood containments of vertices b and c using two subcomputations; we see if neighbors of b in I are a subset of neighbors of c in I, and whether neighbors of b in $N(v)$ are a subset of neighbors of c in $N(v)$. Clearly, neighborhood containment relationships for G are the intersection of these two restricted neighbor-

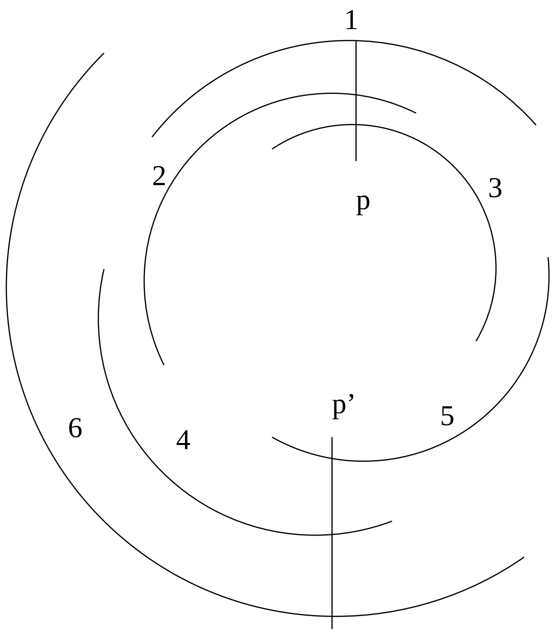


Figure 24: Sample Model for Reduction to Dimension 2 Posets

hood containment relationships.

Restricting neighborhoods to I allows us to use the interval model to compute relationships. Neighborhood containment within I can be computed in linear time, since I is an interval graph (exercise ?). Let i be in I, and let w be in $\mathrm{N}(v)$. We do not need to concern ourselves about whether $\mathrm{N}(w)$ is a subset of $\mathrm{N}(i)$ when restricted to I (although this can also be computed efficiently), since we know that $\mathrm{N}(w)$ cannot be a subset of $\mathrm{N}(i)$ in G. We can find all i such that $\mathrm{N}(i)$ is a subset of $\mathrm{N}(w)$ when restricted to I in $\mathrm{O}(n)$ time by stepping left to right through an interval model for I, keeping track at each point of the last interval i' seen which was nonadjacent to w. When we reach the right endpoint of i, $\mathrm{N}(i)$ is a subset of $\mathrm{N}(w)$ if and only if i is nonadjacent to i'.

The key to efficient algorithms for neighborhood containment when neighborhoods are restricted to $\mathrm{N}(v)$ is to reduce the problem to computing neighborhood containment on a chordal bipartite graph. Neighborhood containments in chordal bipartite graphs can be computed in $\mathrm{O}(n^2)$ time, as described in chapter ?.

Construct a bipartite graph D as follows. Make a copy $N'(v)$ of one color class, and V the other. Add an edge from w' in $N'(v)$ to x in V if and only if x and w are nonadjacent in G, and x is not equal to w. D is chordal bipartite (exercise ?). This allows us to compute all neighborhood containments in D in $\mathrm{O}(n^2)$ time, as discussed in chapter ?. Thus, we can compute neighborhood containment restricted to $\mathrm{N}(v)$ for neighbors b and c in G by determining whether $\mathrm{N}(b)$ is contained in $\mathrm{N}(c)$ in D. Thus, all neighborhood containment relationships in a circular-arc graph can be found in $\mathrm{O}(n^2)$ time.

We next need to distinguish pairs of neighbors which have circle-covering relationships from those which have overlap relations. This process is simplified by the fact that this information is only needed by the algorithm on subgraphs of G which have clique cover number two. Let A, B be the color classes of a circular-arc graph G with clique cover number two, and assume that we have already computed all neighborhood containment relationships.

Create a bipartite graph H on color classes $A \cup B$ and B', where B' is a copy of B. Add an edge (a,b') for a in A, b' in B' if and only if a and b are nonadjacent in G. Add an edge $(b_1,b_{2'})$ for b_1, b_2 in B if and only if b is not equal to b' and $\mathrm{N}(b_2)$ is contained in $\mathrm{N}(b_1) \cup b_1$; H can be constructed in $\mathrm{O}(n^2)$ time if we know neighborhood containment relationships between neighbors in G.

H can be shown to be chordal bipartite [59], which allows all neighborhood containment relationships in H to be computed in $\mathrm{O}(n^2)$ time. Two arcs from the same clique cannot cover the circle, unless one of the corresponding vertices is adjacent to every vertex in G; such vertices can be removed as a preprocessing step. Consider a pair of vertices a in A, b in B. Then a and b cover the circle if and only if for every x in $\mathrm{N}(b)$ - $\mathrm{N}(a)$, arc x is contained in arc b; this corresponds exactly to $\mathrm{N}(a)$ being a subset of $\mathrm{N}(b)$ in H. Thus, we can distinguish circle-covering pairs from overlapping pairs in a clique cover number two circular-arc graph in $\mathrm{O}(n^2)$ time.

Together, these techniques give an algorithm for recognizing circular-arc graphs. As far as I know, Tucker's algorithm has never been written up as a program, and was generally viewed as extraordinarily complex. Hopefully, this section has made the algorithm seem less intimidating rather than more so. I feel that the algorithm is not so difficult to understand conceptually, and would not be difficult to program if you are satisfied with an $\mathrm{O}(n^3)$ implementation, but I may be too close to the problem to view this objectively.

13.4 Restrictions on Intervals and Arcs

A number of natural restrictions of interval and circular-arc graphs have received attention. The most widely studied are requiring intervals or arcs to have the same length, and requiring that no interval or arc properly contains the other. The first restriction defines the unit interval and unit circular-arc graphs, while the second gives rise to proper interval and proper circular-arc graphs. We also deal briefly with Helly circular-arc graphs; in these, cliques are required to share a common point on the circle.

13.4.1 Unit and Proper Interval Graphs

The classes of unit and proper interval graphs are identical [137]. We will show a constructive proof of this equivalence below; another useful characterization is that G is a proper/unit interval graph if and only if G is an interval graph, and G contains no $K_{1,3}$ [137]. In chapter ?, we present a simple algorithm which recognizes unit interval graphs from scratch. We note below that any linear time algorithm which constructs an interval model can be modified to recognize unit interval graphs in linear time.

Given an interval representation of G, it is very easy to construct a consecutive clique arrangement for G (exercise ?). Assume that we have a consecutive clique arrangement for an interval graph G. It is easy to get a proper interval model for G, if such a model exists; if two intervals share a common leftmost (rightmost) clique, order their left endpoints in the reverse order of their corresponding rightmost (leftmost) cliques. We can test whether the model is a proper interval model in O(n) time, by checking whether the ordering of left endpoints is the same as the ordering of right endpoints in this model. If the ordering and left and right endpoints is the same, it is not difficult conceptually to construct a unit interval model. Vertices from the leftmost maximal clique are placed with left endpoints in the same order, very close together. Other vertices added from left to right. Each vertex v is placed just after the rightmost nonneighbor x of v, and any left endpoints to the right of x. Since left and right endpoints occur in the same order, the neighbors of v are exactly the vertices which occur after x, and if all intervals have unit length then v extends past all these vertices. Some other issues arising from this construction process are raised in chapter ?.

13.4.2 Proper Circular-Arc Graphs

The equivalence between unit and proper models breaks down for circular-arc graphs; it is easy to see that the graph of figure ?-a, shown with a proper circular-arc model, is not a unit circular-arc graph. We will give an algorithm for constructing a proper circular-arc model for G, if such a model exists, and show how to test whether this can be transformed into a unit circular-arc graph.

Note: the following discussion should be omitted/changed in the final version, but it is interesting enough for me to say something about it.

Proper circular-arc graphs can be recognized in O(n+m) time, as we will show below. The history of this problem is quite confusing, coming from the fact that the algorithms rely on a sequence of papers which spread over more than 30 years. The net result is that while no authors make a mathematical error, I believe that the published linear time bounds do not stand up, as authors misinterpret statements which appear in previous papers.

The first algorithm to solve this problem is due to Tucker in 1971 [169]. The algorithm comes from the following mathematical characterization of proper circular-arc graphs which forms the core of the paper.

Tucker's algorithm divides the problem into two cases, depending on whether the input graph has clique cover number two or not. If G does not have clique cover number two, Tucker shows that G is a proper circular-arc graph if and only if the augmented adjacency matrix of G has the circular 1s property for columns. He invokes an algorithm due to Fulkerson and Gross from 1965 [68] for solving the circular 1s problem. The clique cover 2 case is not covered in full detail in the paper; we will show an efficient algorithm for handling the clique cover two case below.

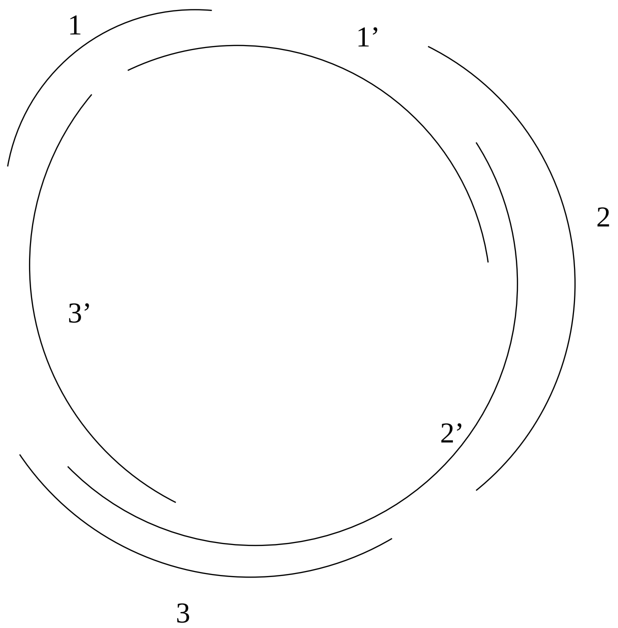


Figure 25: A Proper Circular Arc Graph which is not Unit Circular Arc

Tucker does not give an explicit time bound for the algorithm. A later paper quotes a time bound of $O(n^2)$ for Tucker's algorithm.

[49] gives an algorithm which solves the problem in linear time for sparse graphs, and uses Tucker's algorithm for the case when the graph is sufficiently dense, thus getting a linear time bound, assuming that Tucker's algorithm is $O(n^2)$.

Unfortunately, there is a slight problem with the claim of $O(n^2)$ running time for Tucker's algorithm. This comes from a statement in Fulkerson and Gross which is mathematically correct, but which is quite misleading for more recent readers (get exact quote). They give their time bound as $O(n^2)$, if taking the dot product of 2 n-vectors is viewed as a constant time operation. If the running time comes from this statement, we must give the time complexity as $O(n^3)$ rather than $O(n^2)$.

However, the linear time bound can be derived in another way. PQ-trees, which were developed after Tucker's paper was written, can be used to solve the circular 1s property for a matrix in linear time. Thus we only need to find an $O(n^2)$ algorithm for the clique cover two case to get a linear algorithm for proper circular-arc graph recognition.

If G has clique cover number two, we use an easier form of the reduction from general circular-arc graph recognition to permutation graph recognition. If G is a proper circular-arc graph, we can assume that all adjacent vertices are overlapping; containment relationships are not allowed in the model, and covering the circle is not necessary (exercise ?). This implies that the target permutation graph can be constructed very easily; it is simply the complement of the input graph. It also means that the resulting graph is bipartite. Linear time algorithms for recognizing bipartite permutation graphs are much easier than those for recognizing general permutation

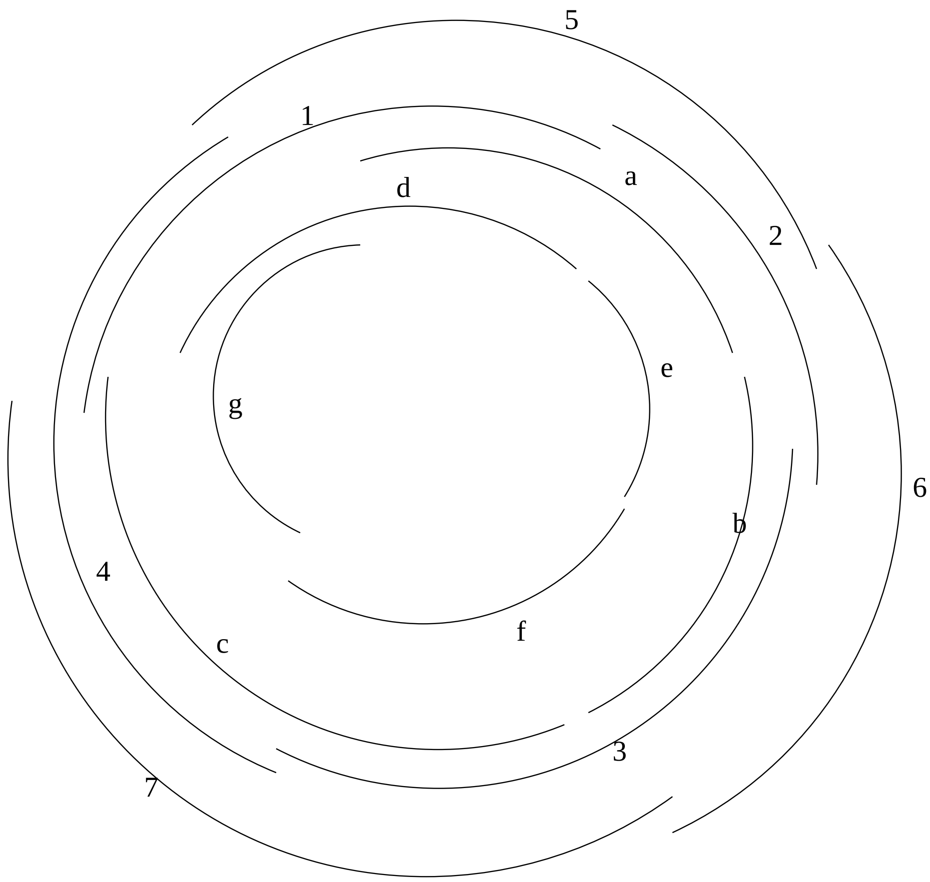


Figure 26: More Complex Proper but not Unit Circular-Arc Graph

graphs [154, 160]. Since we are allowed to use $O(n^2)$ time in this case, the problem is even simpler; we give one such algorithm for the sake of completeness. In a connected bipartite permutation graph, the only modules are pairs of vertices with identical neighborhoods; these are easy to identify. We can skip the modular decomposition step, and use the remainder of the $O(n^2)$ partitioning algorithm for permutation graph recognition from chapter ? to solve this problem.

13.4.3 Unit and Helly Circular-Arc Graphs

Although no paper has explicitly given a polynomial time bound for recognizing unit circular-arc graphs, an efficient algorithm is implicit in a paper of Tucker's [170].

If you are asked to provide an example of a graph which is a proper circular-arc graph but not a unit circular-arc graph, you will almost certainly come up with an example such as figure ?, in which some clique has the same size as an independent set. It is tempting, but not correct, to conjecture that every proper circular-arc graph is a unit circular-arc graph unless this occurs. A counterexample is given in figure ?.

We can see that this is not a unit circular-arc graph. The circular set of vertices

1,2,3,4,5,6,7 are consecutively adjacent, each i is nonadjacent to $x-2$, and wrap around the circle 2 times. This implies that the circumference of the circle is less than 7/2 if every arc has unit length. The set of vertices a,b,c,d,e,f,g are consecutively nonadjacent (must add conditions) and wrap around the circle 2 times. This implies that in any unit interval model, the circumference must have size at least 7/2.

More generally, an (n,k) circuit of a proper circular-arc graph is a set of n vertices such that vertices i and i+1 are adjacent, arc i+1 is clockwise of arc i+1, and the set of arcs wraps around the circle k times. An (m,l) independent set is a set of m vertices such that vertices i and i+1 are nonadjacent, arc i+1 is the first arc of the set which starts clockwise of the counterclockwise endpoint of arc i, and the arcs wrap around the circle l times.

Tucker [170] was able to show that G is a unit circular-arc graph if and only if an arbitrary proper circular-arc model (without a circle-covering pair) does not contain an (n,k) circuit and an (m,l) independent set such that $n/k \leq m/l$. In fact, Tucker proved stronger results; for example, if G is not unit circular-arc, there must be both an (n/k)-circuit and an (n,k) independent set. Thus, we will describe a method for searching for an obstruction of this form.

We look for upper and lower bounds on the circumference in greedy fashion, starting with every arc. Each search can be implemented to run in O($n$loglog$n$) time, so that the entire algorithm runs in O(n^2loglog$n$) time if we start with the proper interval model. Let x be any vertex; we show how to find the lower bound starting with x.

Maintain a circularly ordered list of unused arcs, the number of arcs used thus far, and the number of times you have traversed the circle. Let c be your current arc. Find the next unused arc y which starts after c; replace c with y, and continue as long as some arc is nonadjacent to c. If y is adjacent to x, add 1 to the number of times you have traversed the circle. At each point, calculate the current ratio of number of arcs used to the number of times you have traversed the circle; keep the maximum value of this ratio over all starting arcs x as your lower bound on the circumference. To calculate an upper bound, choose the unused neighbor which extends furthest at each step rather than the next unused nonneighbor.

The time complexity of the search from x is O(loglog$n$), if we use the data structure of von Emde Boas [57] which allows for searching for predecessors and successors of elements in the range 1..O(n) time per operation, with insertions and deletions allowed. Therefore, we can recognize unit circular-arc graphs in linear time; I imagine that this bound can be improved. For issues which arise if we want to construct a unit circular-arc model, see chapter ?.

For some optimization problems, Helly circular-arc graphs are simpler than general circular-arc graphs [73]. Again, no paper has been written on the recognition problem for Helly circular-arc graphs, but these can certainly be recognized in polynomial time. For Helly circular-arc graphs, all cliques meet at a point (by definition), which means that there are O(n) maximal cliques. We can therefore use PQ-trees to enforce a correct circular-ordering on the maximal cliques; cliques containing each vertex v

must appear consecutively on the circle. The bottleneck step is enumerating the maximal cliques, which takes ? time for general graphs and ? time for circular-arc graphs. I feel certain that some enterprising reader will improve this algorithm, either by an improved algorithm for enumerating maximal cliques for this special case or by a direct solution of the problem.

13.5 Trapezoid Graphs and Related Classes

In this section, we present an algorithm which solves a number of closely related recognition problems. The problems solved include trapezoid graph recognition, Ferrers dimension 2 digraph recognition, biorder dimension 2 and interval dimension 2 recognition, 2-chain subgraph cover, and threshold dimension 2 for split graphs. This section is called trapezoid graph recognition, since trapezoid graphs are most directly linked to the other classes treated in this chapter. The problems are all solved by reduction to the 2-chain cover problem; we then present an algorithm for the two chain cover problem.

The history of this problem is remarkably confusing. This comes from the different names and applications of essentially the same problem; authors would know relations between some pairs of graph classes, but not know about relationships to others. Another source of confusion is that two crucial papers by the same author were written on recognizing this class in polynomial time, and since the papers were presented without explicit time bounds other authors cited the simpler (but slower) algorithm rather than the faster algorithm. Improvements were then made on the time bounds in the slower algorithm, and at least three interesting new algorithms were developed which were faster than the algorithm of one paper, but were not in fact as fast as an algorithm implied by the first paper. Finally, the algorithm of the first paper was modified to give an algorithm which was both faster and newer than previous algorithms; we will present the modified algorithm here.

13.5.1 Relationships Between Problems

We outline the close relationships between a number of problems which have at times been studied independently. Some of these problems can be thought of as different views of the same underlying problem, and reductions in both directions can be found in [118]. For our purposes, it will be sufficient to reduce these problems to a single problem, called the 2 chain subgraph cover problem, which we will solve efficiently in a later section.

A chain graph is a bipartite graph in which neighborhoods of all vertices from a single color class can be ordered by set containment; in other words, the bipartite graph contains no induced $2K_2$. Chain graphs were used by Yannakakis to show that the partial order dimension problem and the minimum fill-in problem were NP-complete [183, 182]; these were two of the original open problems posed in Garey and Johnson's book [71].

The bipartite complement of a graph is the bipartite graph on the same vertex set such that two vertices from different color classes are adjacent if and only if these vertices are nonadjacent in G. Clearly, the bipartite complement of a chain graph is itself a chain graph.

The 2 chain subgraph cover problem asks whether there is a pair G_1, G_2 of subgraphs of a bipartite graph G such that G_1 and G_2 are subgraphs of G, and every edge of G is contained in at least one of G_1 or G_2.

The chain dimension of a bipartite graph G is the minimum number of chain graphs which give G as their intersection.

It is easy to see that G can be covered by 2 chain graphs if and only if the bipartite complement of G has chain dimension 2; any graph used in one problem can be replaced by the bipartite complement in the other problem. Some of the problems in this chapter reduce naturally to chain cover 2, while others are more easily viewed as reductions to chain dimension 2, often in the bipartite complement.

A Ferrers digraph is a digraph such that for any x, y, either $\mathrm{N}(x)$ is a subset of $\mathrm{N}(y)$ or $\mathrm{N}(y)$ is a subset of $\mathrm{N}(x)$. The Ferrers dimension of a digraph G is the smallest number of Ferrers digraphs on the same vertex set such that G is exactly equal to the intersection of this set of Ferrers digraphs.

There is a natural correspondence betweens Ferrers graphs and chain graphs, since both correspond to neighborhoods being ordered by set containment. To convert a Ferrers dimension k problem to a chain dimension problem, make a bipartite graph $\mathrm{B}(G)$ by making two copies V and V' of the vertices, and add an edge from v to w' if and only if $v \to w$ in G. The chain dimension of $\mathrm{B}(G)$ is equal to the Ferrers dimension of G.

The interval dimension of a poset P is the minimum number of interval orders which intersect to give P. The connection to chain graphs comes from the characterization that interval orders are exactly those partial orders such that the neighbors of each vertex are ordered by set containment. Thus, interval dimension on P can be reduced to chain dimension on $\mathrm{B}(P)$ as defined above, since there is a direct correspondence between chain graphs which contain $\mathrm{B}(P)$ and interval orders which contain P.

Trapezoid graphs, also called II graphs, are the intersection graphs of trapezoids defined by intervals defined along two parallel lines; for variants of these, see chapter ?. It is easy to see that the complement of a trapezoid graph must be orientable to be an interval dimension 2 poset. Since interval dimension is a comparability graph invariant [86], G is a trapezoid graph if and only if the complement of G has interval dimension 2. Thus, we can take a transitive orientation algorithm for the complement of G as described in chapter ?, and turn the trapezoid graph recognition problem into a chain cover problem (by way of interval dimension 2) in $\mathrm{O}(n^2)$ time.

We note that for recognizing partial orders with interval dimension 2 and trapezoid graphs, we do not want to have to check that the input is a comparability or cocomparability graph; recall that the transitive orientation problem will find some orientation for a graph even if the graph has no transitive orientation. However, our algorithm for the chain cover two problem also allows us to construct the chain graphs which cover G, which are easily translated into a pair of interval orders which represent P or a trapezoidal representation of G. After these are constructed, we verify in linear time whether the representations are correct; this is not a bottleneck step (exercise ?). Thus, our reductions allow recognition of interval dimension 2 posets and trapezoid graphs in $O(n^2)$ time, and can construct the corresponding models in the same time bound.

Reductions from the bidimension problem and threshold dimension 2 on split graphs to chain dimension can be found in [118].

13.5.2 Cogis' Reduction

Cogis was interested in the Ferrers dimension k problem, both for general values and for the special case $k = 2$. One paper [34] provided a simple reduction from Ferrers dimension k problems to poset dimension k recognition as follows.

For any digraph G, create a poset $P(G)$ as follows; elements of P correspond to vertices of G. Add an edge from x to y whenever $x \rightarrow y$ in G, or $N(x)$ a subset of $N(y)$ in G.

Theorem [34]: The Ferrers dimension of G is equal to the dimension of $P(G)$.

For the special case of Ferrer's dimension 2, this gives a polynomial time recognition algorithm. The time complexity is dominated by the cost of the reduction, though this was not true when Cogis wrote his paper, since the most efficient poset dimension 2 recognition algorithms had not been developed. The dominant step is computing neighborhood containment, which takes time proportional to matrix multiplication. If a simple algorithm is preferred, there are easy graph theoretic algorithms for all steps which take $O(n^3)$ time.

Unfortunately for the tangled history of the problem, Cogis also was the author of another paper which recognized digraphs with Ferrers dimension 2 in polynomial time [33], which became widely known as Cogis' algorithm. Cogis showed that one could construct a graph $H(D)$ with vertices corresponding to 0 entries in the original digraph, and connect vertices in $H(D)$ if and only if the vertices ??? in D (x,y induce a 2 by 2 permutation matrix?), and that $H(D)$ is bipartite if and only if D has Ferrers dimension 2. This simple algorithm has time complexity $O(n^4)$, and is closely related to an alternative solution given in terms of whether a split graph has threshold dimension 2 [102].

13.5.3 Reducing the Time Complexity

We present a different reduction to the partial order dimension problem which leads to an $O(n^2)$ algorithm for the chain cover two problem. Proofs which are not given here can be found in [118].

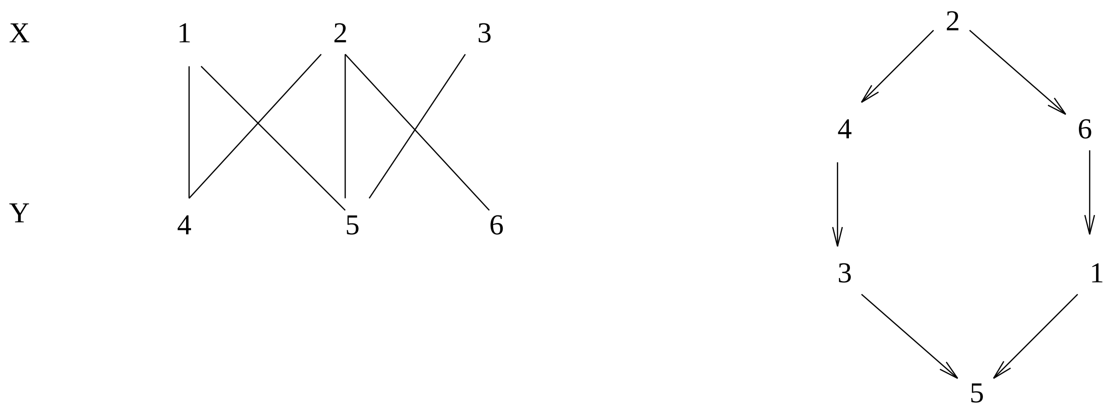


Figure 27: Graph and Transitive Reduction of Poset for Chain Cover

The first idea behind the new reduction is that we do not need to concern ourselves with the time for computing a general reduction from k-chain cover to partial order dimension k recognition; time complexities are only important for the special case $k = 2$. We then tailor our reduction so that the bottleneck step of neighborhood containment can be performed on a chordal bipartite graph rather than a general bipartite graph; using results from chapter ?, neighborhood containment on a chordal bipartite graph can be performed in $O(n^2)$ time.

Our reduction from the chain cover two to two dimensional poset recognition works as follows. Let G be a bipartite graph, and let X and Y be the color classes of G. We first remove all vertices with identical neighborhoods; we then add edges to P as follows. Add an edge from x_1 to x_2, where x_1 and x_2 are in X, if $N(x_1)$ contains $N(x_2)$. Add an edge from y_1 in Y to y_2 in Y if $N(y_1)$ is contained in $N(y_2)$. Add an edge from y in Y to x in X if x and y are nonadjacent. After these are done, we add an edge from x in X to y in Y if all neighbors of y in X are also neighbors of x. An example of this reduction is given in figure ?.

In general [118] shows that the dimension of the poset constructed is equal to the number of chain subgraphs needed to cover G. We are only interested in the chain cover two problem; since two dimensional partial orders can be recognized in linear time, the only step which can take more than $\Theta(n^2)$ time is the reduction itself.

Suppose that the bipartite complement of the input graph G is not chordal bipartite. Then G must have chain cover number at least 3, since putting the bipartite complement of a cycle C_k of length at least 6 through the reduction above produces C_k, and two dimensional partial orders cannot contain C_k for $k > 4$.

Neighborhood containment in G can be derived trivially from neighborhood containment in the bipartite complement. Suppose that we want to test whether G has chain cover number two. We first test in $O(n^2)$ time whether the bipartite complement is chordal bipartite; if it does not, we answer that G does not have chain cover number two. Otherwise, we perform the reduction above. Steps involving computing neighborhood containment in G take $O(n^2)$ time, since they can be computed on the bipartite complement. The only other part of the reduction which could be a problem is the last step, in which edges are added from x to y if all neighbors of y in X are also neighbors of X. However, the bipartite graph formed by making a copy of X and Y into one color class, a second copy X' of X as another color class, and adding an edge from v to x' if and only if an edge has been added in the poset from v to x' is also chordal bipartite [118]; this allows the final step of the reduction to be computed in $O(n^2)$ time as well.

13.5.4 Historical Note

The reader may note a similarity between the fast algorithm for circular-arc recognition and the algorithm for trapezoid graph recognition. Both use chordal bipartite graphs to avoid neighborhood containment calculation in a reduction to the poset dimension two problem. At one point, I thought that this was a coincidence, illustrating the usefulness of chordal bipartite graphs rather than showing a direct relationship between these problems. However, Trotter and Moore [166] showed that a graph with clique cover number two is a circular-arc graph if and only if the complement has interval dimension two; this shows a direct relationship between the problems. For this book, I decided not to use the reduction to interval dimension 2 recognition as part of the circular-arc recognition algorithm, so that a reader interested in implementing a circular-arc graph recognition algorithm would only need a single companion technical paper, but that reduction can certainly be used as well.

The paper showing the relationship between circular-arc graphs and interval dimension was written in 1976. Given my longstanding interest in circular-arc graphs and general admiration of Trotter's work, I read that paper at some time; I even cite the paper in [155]. At that time, I would have considered interval dimension two to be a less tractable problem than circular-arc recognition. By the time I became involved in developing recognition algorithms for circular-arc graphs and interval dimension 2 partial orders, I had completely forgotten Trotter and Moore's paper. Thus, I am in no position to criticize other authors who have been unaware of the complete history of this topic!

13.6 Circle Graphs

This section provides an algorithm for recognizing circle graphs, which are the intersection graphs of chords in the circle. Circle graphs have also been called alternance graphs [22] and overlap graphs; the latter comes from a characterization studied in problem ?.

Circle graph recognition received a good deal of attention after being raised as an open problem by Golumbic [79]. The first polynomial algorithm, due to Naji, runs in $O(n^9)$ time, as described by Bouchet [22]; this was successively improved to $O(n^5)$ by Bouchet [22], Gabor, Hsu, and Supowit to $O(n^3)$ [69] and Ma and Spinrad [120, 158] to $O(n^2)$. We will not present the fastest known algorithm here, since the algorithm is quite complex. We present a conceptually simpler algorithm due to Bouchet [22], which runs in $O(n^4)$ time when one takes into account improvements improvements in certain algorithms which were used as subroutines by Bouchet.

13.6.1 Join Decomposition

Recognition of circle graphs is closely tied to a form of graph decomposition known as join decomposition. The correspondence between the decomposition and recognition of circle graphs is very similar to the correspondence between substitution decomposition and comparability graph recognition or permutation graph recognition discussed in chapter ?. Indecomposable graphs have a unique representation, and a graph is a circle graph if and only if each component of the decomposition is a circle graph. The relationship between join decomposition and circle graph recognition was discovered independently by [69] and [22].

The join decomposition was originally intended to be a tool for dealing with directed graphs [46, 45], but the most important applications have proved to be on indirected graphs. A split of a graph is a partition of the vertex set into V_1, V_2 with the following properties. Let V_{1io} be the set of vertices in V_1 which are adjacent to at least one vertex in V_2, and let V_{2io} be the set of vertices of V_2 which are adjacent to at least one vertex in V_1.

1) $|V_1|$ and $|V_2|$ are at least 2.

2) Every vertex of V_{1io} must be adjacent to every vertex of V_{2io}.

In other words, the set of edges which go between V_1 and V_2 must form a complete bipartite graph. The join decomposition takes any split of G, replaces G by the subgraphs induced by $V_1 \cup m$ and $V_2 \cup m$, where m is a marker vertex adjacent exactly to vertices in V_{1io} and V_{2io}, and decomposes these graphs recursively. Using this marker vertex, it is easy to reconstruct G from the components after any split. An example of a step of this decomposition is given in figure ?. [45] gives uniqueness theorems for the final components of the join decomposition.

It is not hard to see that G is a circle graph if and only if each component of the join decomposition is a circle graph. Each component is an induced subgraph of the original graph, so the representation of the entire circle graph gives a representation of each component. Suppose that there are circle graph representations of two components formed by a split with marker m. These representations will be called A,m,B,m and X,m,Y,m, where A, B, X, and Y are orderings of vertices on the semicircles formed by removing m in the representations. Then the entire graph can be represented by A,X,m,B,Y,m, since orderings of pairs of vertices from a single component are preserved, and chords from different components cross if and only if both chords cross m. Circle graph representations of the two subgraphs of figure ? and the corresponding representation of the entire graph are shown in figure ?.

[69] and [22] show that if there is no split of a circle graph G, then G has a unique representation (ignoring reversal) on the circle. This will be crucial to the algorithm described below, but the proof will be omitted here in order to focus attention on

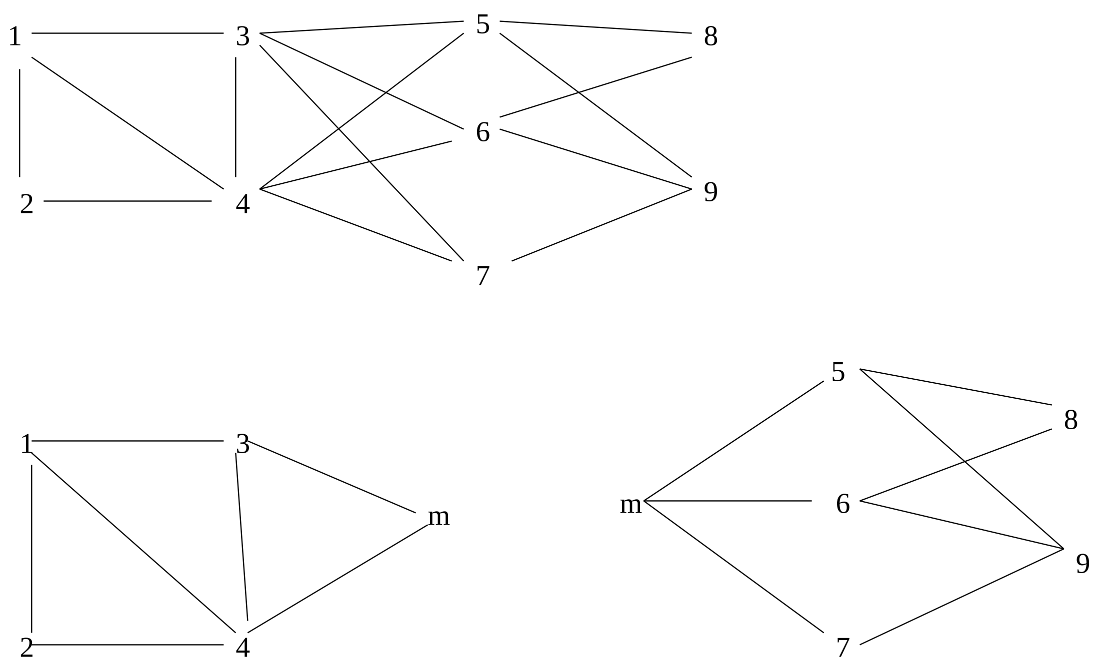


Figure 28: A Step of the Join Decomposition

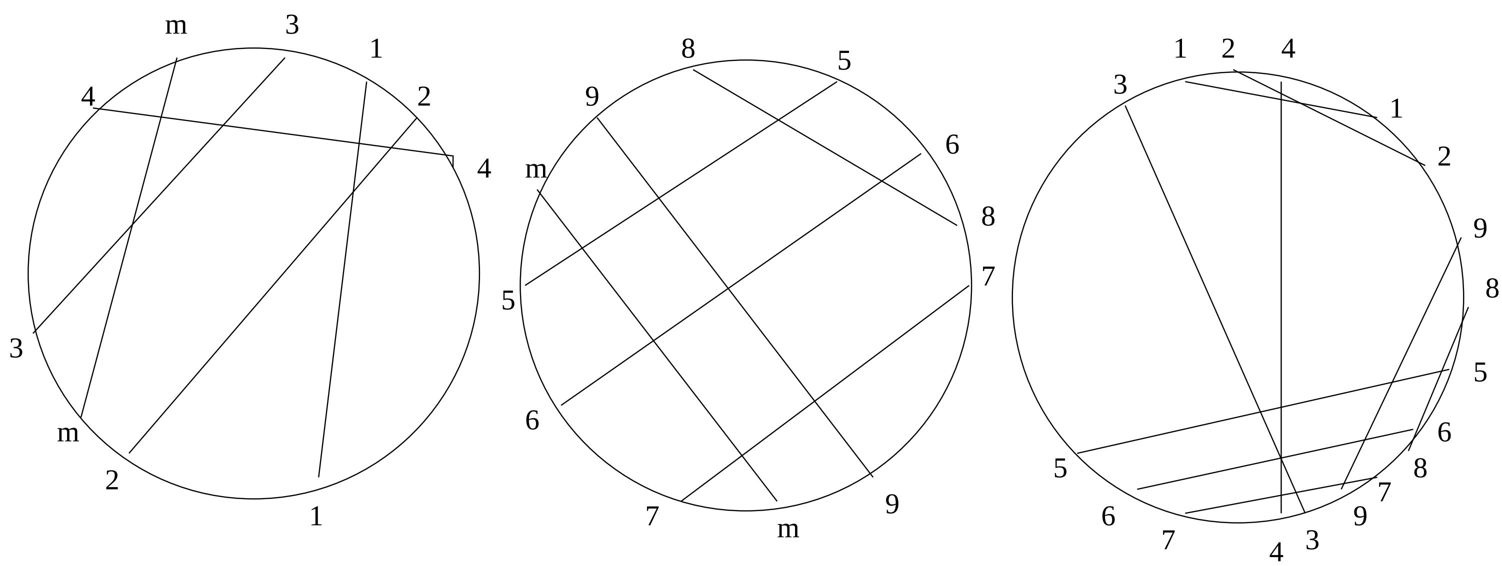


Figure 29: Combining Circle Representations for Join Decomposable Graphs

the recognition algorithm. We will now look at a simple O(nm) algorithm for finding the join decomposition, due to Cunningham [46] as modified in [69]. The fastest algorithm for finding the join decomposition runs in O($n+m$) time [47].

Let (x,y) be any edge of G. We test whether there is a split of G such that this edge crosses the split as follows. Let z be any other vertex of G. We test whether there is a split with x and z on one side, and y on the other, and we test whether there is a split with y and z on one side, and x on the other. Since the procedures are equivalent, we assume below that we are testing for a split in which x and z are in V_1 and y is in V_2.

Place vertices on a queue when they are known to be in V_1; originally, this queue contains vertex z. Vertices of V - x, z are placed in the set $V_{2io?}$ if they are adjacent to x, and in the set $V_{2?}$ if they are nonadjacent to x. While the queue is nonempty, we remove a vertex from the queue; call this vertex v. If v is nonadjacent to y (so that v must be in V_1 - V_{1io}), we remove all neighbors of v from $V_{2io?}$ and $V_{2?}$, and place them on the queue. If v is adjacent to y (v is in V_{1io}), remove neighbors of v from $V_{2?}$ and place them on the queue, and remove nonneighbors of v from $V_{2io?}$ and place them on the queue.

If this procedure stops with at least 2 vertices in $V_2io?$, we have found a split; otherwise, there is no split with x and z in V_1, y in V_2. For each vertex v, the time taken to partition $V_{2?}$ and $V_{2io?}$ is O(N(v)), so the total time spent partitioning sets is O(m). There is a possibility of O(n) work in adding vertices to V_1, meaning that we test for a split which is crossed by (x,y) in O(n+m) time.

If implemented as is, this gives quite a high time complexity for finding the join decomposition, but slight changes make for large improvements, as was noted in the original analysis by Cunningham of this algorithm [46]. Our initial analysis could involve testing m edges at a decomposition step, giving O(m^2) time to find a single split. Since there are at most n splits found by the join decomposition (exercise ?), this would give an O(m^2n) time bound for computing the entire decomposition.

Let T be any spanning tree for G. If there is a split, then some edge of T crosses the split. This allows us to test whether G has a split in O(nm) time, and gives an O(n^2m) algorithm for join decomposition. Another factor of n can be saved by the fact that if an edge does not cross any split of G, then that edge cannot cross any split of any components of G formed by the join decomposition [46]. This implies that any edge which is tested, and found not to cause a split can be reused in the spanning tree of one of the components, and does not need to be tested again for causing a split.

Suppose that we reuse all edges which are known not to cause splits in the spanning trees of each component. The total number of edges tested which cause splits is O(n). The final set of spanning trees is a forest on O(n) vertices, and includes all edges which were tested and did not cause a split; this implies that the number of edges tested which do not cause splits is O(n). Since O(n) edges are tested, and each test takes O(m) time, the time to find a join decomposition is O(nm).

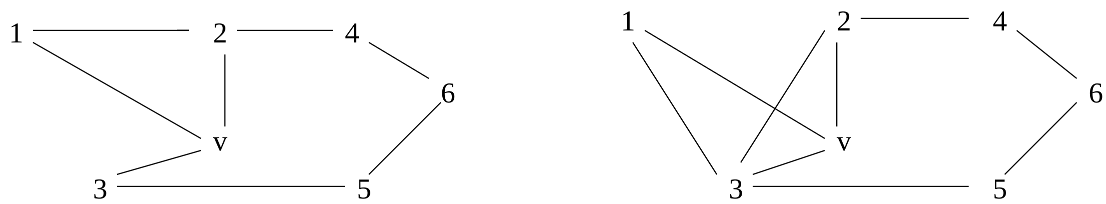


Figure 30: Local Complementation at v

13.6.2 Recognizing Circle Graphs

Using the results of the preceding section, we may assume that we are testing a graph which is indecomposable with respect to the join decomposition. The simplest approach we could hope for in recognizing circle graphs is to find a sequence of vertices to add to the model so that each subgraph encountered is prime; the unique representation of prime graphs means that we would only have to consider adding one chord to the existing model. To find this sequence, we might hope that each prime circle graph on k vertices would contain an induced prime circle graph on k-1 vertices; if this was true, our join decomposition algorithm would give a simple method for finding the sequence of vertices.

Unfortunately, there are prime circle graphs on k vertices which contain no prime circle graphs on k-1 vertices (exercise ?). Bouchet manages to use this basic approach to get a simple recognition algorithm, but needs to modify the graphs using an operation called local complementation, which is described below.

The local complementation of G at v, written as $G * v$, is the graph formed by making two neighbors of v adjacent if and only if these vertices are nonadjacent in v.

See figure ? for an example of local complementation at v. Local complementation corresponds naturally to a simple operation on a circle graph representation. If we have a model for G, a model for $G * v$ can be formed by reversing one of the semicircles defined by the two endpoints of v. Thus, G is a circle graph if and only if $G * v$ is a circle graph. In fact, we will use a stronger property: G is a prime circle graph if and only if $G * v$ is a prime circle graph (exercise ?).

Bouchet finds a sequence of vertices to add using the following theorem, which we present without proof.

Theorem [22]: Suppose that G is a prime circle graph with at least 5 vertices. Then there is a vertex v such that one of the following statements is true.

1) G - v is prime.

2) $G * v$ - v is prime

3) $G * v * w * v$ - v is prime for every neighbor w of v

Given the theorem above, it is easy to recognize circle graphs in polynomial time. First, G is decomposed with respect to the join decomposition; G is a circle graph if and only if each component is a circle graph, and a representation for the entire graph can be built easily from circle representations of the subgraphs. We then repeatedly find a vertex v using the theorem above, using O(n) calls to a primetesting algorithm for the join decomposition at each step, getting a smaller prime subgraph each time. Since we are interested in finding a smaller prime subgraph, it is not necessary for us to check every pair of vertices v,w for condition 3 on v; instead, we test each v with an arbitrary neighbor w of v, and either find a smaller prime subgraph or reject v as a choice.

We are left with a 5 vertex prime subgraph, and we find a circle graph representation of this subgraph. In general, we will go from a prime i vertex subgraph G_i to a prime i vertex subgraph G_{i+1}. If G_i was derived from G_{i+1} by removing v, we simply find appropriate places to add each endpoint of v. If G_i was derived from G_{i+1} via $G_i = G_{i+1} * v - v$, a model could come by reversing a semicircle of G_{i+1} defined by the endpoints of v, and removing v. We can thus try placing v at each possible position, reversing the appropriate semicircle, and seeing whether this represents G_{i+1}. If $G_i = G_{i+1} * v * w - v$, the transformation is a little more complex, but can be deduced using elementary observations; if the ordering on the circle in G_{i+1} is v,A,w,B,v,C,w,D then $G_{i+1} * v * w * v$ produces v,C,w,B,v,A,w,D. Thus, we can choose two endpoints for v, switch the positions of the corresponding circular regions A and C, and see if this represents G_{i+1}.

Finding the vertex sequence involves O(n^2) calls to a primetesting algorithm. This takes O(n^3m) time if we test each edge of a spanning tree using the algorithm described above. Primetesting can be done in O(n^2) time, either by the algorithms in [47, 120] for finding the entire decomposition, or the simpler algorithm in [156] for doing only primetesting; thus, the vertex sequence can be found in O(n^4) time.

As we go from G_i to G_{i+1}, we have O(n^2) possible placements of v to try; these placements may also require the rotation of semicircles if $G_i = G_{i+1} * v - v$ or $G_i = G_{i+1} *vwv - v$. If we perform completely new computations each time, we could take O(n^4) time to go from G_i to G_{i+1}, getting an O(n^5) time bound for circle graph recognition. However, it is easy to keep track of the changes as we go from placing v at position i,j to position i, j+1 efficiently. We leave the details of finding a representation of G_{i+1} from a representation of G_i in O(n^3) time to exercise ?; this produces an O(n^4) algorithm for recognizing circle graphs.

13.6.3 Circle Graph Obstructions

The local complement operation was used again by Bouchet to characterize circle graphs in another way. Even before we had a recognition algorithm, circle graph recognition was obviously in NP, since you can prove that you have a circle graph by giving the representation on the circle. Although we can recognize circle graphs in polynomial time, it is not completely satisfying to show that a graph is not a circle graph by running this algorithm, and saying that it is not a circle graph because the algorithm fails to find a representation. Bouchet was able to find a simpler description which shows that circle graph recognition is in co-NP.

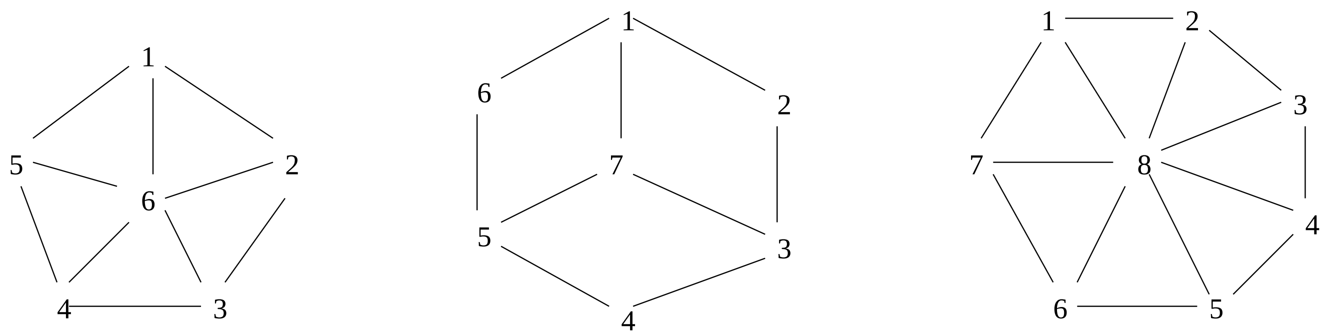


Figure 31: Circle Graph Obstructions

Theorem [21]: G is a circle graph if and only if there is a series of local complementations and vertex deletions which yield one of the subgraphs shown in figure ?.

13.7 Circular Permutation Graphs

Permutation graphs are the intersection graphs of segments which connect endpoints lying along two parallel lines. Circular permutation graphs, first introduced in [141], generalize this by letting vertices have endpoints on two concentric circles rather than two parallel lines. These vertices now correspond to chords, and two vertices are defined to be adjacent if and only if the corresponding chords intersect exactly once. Although they are not defined in this way, I find an alternate representation which was known to the authors of [141] but was not written down to be more appealing; circular permutation graphs are exactly equal to the containment graphs of arcs on the circle (exercise ?). This representation also makes it clear that circular permutation graphs are comparability graphs, which is important to a nice characterization theorem for the class.

It is natural to try to reduce the recognition problem for circular permutation graphs to recognition of permutation graphs. This leads to the notion of cutting the circles at the endpoints of a vertex v, and 'unrolling' the representation into a permutation graph. A chord w which crosses v will have to be sent around in the other direction of the circle; for a single w, this will have the effect of making w adjacent to exactly those vertices which are nonadjacent to w.

Thus, we have a natural operation switch(G,x), which makes x adjacent to V - x - N(x). We say switch(G,S) is the graph formed from G by successively switching all vertices in S; the net effect is to leave adjacencies within S unchanged, and complement adjacencies between S and V - S.

It is not hard to see that if G is a circular permutation graph, then for any v, switch(N(v)) is a permutation graph. This is not a sufficient condition, but becomes one when combined with the fact that this class is contained in the class of comparability graphs.

Theorem [141]: Let v be an arbitrary vertex of G. Then G is a circular permutation graph if and only if G is a comparability graph, and switch(G,N(v)) is a permutation graph.

Theorem ? gives an O(MM) algorithm for recognizing circular permutation graphs, where the bottleneck step is verifying that G is a comparability graph. Sritharan [161] modifies the steps of the algorithm to avoid the bottleneck, and produces a linear time algorithm as described below. He uses the results of chapter ?, which shows that a comparability graph can be transitively oriented in linear time, even if we cannot verify that the result is transitive in this time bound.

In general, switching vertices does not preserve possible transitive orientations. However, it is not hard to see that switching a source or sink of a transitively oriented graph gives a transitively oriented graph; the orientation simply turns a source s in G into a sink in switch(G,N(s)). This gives rise to a notion of directed switching for a source vertex; given a source (sink) s in a directed graph G, switch(G,s) is the graph formed by removing all edges out of s, and adding edges from all nonneighbors of s into s for sources, and to nonneighbors from s for sinks. If s is involved in a transitivity violation $s \to x \to y$, switch (G,s) contains the transitivity violation $x \to y \to s$; thus, a digraph is transitive if and only if switching a source or sink is transitive.

Thus, Sritharan's algorithm works as follows. Let G be an input graph, and let v be a vertex of minimum degree in G. Find the graph G' produced by the linear time transitive orientation algorithm in [124]. While some neighbor s of v in G is a source or a sink in G', let $G' =$ switch(G',s).

If the algorithm switches all neighbors of v, and the final graph G' is a two dimensional partial order, then G is a circular permutation graph; otherwise, G is not a circular permutation graph. If all neighbors are switched, the fact that switching sources preserves transitive orientations means that the original graph must have been a comparability graph, and the underlying graph of G' is a permutation graph equal to switch(G,N(v)). If the algorithm stops before all neighbors of v are switched, then G' is not transitive (exercise ?). Together, this shows that Sritharan's algorithm produces the same output as the algorithm of [141].

We now show that the time complexity of this algorithm is linear. The time complexity is dominated (in the asymptotic sense) by the cost of computing G'. Each neighbor of v is switched once. Since a vertex can be switched in O(n) time, the total cost is O($n|N(v)|$). Since v is a minimum degree vertex, this is less than or equal to the sum of all vertex degrees $= 2m$, so the total cost of finding G' (as well as the final size of G') is O(n+m).

13.8 Paths in Trees

There are many classes of graphs defined as intersection graphs of paths in trees. We will follow the classification scheme from [128], which gives the following variants, as well as a number of other classes we will not cover here.

1) A path may be viewed as a sequence of vertices, or as a sequence of edges.

2) A tree may be either undirected or directed. Directed trees may or may not be rooted.

This allows us to define 6 classes of graphs. A class defined by vertex intersection will end with a V, while a class defined by edge intersection ends with E. Classes defined by undirected trees start with U, directed (not necessarily rooted) trees with D, and rooted directed trees with RD. Since the terminology used for directed trees is not standardized, let us clarify the distinction between D and RD; a directed tree is a digraph such that removing directions from edges gives a tree, and a directed tree is rooted if there is one vertex r which has a path to all other vertices. Thus, the class UV is the intersection graphs of paths in undirected trees, where two paths intersect if they share a common vertex, while the class RDE is the intersection graphs of paths in a rooted directed tree, where vertices are adjacent if the corresponding paths share an edge. The classes RDV and RDE are equivalent (exercise ?), so we are left with 5 classes to study. The UE graphs, more commonly known as EPT graphs, are NP-complete to recognize.

We will present a polynomial time recognition algorithm for the class DE here, and explain how to trivially modify the recognition algorithm for DE to recognize DV as well. The first algorithm for recognizing the class RDV in polynomial time is presented in [74]; a linear time algorithm appears in [50]. For the class UV, or path graphs, the first recognition algorithm due to Gavril runs in $O(n^4)$ time; the fastest current algorithm is due to Schäffer and runs in $O(nm)$ time [142]. Gavril's algorithm places all cliques containing a single vertex in a valid path in a phase; he constructs a necessary and sufficient set of conditions for constructing such a path, and gives an algorithm for constructing the path. Unfortunately, the set of constraints is difficult to describe concisely. Schäffer's algorithm is similar to the algorithm to the algorithm for DE recognition described above, constructing a tree from the subtrees formed by a clique separator, but one step of the construction is much more difficult.

13.8.1 Recognizing Edge Intersection Graphs of Directed Paths in Trees

We present an algorithm due to Monma and Wei [128] to recognize graphs in these classes.

The algorithms for recognizing DV and DE can be summarized as follows. We first find a maximal clique separator tree for G. This can be obtained by modifying Tarjan's algorithm which is presented in chapter ? in $O(nm)$ time; we do not give the modifications which produce a maximal clique separator tree rather than a clique separator tree.

We test each atom (leaf) of the maximal clique separator tree to see that it is in the class; a tree model can be constructed for these if it is in the class. We then look at each separator of the tree in postorder, and see whether the trees on the children can be combined to derive the appropriate tree on the entire subgraph.

It is not hard to see that the atoms of any of these classes must correspond to paths in some directed or undirected star, since any path of length 3 in the tree has a clique separator consisting of paths which go through the middle edge. For the class DE, the atoms of the clique separator tree are line graphs of bipartite multigraphs (exercise ?), which implies a linear recognition algorithm. To design your own algorithm which does not affect the overall running time of the algorithm, first note that the graph induced by neighbors of each vertex must have clique cover number 2; it is not hard to show that this implies that every vertex is in at most two maximal cliques, and that all maximal cliques can be found efficiently. Build a graph G_c with a vertex for each maximal clique, and edges between two vertices if the cliques share a common node. A graph is an atom for DE if and only if G_c is bipartite, and the coloring can be used to derive the appropriate star representation for the atom by making one color class the inedges of the star and the other color class outedges; a vertex of G then corresponds to a path in this star going between its two maximal cliques.

We now show how to take a clique separator C for a graph in which all subgraphs are known to be DE, and test whether the entire graph is DE. The algorithm comes from the following ideas. Every clique C of a DE graph shares a common edge (exercise ?). The paths for each vertex of C then branch off from this common edge e or set of edges in two directions; one direction for edges which come after e in all paths, and one for edges which come before e in all paths. If two connected components G_i,G_j of G - C are connected to different subsets C_1, C_2 of C, where neither C_1 nor C_2 contain the other and both contain a common vertex, then one of G_i,G_j must be attached to C before the edge e, while the other must be attached to C after the edge e. This idea is refined slightly to get a complete condition of when two components can be attached on the same side of e, and attaching the models to C is transformed to a two coloring problem on a conflict graph for the components.

Create a vertex G_i for each connected subgraph of G - C. Create an edge between G_i and G_j if and only if at least one of the following conditions holds for some set of cliques defined as follows. Let C_i, C_i', C_i'' be maximal cliques of $G_i \cup C$, and C_j, C_j' and C_j'' be maximal cliques of $G_j \cup C$, where these cliques need not be distinct, and all cliques coonsidered include at least one vertex of C. We say $C_1 \geq (>)\, C_2$ if $C_1 \cap C$ contains (properly contains) $C_2 \cap C$, and $C_1 = C_2$ if these have the same neighbors in C, and $C_1 \mid C_2$ if the cliques have no common neighbors in C. The conditions imply that one subgraph cannot be attached 'further along a common path' from e than the other subgraph. Components with an edge between them are called antipodal components.

Conditions for adding an edge between G_i and G_j:

1) C_i, C_j share a common vertex, and neither contains the other.

2) $C_i > C_j$, $C_j' > C_i'$.

3) $C_i > C_j$, $C_i' \geq C_j'$, $C_i' \mid C_j''$.

4) $C_i = C_j$, $C_i \mid C_j'$, $C_j'' = C_1'$, $C_j'' \mid C_1''$

If the graph is not two colorable, G is not DE. If the graph is two colorable, there is a natural ordering on each color class which allows us to produce both a valid branching of C and a gluing of the trees on the subgraphs to produce a representation of G. Details are presented in [128]; the reader should understand the general construction, in which G_i precedes G_j if some $C_i > C_j$, which implies that G_i is attached to C closer to e than G_j. Note that for the classes of graphs in question, the number of maximal cliques is $\mathrm{O}(n)$ (for DV this comes from the fact that these are chordal, while [128] shows that edge paths in directed trees satisfy the Helly property), so all cliques in each subgraph can be enumerated and these conditions can be tested in polynomial time.

The algorithm for recognizing the class DV is very similar to the one given above. In fact, the only difference in recognition algorithms is the test as to whether an atom of the decomposition is in the class. Since any internal vertex of a tree representation is a clique, the atoms of a DV graph are either a single clique, or two cliques sharing some common vertices; these conditions can easily be tested in linear time.

13.9 NP-complete Classes

In this section, we list related class of graphs for which the recognition problem is NP-complete. We do not want to imply that the NP-completeness proofs are all easy or uninteresting; authors such as Kratochvil and Yannakakis found nice constructions to show longstanding open recognition problems for classes with important applications were NP-complete. However, the conscious effort in this book to deal less with proofs than with algorithms means that the proofs themselves will not be presented here.

I would like to mention at this point that each NP-complete class which has an implicit representation poses a new problem which has not been studied until now. Can we find an implicit representation of members of the class in polynomial time? We discuss this question in chapter ?.

Partial Order Dimension: Is there a set of k linear extensions of P such that for every nonadjacent pair x, y, x comes before y in at least 1 list while y comes before x in some other?

This was one of the original open problems in Garey and Johnson, and was found to be NP-complete for all $k > 2$ by Yannakakis [183]. Variants of the proof used showed that interval dimension k and boxicity k (as discussed below) are NP-complete for all $k > 3$. Some open problems related to this are given below; more open problems on dimension can be found in [165, 159].

Open problem: Can height 1, dimension 3 posets be recognized in polynomial time?

Yannakakis showed that height 1, dimension k poset recognition is NP-complete for all $k > 3$. Interest in height 3, dimension 1 posets was heightened by an interesting result due to Schnyder [146]. The vertex/edge inclusion poset of a graph G is a height 1 poset with an element for each vertex, and an element for each edge, with an edge from v to e if and only if v is an endpoint of e. G is planar if and only if the vertex/edge inclusion poset has dimension at most 3.

Open Problem: Dimension Approximation. Suppose that you are given a partial order P of dimension k. Can you find a set of f(n,k) linear extensions which realize P in polynomial time?

Even for 3 dimension partial orders, no algorithm for producing a set of o(n) linear extensions which realize P seems to be known.

EPT graphs: Given a graph G, is there a tree T such that G is the intersection graph of paths in T, where paths are defined to be intersecting if they share a common edge?

Surprisingly, the seemingly small difference in definition from path graphs, where paths intersect if they share a common vertex, makes the recognition problem NP-complete. EPT graphs are not perfect; in fact, the coloring and clique cover problems are NP-complete on EPT graphs [80]. The clique and independent set problems on EPT graphs are discussed in chapter ?.

Boxicity: Given a graph G and an integer k, is G the intersection graph of isooriented boxes in k dimensional space?

This was proved to be NP-complete for $k > 3$ in [183]. The boxicity 2 problem, which seems to have applications in such distant fields as ecology [138] and others, was open for many years before being shown to be NP-complete in [109].

Covering Graph: Given a graph G, can you direct G to be the transitive reduction of a partial order?

This was shown to be NP-complete in [25]; note that if transitive reduction is changed to transitive closure, we get the comparability graphs. The problem has not been examined for special classes. Perhaps it would be possible to determine whether it is possible to direct the edges to be the transitive reduction of a two dimensional partial order, giving the covering graph equivalent of permutation graphs, but since dimension is not a covering graph invariant I am less inclined to believe that this is possible.

Planar Posets: A poset is planar if one can embed the transitive reduction in the plane so that every edge goes upwards, and no edges cross. This was shown to be NP-complete in [72].

disk intersection: Is G the intersection graph of disks in the plane?

We discuss circle intersection graphs in chapter ?. These were shown to be NP-hard to recognize in [110].

Interval Number: Given a graph G and an integer k, is G the intersection graph of k intervals per vertex?

This was shown to be NP-complete in [175]; among variants, one of interest to us is given below, since it is not an implicit representation.

Total Interval Number: Given a graph G and an integer k, is there a set of at most k intervals such that each interval is associated with some vertex, and G is the intersection graph of these intervals?

Lines in plane: There are several relevant models here. Hardness results were shown for a variety of such classes in [109]. NP-complete classes to recognize include intersection graphs of horizontal and vertical lines on the plane, and lines in general position. If we allow intersection graphs of not simply lines but 'strings' in the plane, we get the string graphs, which are NP-hard to recognize [109]. It is interesting to note that no algorithm, no matter how large the time complexity, is known for recognizing string graphs.

k-polygon graphs: Given a graph G and an integer k, is G the intersection graph of chords of a k-gon, where a chord must go between two different sides? If k is part of the input, this is NP-complete, but is polynomial for any fixed k [56].

13.10 Open Classes

We collect several of the most closely related problems for which recognition is not known to be polynomial or NP-complete. Visibility graph recognition is dealt with in chapter ?.

Open Problem: Triangle Graph Recognition. There are two related classes of graphs, which contain the permutation graphs and are contained in the trapezoid graphs, and for which the recognition problem remains open. These are the classes PI (point-interval) and PI*, in which vertices map to triangle with endpoints on two parallel lines; the difference between the classes is that in ? all triangles have a single endpoint on the top line, while in ? the single endpoint may be on either the top or bottom line.

Open Problem: Triangles inscribed in circle. This is the intersection graphs of triangles inscribed in the circle, first thought of by ?. The problem is also open for k-gons in the circle for fixed k; intersection graphs of arbitrary polygons inscribed in the circle can be recognized in polynomial time [107] (note to self: find and check this algorithm!!!).

Open Problem: Containment graphs of paths in a tree. For some reason, although there have been many papers on variants of intersection graphs of paths in trees, this class has been overlooked. Perhaps it is because applications of the intersection model are more apparent, but this class seems sufficiently natural to deserve some attention.

Open Problems: Circle orders and angle orders. These are two well known classes of posets, defined as the containment graphs of circles and of angles. There seem to be no results on recognizing these classes; the techniques used for proving circle intersection graphs are NP-complete may be useful here.

13.11 Exercises

1) Design a linear time algorithm for recognizing co-chordal graphs.

2) Let M be a module in a chordal graph. Show that if G is chordal, then either M is a clique or N(M) - M is a clique.

3) Design an O(n+m) algorithm to determine whether an interval representation correctly represents G, assuming endpoints are integers in the range $1..2n$.

4) Show that if v is in k maximal cliques of a chordal graph, then $|\mathrm{N}(v)|$ is at least k.

5) Show that there is a diametral pair of simplicial vertices in a chordal graph; that is, if the maximum distance between pairs of points in G is k, there is some pair of simplicial vertices at distance k apart.

6) show can get within 1 of SP in chordal graph easily

7) Show that a partial order is an interval order iff the downsets are ordered by set inclusion.

8) Design an O(n^2) algorithm which finds a maximal independent set S such that there is no $x \in S$, y, $z \notin S$ such that S - $x \cup y, z$ is an independent set.

(lots of circ-arc probs from Vandy)

circarc clique cover k

2) Show that verification phase of chordal graph recognition is valid.

3) Show that a circular-arc graph can have an exponential number of maximal cliques.

4) Show some circ-arc model has required types of arc overlaps.

5) Show that G is a circle graph iff G is the overlap graph of a set of intervals on the line; in the overlap model, two vertices are adjacent iff the corresponding interval have a nonempty intersection, and neither interval contains the other.

5.1) Show that an interval graph can be preprocessed in linear time so that neighborhood containment queries can be answered in constant time.

5.5) Show that for any minimal arc v in a circular-arc model (i.e. an arc which properly contains no other arc), the subgraph induced by N(v) has clique cover number 2.

5.6) graph D in circ-arc construction is a cbg

6) Show that after O(n) splits using the join decomposition, every subgraph is prime.

7) G prime circle graph iff $G * v$ prime circle (check; could also use G prime iff G * v prime on arbitrary G)

8) Find prime circle G s.t. no induced subgraph prime (try self)

9) Show that circular permutation graphs are exactly equal to containment graphs of arcs on the circle.

10) Find a graph G and vertex v such that switch(N(v)) is a permutation graph, but

G is not a circular permutation graph.

11) approp hwk on adding v to prime circle G on i vertices

12) show can get from G_i to G_{i+1} in O(n^3) time in circle graph recognition

13) Show if G' transitive repeatedly switching source/sink neighbors of v switches all neighbors of v

14) int dim 2 trapezoid graph relationship

15) verification of int dim 2 and trap graph representation

16) We can also test whether G is an interval graph by determining from the interval model whether G contains $K_{1,3}$.

16a) Show how to find for all v from interval model leftmost nonneighbor to right of v in O(n).

16b) Show how to find for all v from interval model neighbor with least left endpoint in O(n).

16c) Show how to determine whether G is a unit interval graph from the above information in O(n) time.

17) containment relationships in paths of tree models.

18) Show that every clique of a DE graph shares common edge in the tree, and that this is not true for cliques in UE graphs.

19) Show that every partial order is the containment graph of substars of a star.

20) Show that there are 3 dimensional posets which are not containment graphs of paths in a tree, and that there are posets which are not 2 dimensional but are not containment graphs of paths in a tree.

(note to self: why not k-dim? know I have worked this out before).

21) Show that if G is a proper circular-arc graph, there is a circular-arc model in which all pairs of adjacent vertices have an overlap relationship.

22) Show how to find an optimal coloring of a circular-arc graph with clique cover number two in linear time.

23) Tucker showed in every model of a circular-arc graph with clique cover number two, there is a pair p_1, p_2 of points on the circle such that every arc passes through at least one of p_1,p_2. Show that this can be extended to show that in every model of a circular-arc graph with clique cover number $k > 1$, there is a set of k points on the circle such that every arc passes through at least one of these k points.

24) Design an O(n+m) algorithm which finds an odd cycle in the complement of G, or determines that G has clique cover number 2.

25) Show that we can construct a consecutive clique arrangement from an interval model in O(n+m) time.

Note: Move question to partitioning chapter?

25) Proper circular-arc graphs can also be recognized as follows. We assume that G is not an interval graph. If G does not have clique cover 2, find an odd cycle C in the complement. If C is an independent set a, b, c, find paths between each pair which avoid all neighbors of the third vertex, and place these on the circle. Otherwise, lay out C on the circle. Place each other vertex according to which regions of the circle an endpoint is in (special group for clones). Use partitioning to get a proper circular-arc representation

a) Show that if C has length > 3, C has a unique placement on the circle.

b) Show that in the partitioning process, no groups meet on both side of the circle. This makes it easier to see that there is only one choice of placements whenever a group is split.

26) Prove or disprove: G has a split iff the complement of G has a split

27) G has a k-join iff the vertices of V can be partitioned into V_1, V_2, each containing at least 2 vertices, and edges which go between V_1 and V_2 come from (make sure not partitioned into) k complete bipartite subgraphs. Show that one can find a k-join of G, if it exists, in polynomial time.

14 Optimization Problems

In this chapter, we look at the relationship between optimization problems and representations of graphs.

Before we look at solutions to optimization problems on specific classes of graphs, we want to emphasize some issues that are rarely addressed in the literature. This is especially important when we deal with classes of graphs which have nice representations, but there is no known polynomial time algorithm for constructing these representations; indeed, recognizing the class of graphs and constructing the representation may be NP-hard. It is often important when dealing with these classes to distinguish between solving the optimization problem on the class if the input is given in the form of a good representation, and if the input is given in the form of adjacency lists or matrices. When the recognition problem is NP-complete, this separates out the question of whether a problem is easy on a class because the graph class is tractable from whether the problem is easy on the class because you are given the answer to an NP-hard problem.

A further distinction is important if the input is given in adjacency list or adjacency matrix form. Suppose that we know a fast algorithm for solving an optimization problem if it is in a special class C of graphs. Is it sufficient to run the algorithm, and say that we have solved the problem on the class C? The answer depends on our assumptions. If we are given a true promise that the input graph is in the class, then we are satisfied. However, we may only suspect, or perhaps merely even hope, that the graph is in C. If we are required to run a recognition algorithm to determine whether G is in the class, we cannot solve any problems on NP-complete classes efficiently. There is another way to solve optimization problems on special classes that should, in my opinion, be entirely satisfactory, even if the recognition problem is NP-complete for the class.

An algorithm which solves a problem (either a decision problem or an optimization problem) on class C is called robust if it satisfies the following conditions.

1) Whenever the input is in class C, the algorithm finds the correct solution.

2) If the input is not in the class C, then the algorithm either finds the correct solution, or answers that the input is not in the class.

An algorithm which satisfies condition 1 above but not condition 2 will be called a nonrobust algorithm for the class C.

Robust algorithms might seem rare, but they are actually quite common. Often, they work by solving the problem on a larger class of graphs. For example, it is NP-complete to recognize planar Hamiltonian graphs, but any algorithm which solves a problem on planar graphs is a robust optimization for planar Hamiltonian graphs. In other cases, the problem solves the problem exactly on the class C, but if the input is in some larger class it may (depending on choices made by the algorithm) either solve the problem or answer that it is not in the class C. We will make this issue more concrete by introducing an open problem related to robust algorithms.

A graph is well covered if every maximal independent set is a maximum independent set. Well covered graphs are studied in []. Well covered graphs are interesting in this context, because finding a maximum independent set in a graph which is known to be well covered is particularly simple (any greedy method works), but recognizing well covered graphs is co-NP complete [147]. Thus, if we are given a promise that the graph is well-covered, independent set is in P; it is not clear what happens in the model that we present here. In our terminology, there is a nonrobust algorithm for independent set in well covered graphs, but the existence of a robust algorithm for the class remains open. We note that the representation problem for well covered graphs is not interesting, since the number of well covered graphs on n vertices is $2^{\Theta(n^2)}$. Later in this chapter, we will show that there is no robust polynomial algorithm for the independent set problem in well covered graphs unless P = NP.

In the opinion of this author, it is important to design robust algorithms for a problem before you declare the problem to be solved on the class of graphs. Although situations exist in which you may have a true promise that a graph is in a class, one cannot completely trust the result in general unless it produces some form of valid output if the graph happens to be outside the class. As an example, ? conjectures that all ? graphs of ecosystems have boxicity 2. Thus, algorithms for solving problems on ? graphs would then want to use boxicity 2 algorithms, if the problem is solved efficiently on boxicity 2 graphs. However, if ? conjecture is wrong, we do not want to get incorrect answers to our problems!

For some difficult problems, we might want to create a library of routines that work for certain restricted classes. The first thought is to have a corresponding recognition algorithm for each class, so that if a new graph is given to the library, we test whether it is in each class, and if it is in a tractable class we run the appropriate algorithm. However, this assumes that the recognition algorithm is at least as fast as the optimization algorithm. We will see examples of problems for which the recognition problem is harder than the optimization problems for the class. For these classes, a robust algorithm lets us avoid the recognition problem entirely.

In the next section, we will examine classes of graphs and problems for which a robust algorithm exists which is faster than any recognition algorithm. In the following section, we look at problems where having a representation leads to algorithms which are much more efficient than any algorithm which is known if the input is given in adjacency list form. Some of these problems will be candidates for problems in which the representation provably helps, an open problem mentioned in chapter ?. We note that independent set in well-covered graphs does not quite fit either model; this is a problem for which a promise, rather than a representation, seems to help. (note: will add a section if I can think of enough of these).

14.1 Robust Algorithms which are Faster Than Recognition

We start with an example of a famous problem on a natural class of graphs, such that there is a robust algorithm which is much faster than the recognition problem.

Consider the maximum clique problem on boxicity k graphs for any fixed value of k. The recognition problem for boxicity k graphs is NP-complete for any fixed k greater than or equal to two (see the recognition chapter).

It is easy to find a maximum clique in a boxicity k graph if we are given a representation in k-space. The key observation is that k-dimensional boxes have the 'Helly property'; this says that any clique must meet at some common point. It is easy to see that a boxicity k graph can be represented on a kn by kn grid. Thus, we can simply check how many boxes cover each of the grid points; the maximum value will be the size of the maximum clique.

It is not difficult to modify the above approach to produce a robust algorithm for the maximum clique problem on boxicity k graphs. The argument above shows that if G is a boxicity k graph, then the number of maximal cliques of G is at most k^2n^2, since each maximal clique corresponds to some grid point in a kn by kn grid. Maximal cliques of a graph can be enumerated in O(??M) time, where M is the number of maximal cliques, using an algorithm which takes O(??) time to enumerate each new maximal clique []. Given an arbitrary graph as input, we can enumerate maximal cliques until either the number of maximal cliques is greater than k^2n^2, in which case we answer that G does not have boxicity k, or we have enumerated all of the (at most k^2n^2) maximal cliques, in which case we solve the maximum clique problem correctly whether or not the input graph has boxicity k.

A similar, but slightly more complex example, deals with EPT graphs. These are the intersection graphs of paths in a tree, if two paths are said to intersect when they share a common edge rather than a common vertex as in path graphs. This apparently small variant makes the class NP-complete to recognize; the coloring and clique cover problems also remains NP-complete on the class [80]. We will look at polynomial algorithms for the clique and independent set problems on EPT graphs; the algorithms are based on properties which come from the model but do not need to be given the model as input.

For the clique problem, although all cliques must go through a common tree vertex, it is not true that all sets which go through a common tree vertex form a clique; thus, we might think that the number of maximal cliques could be exponential. It is not necessary for all cliques to share a common edge (exercise ?). However, Golumbic and Jamison [80] were able to show that each maximal cliques either corresponds to a set of paths which share a common edge of T, or a set of paths which go through two edges of some claw ($K_{1,3}$) in T. Since the size of T can be shown to be O(n) (exercise ?), the number of maximal cliques is polynomial, and we can solve the clique problem in polynomial time without being given the tree model.

The algorithm for solving the independent set problem on EPT graphs in polynomial time is due to Tarjan [163]. We introduce a form of graph decomposition called the clique separator decomposition. As the name implies, this involves finding a clique C such that the graph induced by G - C is disconnected; we then decompose G_i *cup* C for each connected component G_i of G - C. Clique separators can be used to decompose an EPT graph into subsets of vertices which correspond to paths in stars;

if the tree is not currently a star, then the set of vertices which pass through any edge e which disconnects the tree form a clique cutset. An algorithm for computing this decomposition is given in chapter ?.

The edge intersection graphs of paths on a star are exactly equal to the line graphs. Let G be an EPT graph of a star. We can assume that all paths in the tree contain 2 edges (exercise ?). Think of the noncentral vertices of the star as vertices of a graph G'. Vertices of G correspond to paths in the tree, which form the edges of G'. Two vertices in G are adjacent if and only if the corresponding edges of G' share a common endpoint; thus, G is the line graph of G'. We note that if G is a line graph, we can find a graph H such that G is the line graph indexline graph of H in linear time []. (give somewhere in book?)

Independent sets in a line graph correspond to matchings in the associated graph; thus, we can use algorithms such as [] to solve the weighted independent set problem in polynomial time for each final graph found by the clique separator decomposition. Let G_i be one of the final graphs found by the clique separator decomposition, and let C be the clique separator which separates G_i - C from the remaining graph. Find the weight w_i of the maximum independent set in G_i - C. For each c in C, compute the weight w_c of the maximum independent set on G_i - C *cup* c. Remove G_i - C from the graph, and give each vertex c in C a weight w_c - w_i. The weight of the maximum independent set in the new graph is equal to the weight of the maximum independent set in G - w_i.

This gives a polynomial time algorithm for finding the maximum independent set problem on an an EPT graph, which does not use the tree model. If the input graph G is not an EPT graph, we will either find that some graph found by the clique separator is not a line graph, or we will find the correct value for the maximum independent set in G.

We noted in chapter ? that the fastest known algorithms for recognizing strongly chordal graphs and finding a strong elimination scheme take $\Theta(n^2)$ time. However, strongly chordal graphs are a subset of the doubly chordal and dually chordal graphs defined in chapter ?, and there are linear time algorithms for the dominating set problem on doubly and dually chordal graphs; thus, we can solve the optimization problem on strongly chordal graphs faster than they can be recognized. This is of more interest than, for example, solving an optimization algorithm on 3-colorable planar graphs by using an algorithm designed for general planar graphs, since using the strong elimination ordering would seem to be the natural method for solving the dominating set problem on strongly chordal graphs.

The above problems are examples of solving the problem on larger classes of graphs, such as graphs with bounded number of maximal cliques, and graphs which decompose via clique separators into line graphs. Nevertheless, it is important to realize that the reason that the problem is easy on the class is not because we are given a nice representation, but because the class has interesting properties which allow fast optimization.

Many algorithms will be of the form above. However, it is not always so easy to determine what the larger class is that can be solved efficiently, and finding the larger class can lead to solutions of other optimization problems on the class. (Kloks+Kratsch results).

Finding a robust algorithm does not always mean solving the problem on a larger class of graphs. We give examples in which the class of graphs which the problem is solved by the robust algorithm is not well defined, since there are many graphs which may be determined to be outside of the class or optimized correctly, depending on random choices such as how the vertices of the graph are numbered.

Our first example is the clique problem on comparability graphs. As we mentioned in chapter ?, there is a linear time algorithm for producing a transitive orientation of a comparability graph, but there is no known recognition algorithm for comparability graphs which is faster than matrix multiplication. The orientation algorithm will direct the edges of a comparability graph correctly in linear time, but on other graphs it can produce some other (nontransitive) orientation in linear time; checking the transitivity of the resulting digraph is the bottleneck step of the recognition problem.

Suppose that we are given a linear extension L of a transitive orientation of a comparability graph G. The following algorithm finds the size of the maximum clique in G. We keep track of the size of the largest clique $\mathrm{C}(i)$ found involving each vertex i; initially $\mathrm{C}(i) = 1$ for all i. Step through L in order; let x be the next vertex of L. For each edge $x \rightarrow y$ of the transitive orientation of G, let $\mathrm{C}(y)$ be the maximum of $\mathrm{C}(y)$ and $1+\mathrm{C}(x)$. When we reach the end of L, the maximum clique in G is the largest value of $\mathrm{C}(i)$.

We first show that this algorithm works to find a maximum clique if G is a comparability graph, but does not necessarily work if G is not a comparability graph. We then give a simple modification which makes it a robust algorithm

Suppose that G is a comparability graph. Suppose that G has a clique $C = c_1$, c_2,...,c_k, where vertices of C are numbered in increasing order of position in L. When each c_i is reached, the edge $c_i \rightarrow c_i + 1$ causes c_i to have a final value which is at least one larger than $c_i + 1$, so we will answer that G has a maximum clique of size at least k. Suppose that the algorithm answers that G has a clique of size j. We can find a sequence c_1,c_2,...,c_j such that c_i caused $c_i + 1$ to be labeled with the value j. This implies that there are edges from c_i to $c_i + 1$ for $i = 1$,2,...,n-1; since the orientation is transitive, this implies that c_1,c_2,...,c_j form a clique of size j in G. Therefore, if G is a comparability graph, we can find the maximum clique in G in linear time.

Of course, this algorithm may produce an incorrect value if the input graph G is not a comparability graph. Consider a graph consisting of $C_7 =$ 1,2,3,4,5,6,7,1 and a vertex 8 adjacent only to 1, 3, and 5, and suppose that the orientation algorithm produces the list $L = 87645213$ as its incorrect linear extension. Then $\mathrm{C}(6) = 3$ at the end of the algorithm, although G has no clique of size 3. However, this problem can be remedied easily, using the observation that the value found by the algorithm is always an upper bound on the value of the maximum clique.

Our modification of the algorithm above to make it a robust is straightforward.

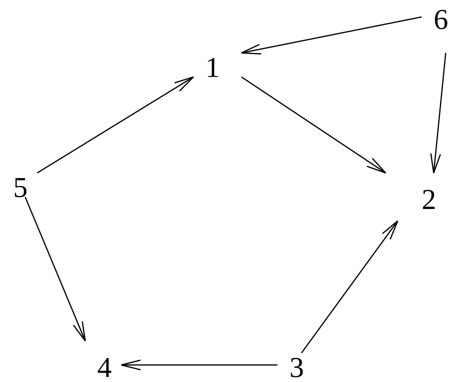


Figure 32: Possible Orientation for a Noncomparability Graph

For each vertex v, maintain a parent pointer p(v), with all parent pointers initially nil. We run the same algorithm, but whenever vertex x changes the value of C(y), we change the parent value of y to point to x. When the maximum value C(i) is found at the end of the previous algorithm, we use the parent pointers to construct the set S which should be a clique of size i if G is a comparability graph. If S is a clique, then we say that the maximum clique in G has size i, and if S is not a clique we answer that G is not a comparability graph.

If G is a comparability graph, this algorithm works for the reasons proved earlier. Suppose that G is not a comparability graph. If the algorithm answers that G is not a comparability graph, we have found a set S such that the orientation algorithm produced edges $s_1 \rightarrow s_2 \ldots \rightarrow s_i$, but S was not a clique of G. Since the orientation produced was not transitive, G is not a comparability graph. Suppose that G has a clique C of size j. If the vertices of C are ordered $c_1 \ldots c_j$, it is clear that the value associated with vertex c_i is less than the value C($c_i + 1$). Thus, the maximum C value over all vertices is at least as large as the maximum clique, and if the algorithm returns a value it actually finds a clique of this size. Therefore, if the input is not a comparability graph, the algorithm will either answer that G is not a comparability graph, or find a maximum clique in G. All steps of the algorithm can be implemented easily (once we are given the linear extension L) in linear time, despite the fact that we cannot recognize comparability graphs in linear time.

It is interesting to note that we cannot say that this algorithm is solving the problem on a specific larger class of graphs. Consider a 5-cycle 1,2,3,4,5,1 plus vertex 6 adjacent to exactly 1 and 2. This graph has no transitive orientation, and the algorithm may come up with the orientation of figure ?. The linear extensions 563412 and 653412 are both consistent with this orientation, and each could be produced using the same algorithm by renaming vertices. If 536412 is used, the algorithm would find the candidate clique 5,1,2, determine that this is not a clique, and answer that G is not a comparability graph. If 653412 is used as a linear extension, the algorithm answers that 6,1,2 is a largest clique in G. Thus, depending on choices made by the algorithm, the algorithm may or may not find the maximum clique on this graph.

The issue may become more clear if we outline another problem which is optimized more quickly than the class can be recognized. We outline an algorithm from [101] for finding the maximum independent set in a circular-arc graph in linear time. Note that circular-arc graphs can be recognized in polynomial time, but that this algorithm will solve the optimization algorithm faster than any known recognition algorithm. We present only part of the algorithm here; the key point is that we eliminate vertices which do not change the size of the maximum independent set, but may change a graph from being outside the class to being in the graph class.

The algorithm selects any minimum degree vertex v from the graph. We remove any neighbor w of v such that N(v) is a subset of N(w), since w could be replaced by v in any maximum independent set, whether or not G is a circular-arc graph. It then checks a number of conditions which must hold for G if G is a circular-arc graph. These include testing whether the graph induced by V - N(v) is an interval graph. If the set of conditions, all of which are testable in linear time, are satisfied, the algorithm takes O(n+m) time, and either finds the maximum independent set on the remaining graph whether or determines that the remaining graph is not a circular-arc graph. Details of the algorithm can be found in [101].

Consider how this algorithm would behave if the input is the cycle 1,2,3,4,5,6,7,1 plus vertices x adjacent to 1,2,4,7 and y adjacent to 2,3,4,5,6,7. If 1 is chosen as a minimum degree vertex, G - N(1) is an interval graph; in fact, x will be removed and the remaining graph is a circular-arc graph, so the algorithm will find the maximum independent set in G. If 6 is chosen as the minimum degree vertex, G - N(6) is not an interval graph (it contains the chordless cycle 4,x,2,3), so the algorithm will answer that G is not a circular-arc graph. Both answers are acceptable, but the algorithm may or may not find the maximum independent set when this graph is given as input.

We now present an example of a problem which is well known in the literature, although it is not usually recognized that the optimization problem is solved much more quickly than the recognition problem on this class of graphs. We will discuss the matching problem on claw-free graphs. Recognition of claw-free graphs is discussed in chapter ?; no algorithm seems to be known other than those which can be derived easily from the definition. The best recognition algorithm seems to take n matrix multiplications.

For general graphs, matching takes ? time. There is a very simple algorithm for finding a maximum matching on claw-free graphs, however. Construct a breadth first search tree T for G; we will process T in postorder. Consider the set of children of any node p in T which is not the root. The children must form a clique, or p, parent(p), and any two nonadjacent children of p are a claw. Thus, we match consecutive children at such a level; if 1 child remains, we match the child with p. If any of these consecutive children are nonadjacent, we declare that G is not claw-free.

We are left with a set of children of the root r. Take the first three children; there must be some edge between a pair, or they form a claw with r; match any which have an edge. Repeat this on any 3 remaining children, until we have a root and at most 2 children. If any children remain, match the root with 1 child. Thus, we match all but at most 1 vertex. Since each new matching edge is found in constant time, this algorithm takes $\mathrm{O}(n)$ time once the BFS tree has been constructed. If we find a matching with at most 1 vertex unmatched, the matching is clearly optimal, whether or not the original graph is claw-free; otherwise, we will declare that the input is not claw-free. Note that if G is not claw-free, our result may depend on the particular breadth first tree which is constructed for G.

Thus, the frequently cited theorem that every claw-free graph on an even number of vertices has a perfect matching is also an example of a robust linear time optimization problem on a class which cannot be recognized in linear time.

An interesting robust algorithm known as struction was developed by Hammer, Mahadev, and DeWerra [89] for solving independent set on graphs without any of the forbidden subgraphs shown in figure ?; such graphs are called CAN-free (for claw, antenna, net free). In fact, their algorithm solves the independent set on any input graph, since it can take any input graph G and produce an output graph G' such that the size of the maximum independent set in G is equal to the size of the maximum independent set in $G' + 1$. Since they have not shown that P = NP, it is clear that in general the size of G' can be much larger than the size of G. They were able to show that if G does not contain any of the forbidden subgraphs, then G' is no larger than G and is CAN-free. Since each transformation takes $\mathrm{O}(n^2)$ time, where n is the number of vertices in the output graph, this implies an $\mathrm{O}(n^3)$ algorithm for the independent set problem on CAN-free graphs. If we do not know whether G is CAN-free, we can simply run this algorithm; if the graph ever grows during a transformation, we know it is not CAN-free, and otherwise we solve the independent set problem correctly on G. In fact, a graph with a forbidden subgraph may be transformed into a graph with no forbidden subgraphs by the algorithm, so it sometimes optimizes other graphs as well; there seems to be no obvious method for testing whether G is in the class besides trying all k vertex subgraphs. The algorithm above is interesting in that a graph class is defined on the basis of an algorithm, as well as the natural interest in a general algorithm for solving the independent set problem. The technique was later refined to solve the independent set problem in claw-free net-free graphs [90].

14.1.1 Cliques in Unit Disk Graphs

We now present an algorithm which takes a problem which was known to be solved in polynomial time if a model was given as input, and modifies the algorithm to be a robust algorithm. The author first became of this problem through the web-based survey on intersection graphs written by Prisner [].

G is a unit disk graph if G is the intersection graph of a set of circles with same radius. The first paper dealing with cliques on unit disk graphs was by Clark, Colbourn, and Johnson []; they give a method for solving the problem if the centers and radii of each circle was known. However, it has been shown since that time that it is NP-hard to recognize unit disk graphs (membership in NP still seems to be open), so one might think that an algorithm which uses the model would not help in designing

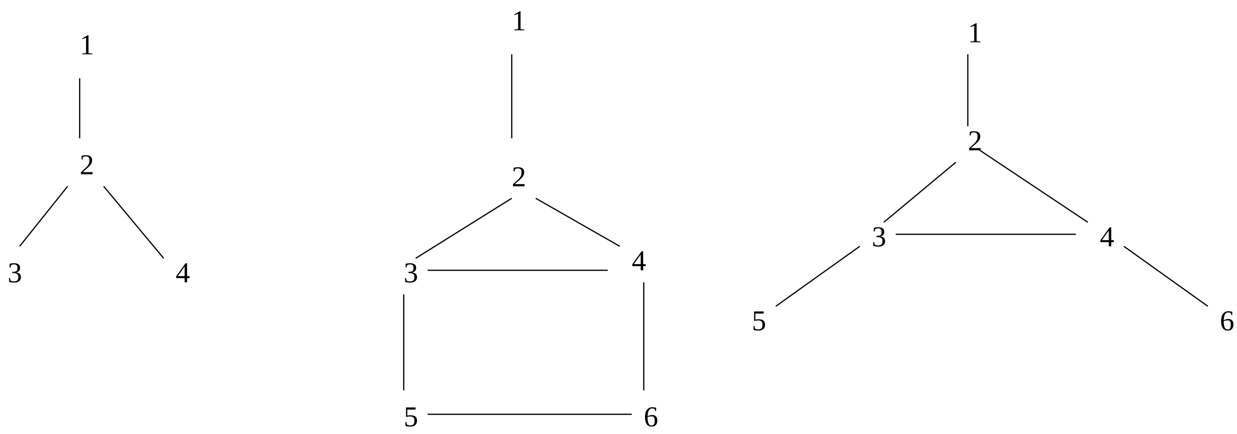


Figure 33: Claw, Antenna, and Net

a robust algorithm which takes the adjacency matrix as input. We will see that by focussing on certain steps of the algorithm which lead to efficient computation of the clique problem for this class, we are able to remove the dependence on the model.

Clark, Colbourn, and Johnson's algorithm takes the model as input, and examines every pair of adjacent vertices x,y. Let $d(x, y)$ be the distance between the centers of circles x and y. For each of x,y, draw a circle with this center and radius $d(x, y)$ as in the figure below. Let $S(x, y)$ be the set of vertices with centers in both of these circles; we note that this will not include all common neighbors of x and y.

It is not hard to see that $S(x, y)$ induces a graph which can be covered by two cliques, i.e. a cobipartite graph. If we divide the area of common intersection by a line between x and y, the distance between any of the centers in one section can be no larger than the distance between the center of x and the center of y, implying that the set forms a clique. Thus, finding the maximum clique in $S(x, y)$ corresponds to finding the maximum clique in a cobipartite graph. This is equivalent to finding the maximum independent set in a bipartite graph; it is well known that this can be done as quickly as bipartite matching [], which is $O(n^{2.5})$.

However, since $S(x, y)$ does not include all common neighbors of x and y, it is not immediately apparent that we can use this to find the maximum clique. Let C be a maximum clique, and let x,y be a pair of centers of maximum distance in C. It is easy to see that C is the maximum clique in $S(x, y)$; thus the size of the maximum clique found in all $S(x, y)$'s will be the size of the maximum clique in G.

We now modify the solution to be a robust algorithm for the maximum clique problem on unit disk graphs even if the model is not given as input. We focus on what makes the above solution polynomial; we want to search for maximum cliques in cobipartite subgraphs in such a way that we are guaranteed that one of these contains the maximum clique in G.

An ordering of the edges of G is called a cobipartite edge elimination ordering if the following property is fulfilled. Let G_i be the graph formed by including only edges which occur after edge i in the ordering.

for all $e_i = (x, y)$, the common neighbors of x and y in G_i induce a cobpartite graph in G (give example).

We will show that every unit disk graph has a cobipartite edge elimination order, and that we can find such an ordering (if it exists) in an arbitrary graph in polynomial time. We then show that we can find a maximum clique in any graph with a cobipartite edge elimination order efficiently; together, this gives a robust polynomial algorithm for the clique problem on unit disk graphs.

First, we show that every unit disk graph has a cobipartite edge elimination order. Given a unit disk model, order the edges by decreasing distance between the corresponding centers. For any edge $e_i = (x, y)$, the neighbors of x and y in G_i are those vertices closer to x than y is, and closer to y than x is. These are exactly the vertices $S(x, y)$ in the Clark, Colbourn, Johnson algorithm, and are cobipartite in G for this reason.

Next, we show that it is possible to find a cobipartite edge elimination order in polynomial time, if such an ordering exists. We use a greedy algorithm; simply take any e such that the common neighbors of the endpoints of e induce a cobipartite graph, and place it next on the list, and delete it from the set of edges in G_i. Suppose that this fails (i.e. this halts before putting all edges in the list), but there is a valid ordering O for the edges of G. Let e be the first edge in O which is not in the output list. Then the common neighbors of e in G_i with respect to our list are a subset of the common neighbors of e when it was placed in O (since all earlier edges are also in our list); thus, our algorithm would not have terminated at this point.

Finally, it is easy to solve the maximum clique problem in polynomial time, if a cobipartite edge elimination order is given. We simply take the maximum clique over all the cobipartite $\mathrm{N}(e_i)$ with respect to the ordering. Let C be the maximum clique in G, and let (x, y) be the first edge of C in the order. Then C is a subset of the joint neighbors of x and y, and will be found at this step.

Therefore, we have modified the algorithm given the model to solve the problem robustly for unit disk graphs. If the input graph does not have a valid ordering, we can stop and declare that the input is not a unit disk graph. If the input has such an ordering, we will solve the clique problem correctly (whether or not the input is in the class); since we solve the problem correctly on all unit disk graphs and give correct answers on all graphs not in the class, this algorithm is robust.

14.2 Problems Helped by a Representation

This section studies problems which can be solved more efficiently if one is given a good representation of a graph class than the best known algorithms for the problem if the graph is given in adjacency list form. We should mention that many problems have faster algorithms if one is given the representation. For example, the independent set problem for circular-arc graphs studied in the previous chapter can be solved in O(n) time if the input is in the form of a circularly ordered list of arc endpoints [99, 78, 122]. We are primarily concerned here with problems where the algorithm is polynomial if the representation is given, but no polynomial algorithm is known if the input is in the class but is given in adjacency list form. The major open problem, stated as problem ? in chapter ?, is to find a problem which is not solvable in polynomial time on a class if given in adjacency matrix form, but is polynomial on the class if given the good representation scheme.

It is surprisingly difficult to find problems which are hard if the representation is not given, but easy if it is given. for resolving the open problem. In fact, the author knows of no natural problem on a graph class such that the problem is solvable in polynomial time if the graphs are given in the form of a good representation, and we know that the representation is NP-hard to construct, except for cases in which the problem is also solvable in polynomial time without the representation.

We will discuss in more detail two problems which can currently be solved much more efficiently if a good representation is given. The first is the clique problem on visibility graphs, which can be solved in polynomial time if the visibility polygon is given; in this case, we do not yet know either whether the problem is difficult if given in adjacency matrix form, or whether finding the visibility polygon is NP-hard. The second problem we deal with is coloring perfectly orderable graphs; here, the problem is very easy if given a 'representation'; it is known to be polynomial, though in a somewhat unsatisfactory sense, if the input is given in adjacency matrix form. Finding the representation is known to be NP-complete for perfectly orderable graphs.

Note: or in VG chapter?

We will now discuss the clique problem on visibility graphs. First let us note that the Hamilton circuit problem on visibility graphs could also be said to be a candidate problem, but in this case the problem is trivial if the representation is given, in the sense that the answer to the Hamilton circuit problem is always yes.

As we discussed in chapter ?, visibility graph recognition is a major open problem; in fact, the problem is not known to be in NP. The clique problem on visibility graphs was first solved in [52]. We will describe the algorithm quite differently, however, since we intend to focus on what is possible if the model is not given, whereas that paper was written with the assumption that the polygon was part of the input.

It is possible to compute the maximum clique which contains point p as its leftmost endpoint; since the maximum clique must have some leftmost endpoint, this gives a polynomial algorithm for the clique problem. We will now describe the basic property which allows us to compute the maximum clique containing p as leftmost endpoint in polynomial time.

Order the vertices of N(p) which are to the right of p in the same order as they appear in the polygon, assuming that the polygon starts ast p. Suppose that there are $p_1 < p_2 < p_3$ in this ordering, such that these three vertices form a triangle in the visibility graph. For any vertex of N(p) $p_0 < p_1$ which is adjacent to p_1 and p_2, simple geometric arguments can be used to show that p_0 must be visible from p_3. This implies that if we want to determine whether a clique can be extended to include a new vertex with higher number in the ordering, it is sufficient to keep the two highest numbered vertices in the clique.

Thus, we get a natural dynamic programming algorithm to find the largest clique containing p as leftmost vertex. Let C_{ij} be the size of the maximum clique containing i and j as the second to last and last vertices of the ordering. Vertices are added to the set under consideration in order; let k be the next vertex added. For all $i < j < k$, if k is adjacent to both i and j, then one candidate for C_jk is $C_{ij} + 1$; if this is larger than the current value of C_jk, we make it the current value. It is easy to keep track of the clique itself, rather than just the size of the clique, without affecting any time bounds (exercise ?).

Now let us think of what is possible if the input is given in adjacency matrix form. If G is really a visibility graph, there must be some vertex v such that the neighbors of v can be ordered so that if $n_1 < n_2 < n_3$ are adjacent to each other, the neighbors of n_1 and n_2 which precede n_1 in the ordering are all neighbors of n_3. Let us call such an ordering of the vertices a triangle-extendible?? ordering. If you are given a triangle-extendible ordering of the neighbors of v, then you can calculate the maximum clique containing v, whether or not G is a visibility graph. Thus, you can convince me of the size of the maximum clique in a visibility graph in polynomial time; successively show me triangle-extendable orderings of the neighborhood of some vertex v, calculate the maximum clique containing v, and eliminate v from further consideration. Thus, even though we cannot be convinced that G is a visibility graph, we can be convinced of the size of the maximum clique in a visibility graph; the clique problem is not likely to be NP-complete for this class even if input is given in adjacency matrix form. Of course, if we could find a triangle-extendible ordering for a graph G in polynomial time, we could find a robust optimization problem for maximum clique in visibility graphs in polynomial time, so we pose the following open problem.

Open Problem: Can you determine whether a set of vertices has a triangle-extendible ordering in polynomial time?

Note that triangle-extendible orderings can also be viewed as related to transitive orientations; a graph is transitively orderable (i.e., a comparability graph) if and only if the there is an ordering such that for each edge $i \to j$ where $j > i$, N(i) contains N(j). Thus, transitive orientation allows each edge which goes forward to extend to all possible cliques, while the triangle-extendible orderings allow each triangle to extend to a clique.

Note that it is only the neighborhood of some vertex which has a triangle-extendible ordering, rather than the graph itself. It also interesting to study the class of graphs which have a triangle-extendible ordering for the entire vertex set; this class includes comparability graphs. Of course, clique is solvable on k-clique extendible ordered graphs for any fixed k given the ordering; if x is adjacent to all vertices of an earlier k-clique K then x is adjacent to all joint neighbors of K which precede K in the ordering. Many variants are possible; we might consider the k-extendibility of G as a graph theoretic parameter, for example. Resolving the question of whether the clique problem is solvable on triangle-extendible graphs given only the adjacency matrix would be interesting, whether the problem proves to be NP-complete or polynomial. If it is polynomial, we have a very large class for which the clique problem can be solved efficiently, and get a clique algorithm for visibility graphs even if the model is not part of the input. On the othe hand, if it is NP-complete, we have found a problem which is solvable given a 'representation' but is NP-complete if none is given.

(k-extendibility of random graphs; for which k expected? related to proof G has no clique of size ¿ k)

It is also interesting to think of what is possible if only the ordering of endpoints in the polygon is given, rather than a complete model. In terms of size of the representation, of course, we are much more comfortable with being given the ordering than being given the entire polygon. Recall that for recognizing visibility graphs, we do not know how to use a known order of endpoints to help us with the recognition or construction problem. However, being given the order of endpoints is entirely sufficient to solve the clique problem in visibility graphs. There must be some vertex v such that the ordering on neighbors of v is a triangle-extendible ordering; we then remove this vertex and continue. Since we can easily test whether an ordering is triangle-extendible in polynomial time, this gives us a polynomial algorithm for the clique problem on visibility graphs if only the endpoint ordering is known.

A further restriction on triangle-extendible orderings is given in [1]. An ordering is inversion-free if for every pair of edges (i, j), (x, y) with $i < x < j < y$, (i, y) is also an edge. The neighborhood of the leftmost vertex of the polygon has an inversion-complete ordering as a result of the theorem presented in the next paragraph; this implies that finding an inversion-complete ordering in polynomial time would be sufficient for recognizing visibility graphs even if the model is not given as input.

A polygon is a convex fan if there is a convex vertex which sees all other vertices of the polygon. An ordered cycle $c_1 ... c_k$ is a cycle in which each c_i comes after c_{i-1} with respect to an order. A Hamilton cycle of a graph G is the border of a visibility graph of a convex fan if and only if it is inversion-complete and contains no ordered cycle of length > 3 [1].

Inversion complete graphs are an interesting class for representation; there are $2^{\Theta(nlog^2 n)}$ inversion-complete graphs, and there is a representation using O($log^2 n$) bits per vertex, as shown below.

We give a representation of any inversion-complete graph. Assume that we are given an inversion-complete order, with vertices numbered from 1 to n in the order. Successively divide the order in two, and for each x, store the 'closest' vertex in the other half which is adjacent to x (that is, the earliest vertex if x is in the first half and the latest if x is in the second half). Recursive splitting wil give us log$n$ numbers stored at each vertex. To determine whether x and y are adjacent, we find the first split which separates x from y (this can be determined from the value of x and y), and look at the numbers stored at x and y for this split. Assuming that x is in the

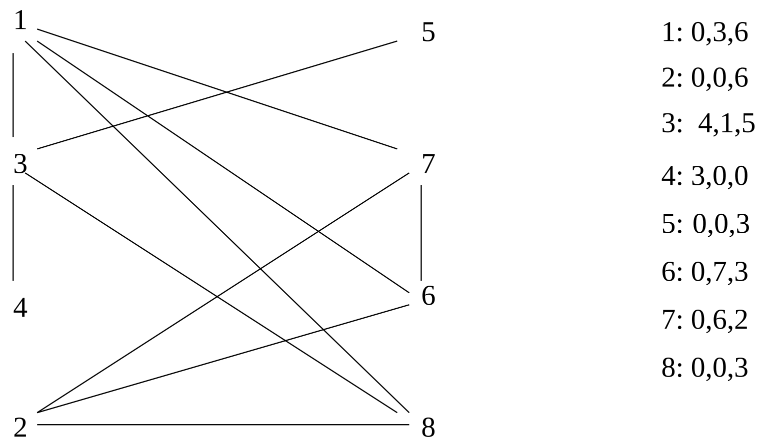


Figure 34: Graph with Inversion-Free Representation

first half of the split, x and y are adjacent iff x is stored at y, y is stored at x, or x stores a number less than y while y stores a number greater than x; the fact that the ordering is inversion complete implies an edge in the latter case.

An example of this storage scheme is given in figure ?, where the inversion-complete ordering is 1,2,3,4,5,6,7,8. We use 0 to mean that there is no edge to the other set. For example, 3 is given the tuple 4,1,5; this means that 4 is the first neighbor of 3 from the set 4, 1 is the last neighbor of 3 from the set 1,2, and 5 is the first neighbor of 3 from the set 5,6,7,8. We can deduce that 2 is adjacent to 8 by the presence of the edges (2,6) and (3,8) which are given directly in the representation.

The question as to whether this is a generalized implicit representation is open, since the counting problem is not resolved for inversion-complete graphs. A natural way to try to reduce the size of the representation is to use a notion of 'nonredundancy' similar to that used in the chapter dealing with chordal bipartite graphs; an edge (x, y) of an inversion-complete ordered graph is redundant if there is a pair of vertices u, v such that $x < u < v < y$, with x adjacent to v and u adjacent to y. For example, in figure ?, the edge (1,6) appears in our representation but is redundant; it can be deduced from the existence of edges (1,3) and (2,6).

The argument above shows that there are O($n$log$n$) nonredundant edges; together with an argument from exercise ? showing that G can be reconstructed from the set of nonredundant edges, any improvement on the upper bound on number of such edges would lead to a better upper bound on the number of inversion-complete graphs. Unfortunately, exercise ? shows that there are inversion-complete graphs with Ω($n$log$n$) nonredundant edges, so this approach will not close the counting problem.

We close the discussion by posing the obvious major open problem, which we feel should be answered affirmatively.

Open Problem: Is there a robust algorithm for the clique problem on visibility graphs?

We now discuss the problem of coloring a perfectly orderable graph. Perfectly orderable graphs arose as a result of an interesting observation of Chvatal's (check). Papers on subclasses of perfect graphs work on finding the special technique which allows the particular class to be optimized effectively. For example, a perfect elimination ordering can be found for a chordal graph, and this allows many optimization problems such as coloring to be solved in a straightforward, greedy manner if given a perfect elimination scheme. Similarly, one can color a comparability graph easily if one is given a linear extension of the transitive orientation. Chvatal thought of generalizing this notion of coloring greedily from a simple orering scheme as far as possible, while keeping the graph class hereditary and perfect.

A greedy coloring is one of the simplest and most natural approaches to graph coloring. Color vertices in some order; let x be the next vertex to be colored. Assign x to have the smallest color which is not used by any previously colored neighbor of x. Given an ordering, one can perform greedy coloring in $O(n+m)$ time; in general, greedy coloring can have very bad performance, using $\Theta(n)$ colors for a 3-colorable graph (even 2 for greedy?).

G is perfectly orderable if there is an ordering of the vertex set such that for every induced subgraph H of G, the number of colors used by the greedy algorithm, with order of vertices chosen according to the ordering scheme, is equal to the size of the maximum clique in H. Subclasses of perfectly orderable graphs include chordal graphs, comparability graphs, (name some more).

We will call the ordering scheme associated with a perfectly orientable graph G, along with the adjacency matrix for G, a representation of G. This is certainly stretching somewhat the notion of a representation of a graph class, but it allows the introduction of some interesting problems. Given this representation, we can solve the coloring problem on perfectly orderable graphs efficiently; simply use the greedy coloring algorithm with this ordering.

It is not immediately clear that this solution can be trusted; what would happen if the representation was not correct, that is, the ordering is not a perfect ordering? Note that this sort of question does not arise with, for example, a representation of a boxicity 2 graph; the difference may have something to do with why we are less comfortable with calling this a representation of the graph. Nevertheless, we can verify that an ordering is a perfect ordering efficiently, as described below, and thus we can trust the result of a coloring algorithm if we are given a perfect ordering.

The characterization of a perfect ordering follows the pattern of making a simple observation on what must be true for a perfect ordering, and proving that this is all that can stop an ordering from being perfect. Suppose that there is a P_4 1,2,3,4 such that the ordering places 1 before 2 and 4 before 3. If we color the P_4 according to such an ordering, vertices 1 and 4 will get color 1, either 2 or 3 (whichever is earlier in the ordering) gets colored 2, and the other vertex receives color 3. This is clearly not an optimal coloring, so no P_4 1,2,3,4 can have 1 before 2 and 4 before 3 in the ordering. [32] shows that an ordering is a perfect ordering iff we can avoid this ordering on all P_4s; this gives us an efficient test for whether an order of the vertices is a perfect order. However, we cannot find a perfect ordering efficiently, since recognizing perfectly orderable graphs is NP-complete [125].

There is a sense in which we can color perfectly orderable graphs efficiently. There is a polynomial time algorithm for coloring perfect graphs, and perfectly orderable graphs are a subclass of perfect graphs. In fact, there is a robust algorithm for coloring perfect graphs; the algorithm will color a perfect graph optimally, and if the input is imperfect will either color it optimally or declare that the graph is not perfect; this can be done despite the fact that there is no polynomial time algorithm known for recognizing perfect graphs. The algorithm for coloring perfect graphs can be found in [84]; depending on one's perspective, it is one of the most wonderful and/or horrible algorithms ever designed in graph algorithms. This algorithm was a major achievement, using deep techniques to solve coloring on a very large class of graphs which still cannot be recognized efficiently. However, the time bound is unusually large; the algorithm involves nested calls of ellipsoid type algorithms developed for solving linear programming problems. This author is also amazed by some of the calculations the algorithm performs; for example, it is hard for me to understand how eigenvalues arise when we are dealing with problems such as clique or coloring. Thus, while the polynomial algorithm is very interesting, it does not resolve the question entirely; we would also like an algorithm which is more combinatorial in nature, as well as having better time bounds. We would like such an algorithm for general perfect graphs, but that would involve a major breakthrough. Given that the problem can be solved efficiently given our representation of perfectly orderable graphs, we pose the following problem.

Open problem: Design an efficient, combinatorial algorithm to color perfectly orderable graphs, if the input is in adjacency matrix form.

As far as the author knows, there no work has been done on the question of whether the algorithm of [84] itself can be simplified to run more efficiently on subclasses of perfect graphs, such as perfectly orderable graphs. The analogy would be that the simplex algorithm for linear programming, which is exponential in the worst case, can be shown to run in polynomial time for various types of linear programs which correspond to natural graph theoretic questions [].

A new open problem comes from graphs with cliquewidth k; this parameter is discussed in chapter ?. Specifically, problems which can be described using a variant of second order monadic logic can be solved in polynomial time for graphs of cliquewidth k, if the construction sequence is given as part of the input. Problems of this type include such NP-complete problems as max cut, ?????? However, there is no known polynomial algorithm for finding the construction sequence (see open problem ?). Therefore, we have an open problem.

Open Problem: Find a robust polynomial time algorithm for solving the ??? problems on graphs with cliquewidth k for fixed k.

approxs (ask Breu re problem) (clique, lines in plane)

(Russian matrix problem)

(Keil trapezoid problem) (MCCC conf q)

There are many graph classes which have nice representations which are difficult to find. Examples of graph classes for which the recognition problem is NP-complete, co-NP-complete, or more difficult include graphs with boxicity k for $k \geq 2$, circle intersection graphs, intersection graphs of line segments either in general position or isooriented on the plane, partial orders of dimension $d > 2$; the list could go on and on. Sometimes, the representations allow efficient algorithms for optimization problems, but it seems that that these usually can be transformed into algorithms which solve the problem efficiently even if the graphs are given in adjacency matrix form.

Q to self or open? domination, dimension 3 poset

14.3 Complexity Issues

One reason problems of this form are difficult to find may be that if the class has an easily verifiable representation, and can be solved in polynomial time with the representation given, we have a short proof of either a yes or no answer to the question (show me the representation, and run the algorithm on it). Thus, given standard complexity assumptions, we will not be able to find a reduction from satisfiability which produces a graph in the class with output yes if the formula is satisfiable, and produces a graph in the class with output no if the formula is not satisfiable; such a reduction would imply that NP = co-NP. This has been the standard notion of being NP-complete on a graph class. We call problems which are difficult in this sense classically NP-complete on a class of graphs (note: needs name suggestion!).

Even with respect to classical NP-completeness, graph classes with nice models may be interesting; this seems to be a way to find fairly natural candidates for problems which are not polynomially solvable, but not NP-complete (since solvability given a model shows that the answers yes and no can both be verified); relatively few such candidate problems are known.

However, we need a new notion of NP-completeness to correspond to our new definition of being robustly solvable. A problem is robust-NP-complete on the class C if there is a polynomial reduction from the satisfiability problem which produces a graph in the class with answer yes if the formula is satisfiable, and produces a graph with answer no (which may or may not be in the class) if the formula is not satisfiable. Thus, a robust algorithm would have to answer yes if the answer to the original problem was yes, and would give some other answer (either no or G is not in the class) if the answer to the original problem is no. Thus, a robust algorithm for such a problem would imply that P equals NP.

We demonstrate this new form of reduction by showing that there is no robust algorithm for the independent set problem on well covered graphs unless P=NP.

We reduce from the independent set problem to the independent set problem on well covered graphs. Let G, k be the input to the independent set problem. We create G' as follows.

Create k copies $v_1, v_2, ..., v_k$ for each vertex v. Let v_i be adjacent to w_j if $i = j$, $v = w$, or v and w are adjacent in G.

Add new vertices x_{ij} for all j not equal to i. The intent of these new vertices is to supplement any independent set to be a maximal independent set of size k-1. Add edges from x_{ij} to all x_{kl} with i not equal to j, and to all v_i and v_j. The meaning of x_ij is as a replacement for vertex j in an independent set which includes no v_i or replacement for v_i.

Suppose that G has an independent set of size k. Then there is an independent set of size k in G', taking the first copy of the 1st vertex of the independent set, the second copy of the next, etc. Suppose that G has no independent set of size k. Then every maximal independent set of G has size k-1. If any vertex x_{ij} is used, then no vertex with first subscript i can be used; however, we will always be able to supplement any other set to a set of size k-1 by using the appropriate x_ih whenever nothing of the form v_h was chosen. If no vertex x_{ij} is used, the vertices correspond to an independent set in G. If fewer than k-1 vertices are in this set, they can be augmented by some x_{ij} where no v_i or v_j is used.

Suppose that we had a robust algorithm for the independent set problem on well covered graphs. We take our independent set problem on G, and produce G' as described. If our robust algorithm answers that the independent set in G' has size k or that G' is not well covered, then we know that G has an independent set of size k. If the algorithm answers that G' has an independent set of size k-1, we answer that G has no independent set of size k.

Thus, a robust algorithm for the independent set problem on well covered graphs is not possible unless P= NP; this may seem surprising in light of how simple the problem is if we are given a promise that the input is well-covered.

As an aside, we note that the reduction above can also be used to show that recognizing well-covered graphs is co-NP-complete, which was already proved using a different reduction [].

We also need a complexity term for problems which are difficult if no model is given, but easy if a model is given. We say that a problem is model-NP-complete for a graph class C if there is a polynomial reduction from satisfiability which produces a model of a graph in the class with answer yes if the formula is satisfiable, and a model of a graph in the class with answer no if the formula is not satisfiable.

Thus, our open question of chapter ? on problems which are easy if given a model and hard if not becomes the following:

Find a natural model for a graph class and problem such that the problem is classically NP-complete on the class, but is not model-NP-complete on the class unless P = NP.

We note that we will (presumably) need a different notion of reduction to prove any result of this form. If a reduction always produces a graph in the class, and there is an algorithm which solves the problem given a model, then there is also a proof for no instances; you show me the model, and show me the result of running the algorithm on the model. Since proofs of a no instance for an NP-complete problem would imply that NP=co-NP, reductions of this form do not seem plausible. Nevertheless, being given the solution to the NP-hard problem of constructing a model should help in solving some problems; the difficulty seems to be in proving that the problem cannot be solved without the model.

14.4 Exercises

1) Show that the number of well covered graphs (see ?) is $2^{\Theta(n^2)}$.

2) Show clique in P for circle intersection graphs. ?????

3) The fact that the independent set problem on circular-arc graphs can be solved more efficiently than recognition may not be surprising if viewed in the following light. Design a simple O(n+m) algorithm to find an independent set which has cardinality at most 1 smaller than the largest independent set in G, if G is a circular-arc graph. Prove that your algorithm is a robust algorithm

4) The following subproblem arises in the independent set problem for circular-arc graphs if G - N(v) is a clique. Design an O(n+m) algorithm to determine whether a circular-arc graph contains an independent set of size 3.

4) Show that for any graph G, there is an ordering of G such that greedy coloring with this ordering produces an optimal coloring.

5) Greedy coloring can perform very badly indeed if the ordering is not good. Show that there are orderings 3(2?)-colorable graphs for which the number of colors used is $\Omega(n)$.

6) Show cliques of EPT graph need not share common edge of T.

7) Show size of tree for EPT can be O(n) (check).

8) Show EPT of star can assume all paths 2 edges

9) Independent set on comparability graphs is somewhat harder than the clique problem. The algorithm, first presented in [] for partial orders, works as follows. Transitively orient G; call the result G'. Create a bipartite graph B(G') by creating two vertices v_in, v_out for each vertex v, and add an edge from v_out to w_in if and only if $v \to w$ in G.

A maximum independent set in a graph is exactly equal to the vertices not contained in a minimum vertex cover, and the minimum vertex cover in a bipartite graph can be found in linear time from a maximum matching []; thus, the following problems show that independent set in comparability graphs is no harder than bipartite matching. We note that it is easy to see that determining the size of the maximum independent set is as hard as determining the size of a maximum matching, since every bipartite graph is a comparability graph (restate this!)

a) Prove that if G is a comparability graph, the optimal vertex cover in B(G') will never use both v_in and v_out.

b) Prove that the optimal vertex cover in B(G') is an optimal vertex cover in G.

10) A graph G is brittle if in every induced subgraph of G, either there is a vertex which is not in the center of a P_4 or there is a vertex which is not at the end of a P_4.

a) Show that the number of brittle graphs is $2^{\Theta(n^2)}$.

b) Show that all brittle graphs are perfectly orderable.

11a) What is the time complexity of the algorithm for finding a maximum clique in a visibility graph given above?

11b) Show that we can get the clique itself as well as the clique size in same bound.

12) Show can solve clique on induced visibility graphs given the model (or, more restrictively, given the order of endpoints on the polygon and adjacencies; can you get around endpoints not in G?)

13a) Show that the following graph 10 vertex graph is not triangle-extendible. Consider a 4-clique and a 6-clique, where the vertices of the 6-clique are nonadjacent exactly to different pairs of vertices from the 4-clique. Richard Borie wrote a computer program which found this and other 10 vertex graphs which are not triangle-extendible, and proved that graph with fewer than 10 vertices is triangle-extendible.

13b) Show that for every k ¿ 2, the graph consisting of a 2(k-1) clique and a 2(k-1) choose 2 clique, with vertices from the second clique nonadjacent to different pairs of vertices from the first clique, is not k-clique extendible.

14) Show that the class of cobipartite triangle-extendible graphs has an implicit representation.

15) Show that every inversion-complete ordering is triangle-extendible.

16) Give an example of a graph which is triangle-extendible, but not inversion-complete.

17) A convex graph is a bipartite graph with color classes X and Y in which the vertices of X can be ordered so that neighbors of every vertex y in Y occur consecutively. Show that every convex graph is inversion-free.

18) Show that any inversion-complete graph G can be reconstructed in polynomial time from the ordering, and the set of nonredundant edges for this ordering. (note to self: how fast can you make this algorithm?)

19) Show that the graph formed in the following way has $\Omega(n\log n)$ nonredundant edges. It is interesting to note that this graph is also the key to exercise ? in chapter ?, in that it can be drawn with all nonredundant edges as pseudolines which touch the outer visibility envelope.

We start with 2 nonadjacent vertices. To get a graph of size $2k$ from 2 graphs of size k, create the ordering x_1 ... x_k y_1 ... $y - k$ and add edges from each x_i to each y_i. This will give a graph G_1 with Ω $n\log n$ edges. Let G_2 be formed by adding all edges to G_1 which are necessary in order to make the result inversion complete. [4]

[4] tournament chess players may note that this graph corresponds to the games played in a Swiss

Show that all edges of G_1 are nonredundant in G_2.

20) In the definition of robust NP-completeness, explain why the following type of reduction was not allowed: Given a yes instance, the transformation must produce a graph in the class for which the answer is yes, and given a no instance the transformation must produce either a graph which is not in the class or a graph in the class with answer no.

system tournament with logn rounds and no upsets

15 Characterization and Construction

The primary focus of this book has been on representation of graphs. In this chapter, we look at some issues in the closely related construction and characterization problems which have been ignored in other general works on classes of graphs. To see the difference between the problems, note that we have completely ignored up to now classes of graphs which have $2^{\Omega(n^2)}$ members, since the adjacency matrix gives a space optimal representation for which adjacency testing can be done in constant time. This representation can easily be made generalized implicit, which has been our goal in most of the book. However, some of these classes have forms of representations which are well studied and can lead to efficient algorithms; prominent questions about some such classes are studied in this chapter.

15.1 Chordal Graphs

Chordal graphs, as discussed in chapter ?, are graphs in which every cycle of length > 3 has a chord. It is easy to see that the number of chordal graphs is $2^{\Omega(n^2)}$. Take any bipartite graph, and make one of the color classes a clique (graphs which can be partitioned into an independent set and a clique are called split graphs; see [79]). The result is chordal; a proof is left to the reader (exercise ?). Since the number of bipartite graphs is $2^{\Omega(n^2)}$, the number of split graphs and bipartite graphs is $2^{\Omega(n^2)}$.

[173, 75, 30] proved that chordal graphs are exactly equal to the intersection graphs of subtrees of a tree. This is an example of a nice characterization which is not at all good as a representation.

There are several ways to think of storing a graph as the intersection graph of subtrees of a tree T. For example, we might store T as a global structure, and the leaves of the subtree corresponding to vertex v at v. It can be shown that we can choose a host tree T which has at most n nodes, but adjacency testing will still take Ω(n) time in the worst case using this storage method. In addition, the storage method does not seem to be space optimal; we could potentially be forced to store $\Theta(n)$ numbers with each vertex v, where each number uses $\Theta(\log n)$ bits, thus using $\Theta(n^2 \log n)$ bits overall. It is not immediately obvious that there is any chordal graph such that this storage method actually requires $\Theta(n^2 \log n)$ bits; this does not seem to me to be a crucial open question to resolve, since there are clearly better ways of storing a chordal graph if optimizing space and time for adjacency testing are our goals.

Nevertheless, these trees are important to construct for solving certain optimization problems and for dealing with subclasses of chordal graphs. We have seen that they are useful in finding an implicit representation of chordal comparability graphs in chapter ?. The tree constructed is sometimes called the clique tree, since nodes of the tree correspond to maximal cliques in G. Note that the algorithm also enumerates all maximal cliques in the graph in linear time.

We now give an algorithm for constructing the clique tree of a connected chordal graph in linear time; we believe this is simpler than standard algorithms for this problem such as the one described in [79]. In particular, it is much simpler to see how this algorithm constructs the tree in linear time. The first algorithm for constructing an intersection model which clearly takes linear time is given by Hsu and Ma in [97]. Their algorithm is tied closely to lexicographic breadth first search, while the algorithm here creates a clique tree from any perfect elimination scheme. Assume that vertices are labeled from 1 to n according to their position in a perfect elimination scheme.

We construct the clique tree for the graph induced on vertices i through n for all vertices, starting with $i = n$ and ending with $i = 1$. Let $\mathrm{C}(v)$ be the clique consisting of v and all neighbors of v which appear after v in the elimination scheme. After each vertex v is processed, v is given a pointer to the clique $\mathrm{C}(v)$ in the tree. We note that vertices may be added to this clique later in the algorithm, but v will always point to a clique which contains $\mathrm{C}(v)$.

Let i be the next vertex considered, and assume we know the clique tree on the graph induced by vertices $i{+}1..n$. We need to add $\mathrm{C}(i)$ to the clique tree. Let j be the first vertex of $\mathrm{C}(i)$ in the elimination ordering, other than i itself. If the clique pointed to by j is equal to $\mathrm{C}(j)$, we add i to this clique; in other words, $\mathrm{C}(i)$ replaces $\mathrm{C}(j)$ in the tree. Otherwise, add $\mathrm{C}(i)$ as a new node of the tree. Connect $\mathrm{C}(i)$ to the tree by adding an edge from $\mathrm{C}(i)$ to the clique pointed to by j.

It is easy to make the algorithm above work in linear time, since we make one decision at each vertex, and we can decide whether the clique pointed to by j is equal to $\mathrm{C}(j)$ by simply checking the number of elements in this clique.

To see that the algorithm is correct, it is sufficient to look at two cases. Either $\mathrm{C}(j)$ is a maximal clique on the elements induced by vertices $i{+}1..n$, or it is not. If $\mathrm{C}(j)$ is a maximal clique, it clearly must be replaced by $\mathrm{C}(i)$, and the algorithm does this correctly. If $\mathrm{C}(j)$ is not a maximal clique, then $\mathrm{C}(i)$ cannot contain any maximal clique on the elements $i{+}1$ through n, and must be added as a new node. All elements of $\mathrm{C}(i)$ - i are in the clique pointed to by j, so the subtrees induced by all elements remain connected.

It is easy to incorporate the standard algorithm for verification of a perfect elimination scheme into this framework. Suppose that the input is not a perfect elimination scheme. Let i be the last vertex such that neighbors of i are not a clique, and let j be the first neighbor of i which appears after i. We will find that the neighbors of i are not contained in the set of neighbors of j. Therefore, when each vertex i is processed, we check that $\mathrm{N}(i)$ is a subset of $j \cup \mathrm{N}(j)$; if it is, we add the clique to the clique tree, and if not we declare that the input is not a perfect elimination ordering. Since this test takes $\mathrm{O}(\mathrm{N}(i))$ time (this step is described in the recognition chapter), we can either construct a clique for G or answer that the input is not a perfect elimination ordering in $\mathrm{O}(n{+}m)$ time.

15.2 Unit Interval Graphs

In chapter ?, we saw that G is a unit interval graph if and only if G is a proper interval graph, and that these graphs are exactly equal to interval graphs with no induced $K_1, 3$. We then gave two linear time recognition algorithms for the class, based on determining whether an interval model contains $K_1, 3$, and finding proper interval models. Finding unit interval models for unit interval graphs is not always stated clearly, so we provide a construction algorithm here.

In chapter ?, we described a method for turning a proper interval representation into a unit interval representation. We process intervals in the natural left to right order as given in the proper interval model, and place each new interval i just to the right of the rightmost interval which is nonadjacent to i, and to the right of any intervals which have already been placed. Although this algorithm is conceptually simple, it neglects several crucial points if we actually want to construct a representation.

Suppose, as is usually the case, that we want to place all endpoints of intervals at integers on the line. Suppose that we place the first interval a starting at position 1, and the second interval i_2 starting at position 2. If all endpoints are integers, we will have a problem if there are two nonequivalent vertices which are adjacent to i_2 but not to i_1; both will have to be placed between the right endpoint of i_1 and the right endpoint of i_2. The most natural way to solve this problem is to place any new endpoint halfway between the bracketing endpoints. However, to make all endpoints integers, this would require the length of an interval to be exponential. This destroys the notion of an efficient representation. To get a space efficient representation, we need to find a representation in which intervals have polynomial size, so that the number of bits used for each interval is $O(logn)$. If integer sizes are exponential, we cannot truly call the running time of the algorithm linear, since manipulating numbers in the range 1 to 2^n requires $\Theta(n)$ time.

We now present an algorithm which constructs a unit interval model from a proper interval model which has the property that all interval endpoints are integers in the range 0 to $O(n^2)$. The running time of an algorithm constructing this representation is $O(m+n)$. The proof of the correctness of the algorithm can be found in [39]. It is also possible to construct a representation which uses intervals of the minimum possible length; see [126] for a description of such an algorithm.

In the first step of the algorithm, we find the leftmost interval. Choose an arbitrary vertex u, and perform breadth first search from u. Let z be a vertex from the last level of the breadth first search which has minimum possible degree. Make z the leftmost interval.

We now do a breadth first search from z. Order vertices from $1..n$ such that all vertices at level i have lower number than vertices at level $i+1$, and vertices in a single level are ordered in increasing order of (neighbors in next level - neighbors in previous level). This step gives a proper interval ordering, if such an ordering exists, by extending each endpoint to the right until we reach a nonneighbor.

Construct a directed tree on the vertices by adding an edge to each vertex v other

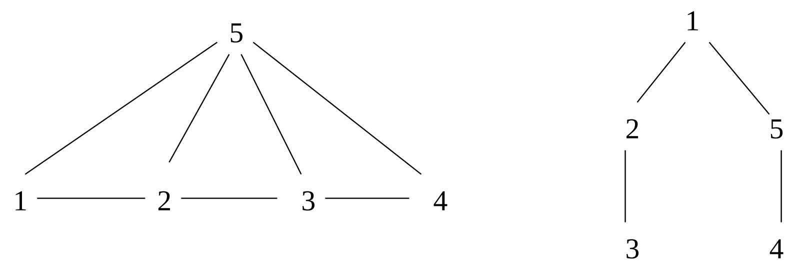


Figure 35: Unit Interval Model Construction

than z from the earliest neighbor of v in the ordering. Construct a second ordering of the vertices called postorder by performing a postorder traversal where smaller numbered children are visited first; i.e., if w has children x and y with $x < y$, we go down the path from w to x before we go from w to y, and consequently postorder(x) $<$ postorder(y).

The unit interval model is now constructed as follows. If v is a vertex from level k of the breadth first search, make the left endpoint of $v = nk +$ postorder(v) and the right endpoint of $v = nk +$ postorder(v) $+ n$. Clearly, all intervals have length n.

In the graph of figure ?, our initial breadth first search may choose 1 as the leftmost vertex z. The proper interval ordering is 1,2,5,3,4; note that 5 has more neighbors at level two than 2, and 3 has more neighbors at level 1 than 4. The tree produced is shown in figure ?. The postorder traversal creates the order 3,2,4,5,1. The left endpoints in the model are placed as follows
1: 5 2: 7 3: 11 4: 13 5: 9
and all intervals have length 5.

The paper [39] shows that if G is a unit interval graph, this is a valid unit interval model. It is easy to determine whether the model is valid in O(n+m) time (exercise ?), so this gives an efficient method for constructing a unit interval model which is an implicit representation.

15.3 Unit Circular-Arc Graphs

In chapter ?, we looked at the recognition problem for unit circular-arc graphs. These graphs were recognized by constructing a proper circular-arc model, and checking whether the model had an obstruction which Tucker [170] proved exists if and only if G is not a unit circular-arc graph. Tucker's proof actually constructs a unit circular-arc model, and thus gives an alternative circular-arc recognition algorithm whcih we will outline very briefly.

The circumference of the circle is set to a length between the lower and upper bounds dictated by a minimal (with respect to n/k) (n,k)-circuit and maximal (with respect to r/l) (r,l) independent set; these terms are discussed in the recognition chapter for this class.

The algorithm works by successively fixing one more arc A to be unit length, while maintaining a proper circular-arc model. The process involves shrinking or lengthening A; to maintain the proper model, other arcs are lengthened or shortened together with A. However, arcs which have already been set to be unit length must maintain unit length; if such an arc must be modified to maintain the proper circular-arc model, it is rotated rather than shrunk or expanded. The rotation of a unit arc may cause other unit arcs to be rotated; fundamentally, we end up rotating (i, j)-circuits and/or (g, h) independent sets, and the algorithm works because the circumference can handle the bounds forced by any such circuit or independent set.

Tucker does not worry about the size of integers in his construction, since it was used primarily as a tool for showing that any proper circular-arc graph which did not contain an obstruction as in chapter ??? is a unit circular-arc graph. This makes analyzing the time complexity of Tucker's algorithm tricky, since manipulation of large integers cannot be assumed to take constant time; sometimes arcs can be handled simply as an ordered list of endpoints, but at other times we must perform tests for length condtions. Therefore, we pose the following open problem.

Open Problem: Is it always possible to construct a unit (in the sense that all sizes are equal) circular-model with endpoints corresponding to integers of polynomial size? If so, how efficiently can such a model be constructed?

15.4 Construction Problem for NP-complete Classes

In my opinion, there is a type of problem which has been unjustly ignored in the literature. Suppose that a class of graphs with a nice form of representation is studied, and the recognition problem proves to be NP-complete for the class. Typically, this does not stop research on purely graph theoretical questions on the class, as the large body of work on such classes as partial order dimension or interval number shows. However, there is very little work on algorithmic problems on such classes. In particular, the problem of finding a nice representation of graphs in these classes has very rarely been studied; how can one represent them nicely if they are NP-complete to recognize?

As an example, let us look at graphs with boxicity 2. The implicit graph question is not interesting for this class, since the definition gives an implicit representation. However, if a graph is given in adjacency list form, we will not be able to find the rectangular representation efficiently. This does not necessarily mean that we cannot find an alternate implicit representation efficiently, however.

The most obvious attempt is to find an approximation algorithm; perhaps we can find an algorithm such that if G has boxicity 2, we can construct a representation of G as boxes in k dimensional space for some fixed k. Since these problems have not been studied carefully, our level of knowledge is very low. Not only do we not know whether we can find a k dimensional representation of a boxicity 2 graph in polynomial time, we do not even know whether we can find a representation is $n^{\cdot}5$ dimensional boxes in polynomial time; even achieving a bound of $o(n)$ is not known, though the last problem might not be difficult. The analogy here is with graph coloring; when Johnson first examined the problem of finding a coloring for a 3-colorable graph, he found that all known algorithms could use $\Theta(n)$ colors [103] This bound has since been reduced to $O(n^{3/8}\text{polylog}(n))$ [17], and the first improvements used very basic graph theory [179].

However, approximation algorithms are not the only direction of research. If we want an implicit representation, we should not be restricted to a representation which is a variant on boxicity. The situation is similar to the ??? optimization problems studied in chapter ?; we might find that boxicity 2 graphs are a subclass of a graph class for which an implicit representation can be found efficiently, or it is possible that we find one of a number of types of implicit representations depending on a choice made by a representation algorithm. Certainly the boxicity 2 graphs have many interesting properties, such as having $O(n^2)$ maximal cliques, which could give rise to an entirely different form of efficient representation.

I do not know of this problem being studied on any class of graphs. In my opinion, finding an implicit representation in polynomial time would be interesting for many classes which are NP-complete to recognize, including boxicity 2 graphs, interval number 2 graphs, partial orders of dimension 3, disk intersection graphs, EPT graphs, and many more. The problem is also interesting for classes which are open with respect to recognition, such as visibility graphs and circle orders. I feel this should be possible for at least some of the classes described above, and is sufficiently important to be posed separately as an open problem.

Open Problem: Can you construct an implicit representation for graphs in these classes in polynomial time?

15.5 Exercises

1) Show that every split graph is chordal.

2) Show that every chordal graph is the intersection graph of a tree with all degrees no larger than 3.

3) Show that if we are given a perfect elimination scheme for a connected chordal graph, every vertex v except the last has an edge to a vertex which appears after v in the perfect elimination scheme.

4) Show how to test whether the unit interval model represents G correctly in $O(n+m)$ time.

5) Prove that in any level of the breadth first search, the postorder ordering of the unit interval construction on a unit interval graph is in the same order as the ordering found by the earlier breadth first search.

6) Show that if G is a unit interval graph, the algorithm finds a valid unit interval model. You may assume that problem 5 has been solved.

7) The following approach due to Yiannis Papoutsakis also constructs an implicit unit

interval representation efficiently. Start with any proper interval representation. The length of an interval is the sum of the weights of all points between its endpoints. Initially, the points are equal to the endpoints; all endpoints are given weight 1. Consider endpoints in left to right order; we will assume that all intervals to the left of the current interval i have the same length l. If length(i) $<$ length (i-1), add a point with weight length(i-1) - length(i) immediately before the right endpoint of i. If length(i) $>$ length(i-1), add points with weight length(i) - length(i-1) in such a way that every interval to the left of i contains exactly one new point.

a) Show that the algorithm halts with all lengths of size O(n).

b) Show that we can make this construction algorithm run in O(n) time if the proper interval representation is given.

8) It is tempting to believe that the maximum length of intervals in a unit interval model can be made no larger than the maximum number of neighbors of a vertex plus 1. Show that there are unit interval graphs such that the maximum length - maximum number of neighbors is arbitrarily large for any unit interval model of the graph.

16 Applications

Before dealing with applications of classes of graphs, I must confess that my own interests are not based on applications at all. I enjoy seeing applications of the classes, but have not worked directly on any of them, and I feel that I am more likely to make erroneous statements here than anywhere else in the book. I also feel uneasy talking about applications, after seeing too many papers which continue to describe an application of a graph class long after the relevant problem has become obsolete. Finally, I cannot hope to compete in this chapter with the writing or the scope of applications in Roberts' book [138]; Golumbic [79] also covers applications of many graph classes very nicely. Nevertheless, quite a number of these classes have applications in areas which did not exist when the books mentioned above were written, so I will give some coverage of these new applications.

16.1 Computational Biology

One of the most important historical applications in the field is the relationship between interval graphs and arrangement of genes on the chromosome. Early work, due to Benzer [13], was important both in biology and for stimulating mathematical research on a variety of graph classes. Enormous recent advances and interest in the field of computational biology have made the actual problem Benzer worked on less important, but the connection between gene placement and interval graphs is behind a number of important new problems connecting the fields. Since graph theorists and biologists started talking, they have also discovered other applications of graph theory, some of which will be discussed here.

16.1.1 Genome Reconstruction

This is the area which is the most direct descendant of Benzer's work. When Benzer did his study, the only way to get information about the arrangement of genes on the chromosome was to look at traits which were observable on the individual, usually in terms of mutations; this is clearly not going to get very far if we want to map the human genome, which is a popular research topic among current biologists. Since that time, many tests have been developed which allow examination of pieces of the genetic structure itself.

An important technique at the current time is to cut relatively small, random pieces of the chromosome, and run chemical experiments on the small pieces to get accurate information about these. This allows researchers to get information about fragments and the frequency of types of fragments, but it loses information about how the fragments are ordered on the chromosome. Thus, we want to reconstruct the chromosome from the fragments. Since each fragment should form an interval on the chromosome, it is natural to model this problem as an interval graph problem, with fragments corresponding to vertices.

For this model to be useful, there must be a way to determine which fragments should overlap in the chromosome, corresponding to which vertices should be adjacent in the interval model. Such tests have been developed, which makes it seem simple to use interval graph techniques to solve the gene reconstruction problem.

There is an important problem with the tests for whether fragments overlap, however; the tests may produce errors. Errors come in two forms; the tests may say that fragments overlap when they do not (a false positive test), or they may say that fragments do not overlap when they really do (a false negative). Depending on which test is used and what thresholds are set, false positives may be much more common than false negatives, much less common, or both may occur with relatively equal probability.

This gives rise naturally to a number of graph theoretic problems. If we assume that all errors are false negatives, we want to know the minimum number of edges we can add to make the graph an interval graph. This problem is called the interval graph completion problem. If we assume that all errors are false positives, we want to know the minimum number of edges we can delete from G to get an interval graph; this is called the ? problem. If errors can be false negatives or false positives, we want to minimize the number of edge additions and deletions so that the result is an interval graph; this is called the interval graph sandwich problem.

All of the problems above are NP-complete. However, there are additional constraints which may make the problem easier. For example, we know that all of the intervals are of approximately (how close this approximation is depends on the exact test) the same length; thus, it is natural to study the same problems on unit interval graphs as well. (look up Shamir results for this case)

Of course, biologists are different from those of us who design algorithms; they do not really want the optimal solution to the graph theoretic problem if this does not look like the actual chromosome arrangement. Thus, solutions to the interval ?, sandwich, or completion problems should be viewed as candidate arrangements rather than final results, and this can affect our choice of algorithms. There are many attempts to find practical solutions to the special cases of NP-complete problems as well, both through imposing further constraints coming from by biology and by designing good approximation algorithms.

16.1.2 Phylogeny

Another link between the field of graph representations and computational biology comes from the phylogeny problem. Fundamentally, the phylogeny problem is an attempt to find the most probable evolutionary development of a set of animals. For example, it is assumed that chimpanzees, gorillas, orangutans, babboons, and humans all had a common ancestor at some point in the past. At some point, some of the animals evolved in one direction while others evolved in another. How do we determine whether there was a split into, for example, a common ancestor of chimpanzees, gorillas, and humans and a common ancestor of babboons and orangutans,

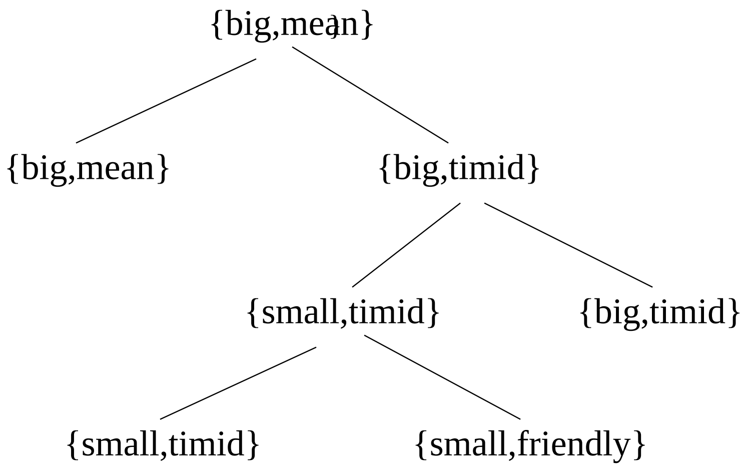


Figure 36: A Possible Perfect Phylogeny Tree

as opposed to a split in which humans evolved on one path and the other apes on another?

The evolutionary sequence can be described by some tree. What would make one evolutionary tree a better candidate for the actual sequence than another? In general, this is not a trivial question to answer, but some rules seem reasonable. Suppose that we have a set of characteristics for each species; each characteristic can have some number of different values. The perfect phylogeny question asks whether there is a single tree with the species as leaves, all internal nodes assigned some value for each characteristic, such that nodes with the same value on a characteristic form a connected subtree. Note that internal nodes correspond to ancestor species.

For example, suppose that we have four species evaluated by size and behavior. Species A is big and mean, B is big and timid, C is small and timid, D is small and friendly. A possible perfect phylogeny is shown in figure ?.

Since this is a model involving subtrees of a tree, it is natural to look for a relationship between perfect phylogeny and chordal graphs. The connection was discovered by Buneman [30] in 1974, and will be described below; he turns the perfect phylogeny problem into a colored chordal graph completion problem, in which one is allowed to add edges between vertices of different colors in order to produce a chordal graph.

LOOK UP CONSTRUCTION: JUST GUESSING THIS THE SAME AS THEIRS

Construct a graph G with a vertex for every value of every characteristic. Add edges between vertices if there is a species which has both of the corresponding characteristic values. Assign a color to each characteristic; thus, vertices with different values for this characteristic will get the same color, which will correspond to not allowing edges between these vertices.

In the example above, there are 5 vertices; big and small get color 1 while mean, timid, and friendly get color 2. There would be edges between big and mean, big and timid, small and timid, small and friendly. Since this graph has a colored chordal completion (actually, it is already chordal), there is a perfect phylogeny.

Suppose that there is a valid solution to the perfect phylogeny problem. It is easy to construct the corresponding chordal graph G', in which two vertices are adjacent iff the corresponding characteristic value trees have a nonempty intersection; it should be clear that G' contains G and includes no edges between vertices of the same color.

Suppose that we can create a graph G' as the solution of the colored chordal graph completion problem. We can easily modify the clique tree for G' to be a solution to the perfect phylogeny problem. The set of characteristics associated with a species correspond to a maximal clique in G'; if this clique is an internal node i rather than a leaf of the clique tree, we can add a child of i with the same characteristic values as i. Thus, chordal colored graph completion is equivalent to the perfect phylogeny problem.

The perfect phylogeny/colored chordal graph completion was open for two decades before being shown to be NP-complete in [18]; various restrictions can be solved in polynomial time. Other phylogeny problems are also studied, and chordal graph models may play a role in this area.

16.2 Networks

There are many connections between intersection graphs and networks, some of which will be discussed here. Since advances make a particular physical model outdated very quickly, I will try to describe applications which are not closely tied to a specific technology.

Fundamentally, networks are arranged in a particular topology, which corresponds naturally to a graph. Messages of various types are passed between nodes in the network. Each message corresponds to a path in the graph. In some models, there is a choice of which path should correspond to a message, while in other models the path is completely specified. Various notions of conflict between messages, such as requiring that messages are node disjoint or share no common links, correspond directly to questions about an underlying intersection graph.

Thus, if your network is a simple ring, messages will correspond to arcs on the circle. If we assume that the routing is specified for each message, we may want to solve the coloring problem on a circular-arc graph in order to minimize (??). Although this problem is NP-complete on circular-arc graphs, good approximation algorithms exist. Other common simple networks correspond to interval graphs, and either path graphs or EPT graphs.

It is interesting to note that coloring is NP-complete on circular-arc graphs, but the clique problem can be solved in polynomial time.

For a long time, no algorithm was known in the western world for finding a coloring of circular-arc graphs which guaranteed a coloring of cOPT, where $c < 2$. This was improved to 5/3 OPT by Hsu and Shih [98]. However, a simple algorithm using

3/2 OPT colors was discovered by Karapetjan in 1980; this algorithm is still largely unknown in the west, since it appeared only in Russian.

This problem has been posed to me more than any other problem by researchers who claimed applications, so I will take the opportunity to describe both algorithms. This may seem to be an odd choice, since the Karapetjan algorithm uses fewer colors, appeared earlier, and is simpler. I choose to present both algorithms because although I have checked the correctness of Karapetjan's algorithm by going through his proof on a line by line basis, I cannot present the proof at a level which I feel is appropriate for this text, while Shih and Hsu's proof follows more directly from the algorithm itself.

We first present Karapetjan's remarkably simple algorithm, completely without proof. Let p be a point of the circle with maximum "load", i.e. there is no point on the circle which has more arcs passing through the point than p. Let L be the load at p. Order the arcs which pass through p as a_1,a_2,...,a_L, with $i < j$ meaning arc j extends further in the clockwise direction from p than arc i.

In phase 1, the algorithm finds $L/2$ independent sets in G - a_1,...a_L as follows. For $i = 1$ to $L/2$, start with the first arc clockwise from a_i, and continue around the circle, always choosing the next clockwise arc from the current vertex.

In phase 2, the algorithm finds L independent sets in G - vertices chosen in phase 1 as follows. For $i = 1$ to L, start with arc a_i, and continue around the circle counterclockwise, always choosing the next arc counterclockwise which is independent of all previous arcs in the family.

This simple algorithm always uses at most 3/2 L colors; the proof can be found in []. Since L is clearly no more than the chromatic number of G, this algorithm uses at most 3/2 OPT colors.

We now present the algorithm of Shih and Hsu, which finds a 5/3 OPT coloring of a circular-arc graph. One of the problems in working with circular-arc graphs is that it is not always true that cliques meet at a point. Shih and Hsu show that it is easy to modify an arbitrary circular-arc graph in polynomial time so that the size of the chromatic number and maximum clique size do not change, and there is a maximum clique C such that all arcs of C go through a common point in the circle. To do this, we find the maximum clique C' in G and the maximum load on the circle, and add a set of tiny arcs on the interval with maximum load to increase the clique size to the size of C'. Exercise ? asks the reader to show that this does not alter the size of the maximum clique or ther chromatic number. For the rest of the discussion, we will assume that C is a maximum clique such that all arcs of C pass through a common point on the circle.

To simplify the description, it is useful to describe a natural greedy notion for coloring a circular-arc graph. If we start at any arc u, we traverse the circle, adding the next arc which is independent of u, until u is encountered again, giving all these arcs a single color. We can extend this single independent set to a longer sequence by starting with the first arc v in our sequence which overlaps with u, and finding our next independent set starting from v. This method of coloring will be called greedy coloring; it is uniquely defined if we are given a starting arc, a finishing arc, and how many times we wish to go around the circle.

A singular pair (singular trio) is a pair (trio) of arcs which cover the circle. Note that any singular pair on a graph with at least three vertices can be extended to be a singular trio. Tucker [] showed that a simple algorithm will use 3/2 $|C|$ colors on a circular-arc graph if there are no singular trios. Hsu and Shih extended this algorithm to find a 3/2 $|C|$ coloring if there is a clique C which contains exactly one vertex from each singular trio. Thus, the initial goal is to eliminate singular trios which do not involve exactly one vertex of C without using too many colors.

We first describe the step used for eliminating bad circular trios. At each removal step, the algorithm removes at least two vertices of C, reduces the load on the circle by 1, and uses at most 2 colors []. If we have made $|C|/3$ removal steps, we stop. In this case we have used at most $2|C|/3$ colors, the load on the circle is at most $2|C|/3$, and the number of arcs passing throught the common point of C is at most $C/3$; this gives us a $5/3|C|$ coloring algorithm by cutting the circle at the common point of C. If we stop after some number t of iterations where $t < |C|/3$, the total number of colors used is at most $2t$ + 3/2 ($|C|$ - t) = $3|C|/2 + t/2 \leq 5|C|/3$. After describing the removal steps, we will describe the coloring algorithm for the case in which no bad singular trios remain.

We begin by successively removing singular pairs from C until none remain. Let the sequence of pairs removed at this point be (u_1,v_1),...,(u_k,v_k). If this removes all vertices of C, it is easy to see that V - C is an interval graph which can be colored with at most $|C|/2$ colors, giving a 3/2 OPT algorithm. Let us assume there are some remaining vertices C' of C after this step. There must be a point d on the circle which is not covered by any arc in C'; let D be the arcs of V - C which cover d.

Construct a bipartite graph with vertices of D on one side, vertices of C on the other, with c adjacent to d if and only if the vertices are nonadjacent in G. Create a maximum matching M on this subgraph; we note that this can be done in O(n^2) time using the techniques presented in chapter ?. Every vertex of D will be matched in M, since C is a maximum clique (exercise ?).

We now eliminate the singular pairs (u_1,u_k)...(u_k,v_k). For each (u_i,v_i), we give one color to u_i and M(u_i) (if it exists), and another color to v_i and M(v_i).

We next eliminate singular pairs which have one vertex in C'; the other vertex of the pair must be in D. Let (c_1,d_1)...(c_j,d_j) be the sequence of singular pairs found at this stage. Once again, we use one color for c_i and M(c_i) and another color for d_i and M(d_i), using two colors and eliminating two vertices of C (this condition is where the matching is useful) and reducing the load by at least 1. At the end of this stage, we may assume that the model has no singular pairs, and that any singular trios contain a vertex which is not in C.

We next eliminate singular trios which contain exactly two vertices of C (and thus exactly one vertex of D). All arcs of C meet at a point p; let c_1 be the arc of C which extends furthest clockwise from p. If there is a singular trio containing two members of C, then there is a circular trio which contains c_1 and some other member of C.

Traverse the circuit in clockwise fashion starting with the counterclockwise endpoint of c_1, maintaining a sequence c_1 through c_i of elements. Add the next independent arc to the set, until some arc c_k passes the endpoint of c_1. Give 1 color to c_1 ... c_{k-1}. If c_k is in C and is different from c_1, we give a separate color to c_k; and if c_k is not in C we give c_k and $\mathrm{M}(c_k)$ a separate color. Exercise ? asks the reader to show that c_k is in D in the latter case, which is necessary for the algorithm to be valid.

We now describe an algorithm for finding a $3s/2$ coloring of a circular-arc graph with load s in which every singular trio contains exactly one vertex from clique Q, where all arcs of Q pass through a common point p. The algorithm works by successively using three colors, and reducing the load by 2.

Start with the leftmost arc u_1 on C, if C is nonempty; otherwise, any arc is chosen as u_1. Find a greedy coloring from u_1 until u_1 is reached a second time; let u_k be the arc found in greedy fashion which overlaps with u_1. If $u_k = u_1$, we are in a good situation (the load has been reduced by 1 at the expense of 1 color); otherwise, assume u_k overlaps with u_1. We find a greedy sequence of arcs starting with u_k which reaches u_k again, and then continue the greedy coloring through the arc which reaches u_1. Color all arcs from u_k until you have reached u_k a second time with one color, and all arcs after u_k is reached a second time with another color. The fact that there are no singular trios involving two arcs of C guarantees that this is a valid coloring []; it reduces the load by two using three colors, and thus leads to a $3s/2$ coloring algorithm for this case.

Permutation graphs, and more generally circle graphs, are often mentioned in the context of network routing problems (look up).

16.3 Programming Languages

There has been a good deal of research dealing with special classes of graphs and the flow graphs of various programming languages. To see a simplified form of this connection, think of solving register allocation problem on a program which has no branching statements whatsoever. A variable remains active over a fixed interval of the program. Two variables must be assigned to different registers if their intervals overlap; thus the register allocation problem on such a program can be viewed as coloring an interval graph.

Some generalizations of the example above are given in Roberts [138]. (DISCUSS VALDES?) A more recent and surprising result due to Thorup relates real programming language flow graphs to the parameter treewidth. Treewidth has been defined in a number of ways [] []; we give the following two characterizations as most in keeping with the ideas in this book.

A triangulation of a graph G is obtained by adding edges to obtain a graph G' which is chordal. The treewidth corresponds to the size of the smallest maximum clique size in any possible triangulation of G - 1. A k-tree is formed by starting with a clique of size k, and repeatedly adding a new vertex adjacent to a k-clique in the existing graph. A partial k-tree is a spanning subgraph of a k-tree. The treewidth of G is also equal to the smallest value k such that G is a partial k-tree.

Specifically, Thorup [] has shown the following results relating treewidth to control-flow graphs. These results imply that one can find an optimal solution to the register allocation problem in all of these types of programs.

Goto-free Algol and Pascal programs have control-flow graphs with treewidth at most 3.

All Modula-2 programs have control-flow graphs with treewidth at most 5.

Goto-free C programs have control-flow graphs with treewidth at most 6.

16.4 A Cautionary Tale

There is an application of interval graphs to the seriation problem in archeology which is well known to theorists in the field; in fact, I have seen this listed as an application of other classes of graphs as well, even though the problem really deals with intervals. In this problem, we are given information about which types of pottery are found in common graves. We assume that a type of pottery is made over a time interval, and we want to find relative dating as to when a pottery type was made.

This maps nicely to an interval graph problem, if we place an edge between vertices which were found in a common grave. An interval model for the corresponding graph maps to a possible ordering in time of the pottery types.

The majority of this book was written while I lived in the unusually pleasant confines of the American School of Classical Studies in Athens, due to my wife's work in the field of archeology. Thus, I was living among one of the densest concentrations of archeologists on the planet. At some point, I was asked what this book was about, and decided to include a description of the archeological application to help introduce graph theory.

Introducing the application proved to be a serious mistake, since the application seems to bear no relationship to ways in which pottery is actually dated. At this point, I looked up the original paper on the subject. The author notes the relationship between interval graphs and dating pottery, but in later papers by the author this is viewed as an unrealistic case, in which all information about time overlap is known. Later papers deal with a much more difficult problem, in which each pair of pottery types is assigned a probability, and one attempts to come up with a model which is a best fit in a probabilistic sense. In fact, all these approaches are theoretical; archeologists do not normally date pottery using any mathematical tools at all.

The moral of this story is to beware of applications passed on in theoretical texts like this one, until you can validate them with people who really work in the field!

16.5 Exercises

1) Consider the intersection graphs of simple paths (no repeated vertices allowed) on a chain of edge pairs as shown below, where intersection is defined to be sharing a common edge.

a) Show that there are $2^{\Theta(n^2)}$ graphs in the class.

b) Show that the class contains the circular-arc graphs, so coloring is NP-complete. Note: you cannot quite interpret the model as a circle by separating the bottom and top of each edge pair, since paths must be simple.

c) Find a 2 times optimal approximate coloring algorithm for this class.

Note: Nomikos [] showed that if you expand the chain of edge pairs to arbitrarily large number of multiedges between a pair of vertices, there is no polynomial approximation algorithm for coloring with constant error factor unless P = NP. If one allows arbitrary number of multiedges, but fix the length of the chain to be constant, the approximation problem seems to be open.

2) Suppose that you are given a circularly ordered set of nodes, and instead of being given paths between vertices (corresponding to circular arcs), you are given endpoints of a proposed message; this allows you to choose an arc which goes in either direction around the circle. Many problems may be proposed in such a scheme; suppose that we want to minimize the maximum "load" of any node; the load of a node is the number of arcs which pass through the node. Design a 2 * optimal algorithm for this problem. The problem can be solved optimally in polynomial time; see [Kleinberg's comments].

3) Design a polynomial time algorithm to determine whether a circular-arc graph is 3-colorable. Note: this can be generalized to any fixed k. It can be solved in O(n) time for any fixed k.

4) Show that the maximum matching in the circular-arc coloring algorithm will match every vertex of D.

4) Show that a circular-arc graph with clique cover number 3 can be colored using at most 5/3 OPT colors in polynomial time. [] gives a 5/3 clique size coloring algorithm for general circular-arc graphs, but that is much more difficult!

5)(This problem solution due to Vijay Raghavan) Show that there are circular-arc graphs with clique size k and chromatic number $3k/2$ for arbitrarily large values of k. Note: making copies of vertices on a 5-cycle will not work here.

6) Show that the maximum clique size in a circular-arc graph can be arbitrarily close to twice the load on the circle.

7) Show adding tiny arcs on load does not change maximum clique size or chromatic number.

8) Show that the load is equal to the maximum number of arcs which overlap some clockwise endpoint.

9) Show that if last arc found in greedy independent set is not in C, it is in D.

10) Suppose that we are given a fixed number of colors, and we want to color the maximum possible number of vertices. Show that greedy coloring on a set of intervals (i.e. color the earliest ending vertex with a legal color, if possible) does not necessarily color the maximum number of vertices. I do not know of an $\mathrm{O}(n)$ solution to this problem.

References

[1] J. Abello, H. Lin, S. Pisupati, On Visibility Graphs of Simple Polygons, Congressus Numerantium 90 (1992), 119-128

[2] P.K. Agarwal, N. Alon, B. Aronov, S. Suri Can Visibility Graphs be Represented Compactly?, Discrete Computational Geometry 12, 347-365, 1994

[3] P. Agarwal, M. Sharis, S. Toledo, Applications of parametric searching in geometric optimization, Journal of Algorithms 17, 292-318, 1994

[4] A. Agarwal, S. Suri, The biggest diagonal in a simple polygon, Information Processing Letters 35, 13-18, 1990

[5] R.P. Anstee, M. Farber, Characterizations of Totally Balanced Matrices, *J. Algorithms* 5 (1984) 215–230

[6] R.P. Anstee, Z. Füredi, Forbidden Submatrices, Discrete Mathematics 62 (1986), 225-243

[7] A. Apostolico, M.J. Atallah, S. E. Hambrusch, New clique and independent set algorithms for circle graphs, *Discrete Appl. Math.* 36 (1992) 1–24, Erratum: *Discrete Appl. Math.* 41 (1993) 179–180

[8] J.C. Arditti, Partially ordered sets and their comparability graphs, their dimension and their adjacency, *Problemes Combinatoires et Theorie des Graphes*, 1976, *Proc. Coll. Int. CNRS*, Orsay, France

[9] P.K. Agarwal, M. Sharir, P. Shor Sharp Upper and Lower Bounds for the Length of General Davenport-Schinzel Sequences J. Combinatorial Theory Series A 52, 1989 228-274

[10] A. V. Aho, J. E. Hopcroft, J. D. Ullman The Design and Analysis of Computer Algorithms Addison-Wesley Reading, MA 1974

[11] N. Alon, E.R. Scheinerman Degrees of Freedom Versus Dimension for Containment Orders Order 5, 1988 11-16

[12] H. Bandelt, H.M. Mulder Distance-Hereditary Graphs, Journal of Combinatorial Theory B 41, 182-208, 1986

[13] S. Benzer On the Topology of the Genetic Fine Structure, Proc. Nat. Acad. Sci. USA 45, 1607-1620, 1959

[14] V.L. Beresnev, A.I. Davydov On Matrices with Connectedness Properties, Upravlyaemye Sistemy 19, 3-13, 1979

[15] C. Berge Farbung von Graphen, deren samtliche bzw. deren ungerade Kreise starr sind, Wiss. Zeitschr. Martin-Luther-Univ. Halle-Wittenburg, 1961, 114

[16] C. Berge, Hypergraphs, North Holland, Amsterdam, 1989

[17] A. Blum, New Approximation Algorithms for Graph Coloring, JACM 41, 470-516, 1994

[18] H.L. Bodlaender, M.R. Fellows, T. Warnow, Two Strikes Against Perfect Phylogeny, ICALP 92, LNCS 623 273-283, 1992

[19] K.S. Booth, G.S. Leuker Testing for the Consecutive Ones Property, Interval graphs, and Planarity Using PQ-Tree Algorithms Journal of Computer and System Science 13, 335-379, 1976

[20] R. Borie, R.G. Parker, C.A. Tovey, A Deterministic Decomposition of Recursive Graph Families, SIAM J. Disc. Math 4, 481-501, 1991

[21] A. Bouchet, Circle Graph Obstructions, JCTB 60, 107-144, 1994

[22] A. Bouchet Reducing Prime Graphs and Recognizing Circle Graphs, Combinatorica 7, 243-254, 1987

[23] A. Brandstädt, V.B. Le, J. Spinrad Graph Classes - A Survey, manuscript (1997)

[24] A. Brandstädt, V.D. Chepoi, F.F. Dragan, The Algorithmic Use of Hypertree Structure and Maximum Neighborhood Orderings, Discrete Applied Math, to appear

[25] G. Brightwell, On the Complexity of Diagram Testing, Order 10, 297-303, 1993

[26] Forbidden Induced Partial Orders, G. Brightwell, D.A. Grable, H.J. Promel, Forbidden Induced Partial Orders, manuscript 1998

[27] H. Broersma, T. Kloks, D. Kratsch, H. Müller, Independent Sets in Asteroidal Triple-Free Graphs, manuscript, 1997

[28] H. Buer, R.H. Mohring A Fast Algorithm for the Decomposition of Graphs and Posets, J Math. Oper. Res. 8, 170-184, 1983

[29] K.B. Cameron Polyhedral and Algorithmic Ramifications of Antichains, PhD Thesis, University of Waterloo, 1982

[30] P. Buneman, A Characterization of Rigid Circuit Graphs, Discrete Math. 9, 205-212, 1974

[31] B. Chazelle A polygon cutting theorem, 23d FOCS 339-349, 1982

[32] V. Chvatal, Perfectly Ordered Graphs, Annals of Discrete Math 21, 63-65, 1984

[33] O. Cogis, A Characterization of Digraphs with Ferrers Dimension 2, Res. Report 19 CNRS Paris 1979

[34] O. Cogis, On the Ferrers Dimension of a Digraph, Discrete Math 38, 47-52, 1982

[35] C. Coullard, A. Lubiw, Distance Visibility Graphs, International Journal of Computational geometry Applications 2, 349-362, 1992

[36] D. Coppersmith, S. Winograd Matrix Multiplication via Arithmetic Progressions, Journal of Symbolic Computation 9, 1-6, 1990

[37] T. Cormen, C. Leiserson, R. Rivest Introduction to Algorithms, MIT Press, Cambridge MA, 1989

[38] D. G. Corneil, P. K. Kamula, Extensions of Permutation and Interval Graphs, Congressus Numerantium 58, 267-276, 1987

[39] D.G. Corneil, H. Kim, S. Natarajan, S. Olariu, A.P. Sprague, Simple Linear Time Recognition of Unit Interval Graphs, Information Processing Letters 55, 99-104, 1995

[40] , D. Corneil, S. Olariu, L. Stewart, A Linear Time Algorithm to Compute a Dominating Path in AT-free Graphs, Information Processing Letters 54, 253-257, 1995

[41] D. Corneil, S. Olariu, L. Stewart, Asteroidal Triple-Free Graphs SIAM Journal on Discrete Math 10, 399-430, 1997

[42] D. Corneil, S. Olariu, L. Stewart, The Ultimate Interval Graph Recognition Algorithm?, SODA, 1998

[43] D.G. Corneil, Y. Perl, L.K. Stewart A Linear Recognition Algorithm for Cographs, SIAM Jornal on Computing 14, 926-934, 1985

[44] A. Cournier, M. Habib, A New Linear Algorithm for Modular Decomposition, Lecture Notes in Computer Science 787, 68-84, 1994

[45] W.H. Cunningham, J. Edmonds, A Combinatorial Decomposition Theory, Canadian J. of Mathematics 22, 734-765, 1980

[46] W.H. Cunningham Decomposition of Directed Graphs, SIAM J. Alg. Disc. Meth. 3, 214-228, 1982

[47] E. Dahlhaus, Efficient Parallel and Linear Time Sequential Split Decomposition, manuscript, 1997

[48] E. Dahlhaus, J. Gustedt, R.M. McConnell, Efficient and Practical Modular Decomposition, 8th Annual SODA, 26-35, 1997

[49] X. Deng, P. Hell, J. Huang, Linear-Time Representation Algorithms for Proper Circular-Arc Graphs and Proper Interval Graphs, SIAM Journal on Computing 25, 390-403, 1996

[50] P.F. Dietz, Intersection Graph Algorithms, Ph.D. Thesis, Comp. Sci. Dept., Cornell University, 1984

[51] G.A. Dirac, On Rigid Circuit Graphs, Abh. Math. Sem. Univ. Hamburg 25, 71-76, 1961.

[52] D. Dobkin, H. Edelsbrunner Searching for Empty Convex Polygons, Algorithmica, vol. 5, pp. 561-571, 1990.

[53] B. Dushnick, E. W. Miller Partially Ordered Sets, Amer. J. Math. 63, 1941, 600-610

[54] P. Eades, R. Tamassia, I.G. Tollis, Graph Drawing, Prentice Hall, Upper Saddle River, New Jersey, 1998

[55] E.S. Elmallah, L.K. Stewart. Independence and domination in polygon graphs, Discrete Applied Math 44, 65-77, 1993

[56] , E.S. Elmallah, L.K. Stewart, Polygon Graph Recognition, Journal of Algorithms, to appear

[57] P. van Emde Boas, Preserving Order in a Forest in Less than Logarithmic Time, 16th FOCS 75-84, 1975

[58] E. Eschen, R. Hayward, J. Spinrad, R. Sritharan, Weakly Triangulated Comparability Graphs, submitted for publication

[59] E. Eschen, J. Spinrad An O(n^2) Algorithm for Circular-Arc Graph Recognition Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms 128-137, 1993

[60] E. Eschen, Circular-Arc Graph Recognition and Related Problems, Ph.D. Thesis, Dept. of Computer Science, vanderbilt University (1996)

[61] S. Even, Graph Algorithms, Computer Science Press, 1979

[62] H. Everett, D. Corneil Negative Results on Visibility Graphs, Computational Geometry Theory and Applications 5, 51-63 (1995)

[63] M. Farber, Applications of Linear Programming Duality to Problems Involving Independence and Domination, Ph.D. Thesis, Simon Fraser University, 1981

[64] C.M. Fiduccia, E.R. Scheinerman, A. Trenk, J.S. Zito, Dot product representations of graphs, manuscript 1996

[65] P. C. Fishburn Interval orders and interval graphs Wiley New York 1985

[66] S. Felsner, R. Muller, L. Wernisch, Trapezoid graphs and generalizations, geometry and algorithms, LNCS 824 SWAT '94, 143-154, 1994

[67] L. Ford, D. Fulkerson Flows in Networks 1962 Princeton University Press Princeton NJ

[68] D.R. Fulkerson, O.A. Gross, Incidence Matrices and Interval Graphs, Pacific Journal of Mathematics 15, 835-855, 1965

[69] C.P. Gabor, W.L. Hsu, K.J. Supowit Recognizing Circle Graphs in Polynomial Time, Journal of the ACM 36, 435-473, 1989

[70] T. Gallai, Transitiv Orientbare Graphen, Acta Math. Acad. Sci. Hung. Tom. 18, 25-66, 1967

[71] M.R. Garey, D.S. Johnson Computer and Intractability: A Guide to the Theory of NP-Completeness, Freeman, San Francisco, CA, 1978

[72] A. Garg, R. Tamassia, Upward Planarity Testing, Order 12, 109-133, 1995

[73] F. Gavril, Algorithms on Circular-Arc Graphs, Networks 4, 357-369, 1974

[74] F. Gavril A Recognition Algorithm for the Intersection Graphs of Directed Paths in Directed Trees, Discrete Math. 13, 237-249, 1975

[75] F. Gavril, The Intersection Graphs of Subtrees of Trees are Exactly the Chordal Graphs, J. of Combinatorial Theory Series B 16, 47-56, 1974

[76] S. Ghosh, On recognizing and characterizing visibility graphs of simple polygons, *Lect. Notes Comp. Sci.* 318, Springer-Verlag, 96-104 (1988)

[77] P.C. Gilmore, A.J. Hoffman A Characterization of Comparability Graphs and Interval Graphs, Canadian J. of Mathematics 16, 539-548, 1964

[78] M.C. Golumbic, P.L. Hammer Stability in Circular-Arc Graphs, Journal of Algorithms 9, 314-320, 1988

[79] M.C. Golumbic, Algorithmic Graph Theory and Perfect Graphs Academic Press New York 1980

[80] M.C. Golumbic, R. E. Jamison The Edge Intersection Graphs of Paths in Trees, J. Combinatorial Theory Ser. B 38, 8-22, 1985

[81] M.C. Golumbic, C.F. Goss, Perfect Elimination and Chordal Bipartite Graphs, Journal of Graph Theory 2, 155-163, 1978

[82] R.L. Graham, B.L. Rothschild, J.H. Spencer Ramsey Theorey, Wiley, New York, 1990

[83] J.R. Griggs, Extremal Values of the Interval Number of a Graph, II, Discrete Mathematics 28, 37-47, 1989

[84] M. Grötschel, L. Lovász, A. Schrijver The Ellipsoid Method and its Consequences in Combinatorial Optimization, Combinatorica 1, 169-197, 1981

[85] M. Habib, R. Jegou N-free Posets as Generalizations of Series-Parallel Posets, Discrete Applied Mathematics 3, 279-291, 1985

[86] M. Habib, D. Kelly, R.H. Möhring, Interval Dimension is a Comparability Invariant, Discrete Math 88, 211-229, 1991

[87] M. Habib, M.C. Maurer On the X-join Decomposition for Undirected Graphs, Discret Applied Math 3, 198-207 (1979)

[88] H. Hadwiger, H. Debrunner, V. Klee, Combinatorial Geometry in the Plane, Holt Rinehardt and Winston, New York, 1964, p. 54

[89] P.L. Hammer, N.V.R. Mahadev, D. de Werra, Stability in CAN-free Graphs, J. Comb. Theory B 38, 23-30, 1985

[90] P.L. Hammer, N.V.R. Mahadev, D. de Werra, The Struction of a Graph: Application to CN-free Graphs, Combinatorica 5, 141-147, 1985

[91] P.L. Hammer, F. Maffray Completely Separable Graphs, Discrete Applied Mathematics 27, 85-100, 1990

[92] K. Han, C.N. Sekharan, R. Sridhar Unified All-Pairs Shortest Path Algorithms in the Chordal Hierarchy, Discrete Applied Math 77, 59-71, 1997

[93] D. Harel, R.E. Tarjan, Fast Algorithms for Finding Nearest Common Ancestors, SIAM Journal on Computing 13, 338-355 (1984)

[94] R. Hayward, Weakly Triangulated Graphs, J. Comb. Theory B 39, 200-209, 1985

[95] R. Hayward, C. Hoang, F. Maffray, Optimizing Weakly Triangulated Graphs, Graphs and Combinatorics 5, 339-349, 1989

[96] A.J. Hoffman, .W.J. Kolen, M. Sakarovitch, Totally-Balanced and Greedy Matrices, SIAM Journal on Algebraic Discrete Methods 6, 721-730, 1985

[97] W.L. Hsu, T.-H. Ma, Fast and Simple Algorithms for Recognizing Chordal Comparability Graphs and Interval Graphs, manuscript

[98] W.L. Hsu, W.K. Shih, An Approximation Algorithm for Circular-Arc Graph Coloring, manuscript, CANT FIND RIGHT NOW; NOT REAL TITLE

[99] W.-L. Hsu, K.-H. Tsai Linear Time Algorithms on Circular-Arc Graphs, Information Processing Letters 40 (1991), 123-129

[100] W.-L. Hsu Decomposition of Perfect Graphs, Journal of Combinatorial Theory, Series B 43, 70-94, 1987

[101] W.-L. Hsu, J. Spinrad, Independent Sets in Circular-Arc Graphs, Journal of Algorithms 19, 145-160 (1995)

[102] T. Ibaraki, U. N. Peled, Sufficient Conditions for Graphs to have Threshold Dimension 2, Studies on Graphs and Discrete Programming, Annals of Disc. Math. 11, 241-268, 1981

[103] D.S. Johnson, Worst Case Behavior of Graph Coloring Algorithms, Proc. 5thSoutheastern Conference on Combinatorics, Graph Theory and Computing, 513-528, 1974

[104] S. Kannan, M. Naor, S. Rudich Implicit Representation of Graphs, SIAM Journal on Discrete Mathematics 5, 1992 596-603

[105] B. Klinz, R. Rudolf, G. Woeginger Permuting Matrices to Avoid Forbidden Submatrices Discrete Applied Mathematics 60, 223-248, 1995

[106] T. Kloks, D. Kratsch, J. Spinrad, On Treewidth and Minimum Fill-in of Asteroidal Triple-free Graphs, Theoretical Computer Science 175, 309-335, 1997

[107] M. Koebe, On a New Class of Intersection Graphs, Fourth Czechoslovakian Symposium on Combinatorics, Graphs, and Complexity, Elsevier, 141-143, 1992

[108] N. Korte, R.H. Mohring Transitive Orientation of Graphs with Side Constraints, Lecture Notes on Computer Science 246, 1-16, 1987

[109] J. Kratochvíl, A Special Planar Satisfiability Problem and a Consequence of its NP-completeness, Discrete Applied Math 52, 233-252, 1994

[110] J. Kratochvíl, Intersection Graphs of Noncrossing Arc-Connected Sets in the Plane, LNCS 1190, 257-270, 1996

[111] J. Lehel, A characterization of totally balanced hypergraphs, *Discrete Math.* 57 (1985) 59–65

[112] C.G. Lekkerkerker, J. Boland, Representation of a Finite Graph by a Set of Intervals on the Line, Fund. Math. 51, 45-64, 1962

[113] H. Lerchs, On Cliques and Kernels, Technical Report, Dept. of Compueter Science, University of Toronto, 1971

[114] Y-L Lin, S. Skiena Complexity Aspects of Visibility Graphs International Journal of Computational Geometry 5 (1995), 289-312

[115] L. Lovasz Normal Hypergraphs and the Perfect Graph Conjecture, Discrete Mathematics 2, 253-267, 1972

[116] A. Lubiw Doubly Lexical Orderings of Matrices, SIAM Journal on Computing 16, 854-879, 1987

[117] T.-H. Ma, J. Spinrad Cycle-free Partial Orders and Chordal Comparability Graphs, Order 8, 49-61, 1991

[118] T.-H. Ma, J. Spinrad On the 2-Chain Sungraph Cover and Related Problems, Journal of Algorithms 1994, 251-268

[119] T.-H. Ma, J. Spinrad Transitive Closure for restricted Classes of Partital Orders, Order 8, 175-183, 1991

[120] T.-H. Ma, J. Spinrad An O($nsup$2) Algorithm for the Undirected Split Decomposition, Journal of Algorithms 16, 1994, 264-282

[121] N.V.R. Mahadev, U.N. Peled Threshold Graphs and Related Topics, Annals of Discrete Mathematics 56, North Holland, 1995

[122] S. Masuda, K. Nakajima An Optimal Algorithm for Finding a Maximum Independent Set of a Circular-Arc Graph, SIAM Journal on Computing 17, 41-52, 1988

[123] R.M. McConnell, J. Spinrad, Linear-time modular decomposition and efficient transitive orientation of comparability graphs, 5th SODA (1997) 536-545

[124] R.M. McConnell, J. Spinrad, Linear-time transitive orientation, 8th SODA (1997) 19-25

[125] M. Middendorf, F. Pfeiffer On the Complexity of Recognizing Perfectly Orderable Graphs, Discrete Mathematics 80, 1990, 327-333

[126] J. Mitas, Minimal Representation of Semiorders with Intervals of Same Length, LNCS 831, 162-175, 1994

[127] R.H. Mohring, Almost all comparability graphs are UPO, Discrete Math 50, 63-70 (1984)

[128] C.L. Monma, V.K. Wei Intersection Graphs of Paths in a Tree, Journal of Combinatorial Theory Series B 41, 141-181, 1986

[129] J.H. Muller Local Structure in Graphs Ph.D. Thesis, School of Information and Computer Science, Georgia Inst. of Technology Atlanta, GA 1988

[130] J.H. Muller, J. Spinrad Incremental Modular Decomposition, Journal of the ACM 36, 1-19, 1989

[131] T. Ohtsuki, A Fast Algorithm for Finding an Optimal Ordering for Vertex Elimination on a Graph, SIAM Journal on Computing 5, 133-145, 1976

[132] J. O'Rourke Art gallery theorems and algorithms, Oxford University Press, 1987

[133] R. Paige, R.E. Tarjan Three Partition Refinement Algorithms, SIAM Journal on Computing 16, 973-989, 1987

[134] C.H. Papadimitriou, M. Yannakakis, Scheduling Interval-Ordered Tasks, SIAM Journal on Computing 8, 405-409, 1979

[135] H.J. Promel, A. Steger, Excluding Induced Subgraphs III. A General Asymptotic, Random Structures and Algorithms 3, 19-31, 1992

[136] R.L. Rivest, J. Vuillemin, On Recognizing Graph Properties from Adjacency Matrices, Theoretical Computer Science 3, 371-384 (1976/77)

[137] F.S. Roberts, Indifference Graphs, Proof Techniques in Graph Theory, Ed. F. Harary, Academic Press, New York, 1969, 139-146

[138] F.S. Roberts, Graph Theory and Its Application to Problems of Society, SIAM, 1978

[139] D.J. Rose, Triangulated Graphs and the Elimination Process, J. Math. Anal. Appl. 32, 597-609, 1970

[140] D.J. Rose, R.E. Tarjan, G.S. Leuker Algorithmic Aspects of Vertex Elimination on Graphs, SIAM J. Computing 5, 266-283, 1976

[141] D. Rotem, J. Urrutia Circular Permutation Graphs Networks 12, 428-437, 1982

[142] A.A. Schäffer, A Faster Algorithm to Recognize Undirected Path Graphs, Discrete Applied Mathematics 43, 261-295, 1993

[143] E. Scheinerman, Local Representation Using Very Short Labels, submitted for publication

[144] E.R. Scheinerman, Characterizing Intersection Classes of Graphs, Discrete Mathematics 55, 185-193, 1985

[145] E. Scheinerman, J. Zito On the Size of Hereditary Classes of Graphs, Journal of Combinatorial Theory Series B 61, 16-39 (1994)

[146] W. Schnyder Planar Graphs and Poset Dimension, Order 5 (1989), 323-343

[147] R. Sankaranarayana L. Stewart Complexity Results for well Covered Graphs, Networks 22, 247-262, 1992 Journal of Combinatorial Theory Series B 61, 16-39, 1994

[148] T. Shermer, Recent results in art galleries, Proceedings of the IEEE 80, 1992

[149] J. Spinrad Nonredundant Ones and Chordal Bipartite Graphs SIAM Journal on Discrete Mathematics 8, 251-257, 1995

[150] J. Spinrad, R. Sritharan Algorithms for Weakly Triangulated Graphs, Discrete Applied Mathematics 19, 181-191, 1995

[151] J.P. Spinrad, On Comparability and Permutation Graphs, SIAM Journal on Computing 14, 658-670, 1987

[152] J.P. Spinrad, Two dimensional partial orders, Ph.D. Thesis, Dept. of EECS, Princeton Univ., 1982

[153] J.P. Spinrad, Doubly Lexical Ordering of Dense 0-1 Matrices, Information Processing Letters 45, 229-235 (1993)

[154] J.P. Spinrad, A. Brandstadt, L. Stewart, Bipartite Permutation Graphs, Discrete Applied Math 18, 279-292, 1987

[155] J.P. Spinrad, Circular-Arc Graphs with Clique Cover Number Two, Journal of Comb. Theory B 44, 300-306, 1988

[156] J.P. Spinrad, Prime Testing for the Split Decomposition of an Undirected Graph, SIAM Journal on Discrete Math 2, 590-599, 1989

[157] J.P. Spinrad, Finding Large Holes, Information Processing Letters 39, 227-229, 1991

[158] J. Spinrad, Recognition of Circle Graphs, Journal of Algorithms 16, 264-282, 1994

[159] J. Spinrad, Dimension and Algorithms, LNCS 831, 33-52, 1994

[160] A.P. Sprague, Recognition of Bipartite Permutation Graphs, Congressus Numerantium 112, 151-161, 1995

[161] R. Sritharan, A Linear Time Algorithm to Recognize Circular Permutation Graphs, Networks 27, 171-174, 1996

[162] R.E. Tarjan Efficiency of a Good but not Linear Set Union Algorithm, Journal of the ACM 22, 215-225, 1975

[163] R.E. Tarjan Decomposition by Clique Separators, Discrete Mathematics 55, 221-232, 1985

[164] R.E. Tarjan, Complexity of Combinatorial Algorithms, SIAM Review 20, 457-489, 1978

[165] W.T. Trotter, Problems and Conjectures in the Combinatorial Theory of Ordered Sets, Annals of Discrete Mathematics 41, 401-416, 1989

[166] W. T. Trotter Jr., J. I. Moore, Characterization Problems for Graphs, Partially Ordered Sets, Lattices, and Families of Sets, Discrete Math. 16, 361-381, 1976

[167] W.T. Trotter Combinatorics and Partially Ordered Sets: Dimension Theory Johns Hopkins University Press Baltimore MD 1992

[168] A.C. Tucker An Efficient Test For Circular-Arc Graphs, SIAM J. of Computing 9, 1-24, 1980

[169] A.C. Tucker, Matrix Characterizations of Circular-Arc Graphs, Pacific J. of Mathematics 19, 535-545, 1971

[170] A.C. Tucker Structure Theorems for Some Circular-Arc Graphs, Discrete Math 7, 167-195, 1974

[171] A.C. Tucker Coloring a Family of Circular-Arc Graphs, SIAM J. of Appl. Math. 29, 493-502, 1975

[172] J. Valdes, R.E. Tarjan, E.L. Lawler The Recognition of Series-Parallel Digraphs, SIAM Journal on Computing 11, 298-314, 1978

[173] J.R. Walter, Representations of Chordal Graphs as Subtrees of a Tree, Journal of Graph Theory 2, 265-267, 1978

[174] H. Warren Lower Bounds for Approximation by Nonlinear Manifolds, Transactions of the AMS 133, 1968, 167-178

[175] D.B. West, D.B. Shmoys, Recognizing graphs with fixed interval number is NP-complete, *Discrete Appl. Math.* 8 (1984) 295–305

[176] D.B. West Introduction to Graph Theory, Prentice Hall, 1996

[177] S.H. Whitesides, An algorithm for finding clique cut–sets, *Inf. Proc. Letters* 12 (1981) 31–32

[178] S.H. Whitesides, A method for solving certain graph recognition and optimization problems, with applications to perfect graphs, *Annals of Discrete Math.* 21 (1984) 281–297

[179] A. Wigderson, Improving the Performance Gurantee for Approximate Graph Coloring, JACM 30, 729-735, 1983

[180] T.V. Wimer, Linear algorithms on k–terminal graphs, Ph. D. Thesis *Dept. of Computer Science, URI-030, Clemson Univ., Clemson, SC* (1987)

[181] T.-H. Wu, An $O(n^3)$ Isomorphism Test for Circular-Arc Graphs, Ph.D. Thesis, Applied Mathematics and Statistics, SUNY-Stonybrook, 1983

[182] M. Yannakakis, Computing the minimum fill–in is NP–complete, *SIAM J. Alg. Discr. Meth.* 2 (1981) 77–79

[183] M. Yannakakis, The complexity of the partial order dimension problem, *SIAM J. Alg. Discr. Meth.* 3 (1982) 351–358

Index