Lecture Notes in
Computer Science 1852

Thomas Thierauf

The Computational Complexity of Equivalence and Isomorphism Problems

Springer

Lecture Notes in Computer Science 1852
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

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

Thomas Thierauf

The Computational Complexity of Equivalence and Isomorphism Problems

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Author

Thomas Thierauf
Universität Ulm, Fakultät für Informatik
Abteilung Theoretische Informatik
Oberer Eselsberg, 89069 Ulm, Germany
E-mail: thierauf@informatik.uni-ulm.de

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Thierauf, Thomas:
The computational complexity of equivalence and isomorphism problems /
Thomas Thierauf. - Berlin ; Heidelberg ; New York ; Barcelona ; Hong
Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2000
(Lecture notes in computer science ; 1852)
ISBN 3-540-41032-5

CR Subject Classification (1998): F.1, F

ISSN 0302-9743
ISBN 3-540-41032-5 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Steingräber Satztechnik GmbH, Heidelberg
Printed on acid-free paper SPIN: 10722133 06/3142 5 4 3 2 1 0

Preface

A *computational model* is a framework to do computations according to certain specified rules on some input data. One of the earliest attempts to capture the intuitive notion of an "algorithm" was the *Turing machine*, which is now used as a synonym for it. Nowadays, there are many computational models that are interesting from a theoretical *and* a practical point of view. These models come for example from automata theory, formal language theory, logic, or circuit theory. All models can be considered with additional resource bounds (on the time, for example) or with syntactic restrictions.

In order to understand the computational power of a model, it is very useful to study in particular the following problems with respect to that model:

- the *satisfiability problem*: given an algorithm of the model, does there exist an input that is accepted by the algorithm?
- the *equivalence problem*: given two algorithms of the model, do they compute the same function?
- the *"almost" equivalence problem*: given two algorithms of the model, is there an "easy" transformation of the algorithms such that they compute the same function?

The theory of computation is the study of the inherent difficulty of computational problems, their *computational complexity*. In this monograph, we study the computational complexity of the satisfiability, equivalence, and "almost" equivalence problems of various computational models. In particular we consider Boolean formulas, circuits, and various kinds of branching programs.

Acknowledgments

This monograph is based on my habilitation thesis (Habilitationsschrift) at the University of Ulm. I have to thank many people who contributed in various ways to this work. First of all I would like to thank *Uwe Schöning*. His way of exploring a problem has certainly influenced my own research, and his very intuitive way of explaining even very complex topics is reflected in this monograph (I hope).

Many thanks go to *Manindra Agrawal* and *Harry Buhrman*, my coauthors on many papers. It is a great pleasure to work with them.

The *Deutsche Forschungsgemeinschaft* supported me with a postdoctoral fellowship. I spent one very productive and exciting year in Rochester with *Lane Hemaspaandra*.

Farid Ablayev introduced me to probabilistic branching programs and *Toni Lozano* to isomorphism problems. *Bernd Borchert* supplied me with the early papers and gave me very useful hints on this topic.

Ingo Wegener made many helpful comments and gave me pointers to the literature. I benefited from discussions with *Sven Baumer, Jin-yi Cai, Lance Fortnow, Jochen Messner, Jacobo Torán*, and *Klaus Wagner*.

The greatest support however clearly came from *Susanne, Katrina, and Jakob*.

April 2000 THOMAS THIERAUF

Table of Contents

Chapter 1

Introduction

A fundamental problem in mathematics and computer science is the following:

determine whether a given boolean formula is a tautology,

that is, whether the formula evaluates to true for all assignments to its variables. It is an open problem already asked by Gödel (in a letter to von Neumann, see [Sip92]), whether there are *efficient* methods to solve this problem.

An obvious algorithm for the *tautology problem* is an *exhaustive search* through all assignments for the formula. But this is impractical: formulas of practical interest usually have so many variables that the fastest existing computers would need decades to explore the whole search space. In contrast, we call an algorithm *efficient* if it has moderate running time.

In a seminal paper, Cook [Coo71] characterized the tautology problem precisely as the set of problems that can be solved by efficient *nondeterministic* algorithms.

Algorithms in the classical sense are deterministic: each step is followed by a uniquely determined next step. *Nondeterministic* algorithms can choose one of several possibilities as the next step. With respect to the tautology problem, nondeterminism can be used to branch to different assignments to the variables of the input formula, so that every possible assignment is reached on some branch. The acceptance criterion now clearly is that *all* possible computations have to lead to a *satisfying assignment* of the input formula.

The characterization given by Cook allows to restate the question whether there are efficient algorithms for the tautology problem as the following question:

can every efficient nondeterministic algorithm be turned into an efficient (deterministic) algorithm?

This is an informal formulation of the famous **P** versus **NP** question. It already had been asked implicitly prior to the work of Cook. (like in the above mentioned letter from Gödel). Cook gave the first *precise* formulation of it.

In the following years, thousands of problems of practical interest were shown to be **NP**-*complete*. That is, they all have about the same computational difficulty in the sense that if one of them can be solved efficiently by a (deterministic) algorithm then this carries over to all of them. The most prominent **NP**-complete problem asks for the *satisfiability of boolean formulas*. That is whether there exists an assignment for a given boolean formula where the formula evaluates to true. This is, in some sense, complementary to the tautology problem. More precisely, a formula is *no* tautology if and only if its negation is satisfiable.

Some of the **NP**-complete problems are very old and have attracted researchers for centuries. The *traveling salesman problem*, where a salesman wants to find a shortest round trip visiting all cities on a given map is another example. Since there has been no success in devising an efficient algorithm for any of them, most people in computer science conjecture that *there is no such algorithm*, or equivalently, that

P *is different from* **NP**.

However, up to now, **there is no proof of this conjecture!** To solve the **P**-**NP** problem is one of the most challenging tasks in computer science.

In the following we consider problems that are related to the tautology problem. Some of them are easier than the tautology problem: they can be solved by efficient algorithms, sometimes with the help of randomization. Others are computationally equivalent or might even be harder to solve. That is, we use the tautology problem or other well known problems as reference points for a new problem. That way, the computational complexity of a new problem is determined *relative* to other, well known problems. We conjecture that such an ordering of problems precisely reflects their computational difficulty. However, until there are major break through results (like a proof that **P** is different from **NP**), this will stay a conjecture. In any case, it is the best we can do right now.

The problems considered here are very interesting and have attracted many researchers. An additional nice aspect is that when we prove results about them, we will encounter a great variety of techniques and notions. Hence this monograph can also be seen as a textbook that uses the various equivalence and isomorphism problems as a vehicle to discover a part of theoretical computer science.

1.1 Equivalence Problems

Boolean formulas represent a syntactic way to describe boolean functions. The mapping is *not* one-to-one: each boolean function can be represented by many (in fact, infinitely many) boolean formulas. Any two boolean formulas that represent the same boolean function are called *equivalent*. Another formulation of the tautology problem is therefore whether a given boolean formula is equivalent to the constant 1 formula.

A seemingly more general problem is to decide the equivalence of two given boolean formulas, the *boolean formula equivalence problem*. However, if F and G are two formulas, then they are equivalent if and only if the formula $F \leftrightarrow G$ is a tautology. Hence the boolean formula equivalence problem is in fact still the tautology problem.

It is very interesting to ask the analog question for computational models other than boolean formulas.

- If the computational model is too general, then the corresponding equivalence problem is *undecidable*, that is, there is no algorithm at all that solves it. This holds for example for Turing machines, LOOP-programs, or context-free grammars (see [HU79]).
- The equivalence problem for regular expressions with squaring was historically the first *natural* problem that, though decidable, was proven to be *not efficiently* decidable [MS72]: it requires exponential time (in fact, exponential space). Here *natural* means that it is not obtained via diagonalization against some computational model. The equivalence of standard regular expressions (without squaring) can be decided in polynomial space.
- For circuits, branching programs and LOOP(1)-programs (no nested loops, see [Sch95]), and star-free regular expressions, the problem is again computationally equivalent to the tautology problem.
- Efficient algorithms are known only for weaker computational models like, for example, finite automata or various kinds of restricted branching programs.

Many problems of practical interest are connected to the respective equivalence problems. For example

- the *hardware minimization problem* was already considered by Shannon [Sha49]: for a given circuit, find the smallest circuit that is equivalent to the input circuit.
- A method currently used in *circuit design verification* is to transform the equivalence problem for the circuits to be verified into an equivalence problem for *ordered branching programs* which can be solved efficiently.
- Ordered branching programs can also be used for *error correction* in logical design as long as the error is very local (single gate error).
- *Sensitivity analysis* is to study the effects of altering the outcome of a single gate in a circuit on the input-output behavior of the whole circuit.

There are many more applications of ordered branching programs (see Bryant [Bry92]) for an overview). Because of the restricted computational power of ordered branching programs, we are interested in more general models that are capable of computing more functions within reasonable size, such that the corresponding equivalence problem is still efficiently solvable. (This is, however, not the only property such a model should have in order to be useful, for example in hardware verification.)

In Chapter 4 we give an overview over the various restricted models of branching programs: besides the already mentioned *ordered and read-once branching programs* we consider *nondeterministic, parity, and probabilistic ordered branching programs*. For each of these models we study the complexity of the corresponding satisfiability and equivalence problems. From the viewpoint of applications, every model (or variants of it) has some advantage over the other models, but some serious drawbacks as well.

For example an extension of ordered branching programs are *parity ordered branching programs* that we consider in Section 4.5. In particular, they allow an efficient equivalence test. However, the current algorithms for dealing with them are much slower than the ones for ordered branching programs.

In Section 4.6 we show that the equivalence problem for *bounded-error probabilistic ordered branching programs* (BP-OBDDs for short) is hard (**coNP**-complete). However, if the error of a BP-OBDD is small compared to its size, then we have an efficient equivalence test.

Examining the existing branching program models and looking for new ones is a very active area of research these days. Still there are more problems open than solved.

1.2 Isomorphism Problems

Suppose we have constructed a circuit that computes some boolean function $f(x_1, \ldots, x_n)$. A trivial modification of the circuit is certainly to reorder (*permute*) the inputs so that, for example, x_1 changes its place with x_3, and so on. Let g be the function computed by the modified circuit. In general, f will not be equivalent to g. But since g was obtained so easily from f, we would still like to call them *almost equivalent*. Below we will see other modifications that are about as easy to accomplish as the above permutation. We start with the permutation.

We call two functions f and g *isomorphic*, if there exists a permutation of the input variables such that f and g become equivalent when the permutation is applied to the variables of f. The *isomorphism problem for boolean circuits* is to decide whether the functions given by two circuits are isomorphic. In an analog way we may consider the isomorphism problem for other computational models such as *boolean formulas* or *branching programs*.

In the next two subsections we describe some transformations that are more general than permutations, but still easy to accomplish. I.e., the resulting function is still (intuitively) *almost equivalent* to the given one.

1.2.1 Congruence Problems

Given some boolean function $f(x_1, \ldots, x_n)$, a *negation mapping* maps each variable x_i either to itself, x_i, or to its complement, $\overline{x}_i$. Two functions f and g are called *negation equivalent*, if there exists a negation mapping such that f and g become equivalent when the mapping is applied to the variables of f. Correspondingly we talk of the *negation equivalence problem for boolean circuits* which is to decide whether the functions given by two circuits are negation equivalent.

A *congruence mapping* is a composition of permutation followed by a negation mapping. We define two boolean functions to be *congruent* if there exists a congruence mapping that makes them equivalent. From this we obtain *congruence problems* for each of the various models to represent boolean functions.

To motivate the name, the boolean congruence relation can be seen as a *geometrical congruence*: there are 2^n assignments for a boolean function f over n variables, that form the nodes of a n-dimensional cube in $\mathbf{R}^n$. The assignments where f evaluates to one constitute a subgraph of the cube, *the n-dimensional geometrical cube that represents f*. Two functions f and g are congruent if and only if the n-dimensional geometrical cubes that represent f and g are geometrically congruent, that is, there is a distance-preserving bijection from one subgraph to the other.

1.2.2 Linear and Affine Equivalence Problems

The most general transformations are as follows. Any permutation of n variables $\boldsymbol{x} = (x_1, \dots, x_n)$ can be written as a product of a permutation matrix $\boldsymbol{P}$ with $\boldsymbol{x}$ over GF(2). That is, an isomorphism can be written as $\boldsymbol{xP}$, and a congruence can be written as $\boldsymbol{xP} + \boldsymbol{c}$, for a vector $\boldsymbol{c} \in \{0,1\}^n$. A natural generalization of the above notions is therefore to consider linear and affine transformations $\boldsymbol{xA}$ and $\boldsymbol{xA} + \boldsymbol{c}$, respectively, where $\boldsymbol{A}$ has to be a bijection on $\{0,1\}^n$. We call two formulas *linear equivalent* or *affine equivalent* if they become equivalent after a linear or an affine transformation of the variables of one of the formulas, respectively.

1.2.3 History

The origins of the isomorphism and congruence problems go back to the 19th century. The first papers were by Jevons [Jev92] and Clifford [Cli76], and subsequent authors called it the Jevons-Clifford Problem. The motivation at that time was about as follows: suppose we want to provide circuits for all n-ary boolean functions. But we don't want to build 2^{2^n} circuits, as there are as many boolean functions. Instead, we build a *basic set* of circuits such that we get the missing functions by modifying one of the basic circuits according to one of the above transformations. This is the point why the transformations are required to be easy: a small amount of extra hardware should suffice to obtain the desired circuit out of a basic circuit. Now the question is: how many basic circuits do we have to build?

Clearly, each of these transformations defines an equivalence relation on the n-ary boolean functions. Hence, another way of putting the above question is: how many equivalence classes do these relations have? It turned out that this is a very hard question. It took until 1937, when Pólya [Pól37, Pól40] obtained some major result in combinatorial group theory that one could obtain even *exact* numbers. In case of the negation equivalence, one can derive an explicit formula: the number of equivalence classes of boolean functions in n variables is

$$\frac{2^{2^n} + (2^n - 1)2^{2^{n-1}}}{2^n}.$$

However, for the other equivalence relations, there are no explicit formulas. But one can derive algorithms from Pólya's theorem, that involve generating functions, to compute these numbers. In Figure 1.1 we give the number of equivalence classes induced by the equivalence relations mentioned above for the first few values of n. A full treatment of Pólya's result and the applications to boolean functions can be found in an article by Harrison [Har71].

n	$\|B_n\| = 2^{2^n}$	isomorphism	negation equivalence
1	4	4	3
2	16	12	7
3	256	80	46
4	65,536	3,984	4,336
5	4,294,967,296	37,333,248	134,281,216
6	$18.4 \cdot 10^{18}$	25,626,412,338,274,304	288,230,380,379,570,176

(a)

n	congruence	linear equivalence	affine equivalence
1	3	4	3
2	6	8	5
3	22	20	10
4	402	92	32
5	1,228,158	2744	328
6	400,507,806,843,728	950,998,216	15,768,919

(b)

Fig. 1.1: The number of equivalence classes on the n-ary boolean functions, B_n, with respect to (a) isomorphism, negation, and (b) congruence, linear and affine transformations.

Considering Figure 1.1, it is tempting to think that indeed, the number of circuits needed to build in order to cover all n-ary boolean functions via transformations, has decreased considerably. For example, when we take the most flexible transformation, affine transformation, from the more than 4 Billion boolean functions with five inputs there are only 328 that are not equivalent.

However, note that if we consider a transformation group T, we have $2^{2^n}/\|T\|$ as a lower bound on the number of equivalence classes, because in the best case, we get $\|T\|$ pairwise different functions when we apply all the transformations of T to some function. This does definitely not hold in general. For example, when we apply permutations to a symmetric function, we get $n!$-times the same function. On the other hand, almost no boolean function is symmetric (2^{n+1}, to be precise). And indeed, Harrison [Har66] showed that the above example is quite exceptional: let T be any of the above transformation groups. Then the number of equivalence classes *asymptotically* matches the lower bound, $2^{2^n}/\|T\|$.

There are 2^n negation mappings, $n!$ isomorphisms, and $n!\,2^n$ congruences. The number of linear equivalences is the number non-singular $n\times n$-matrices over GF(2). There are $\prod_{k=0}^{n-1}(2^n - 2^k)$ such matrices which can be bounded by $\frac{1}{4}2^{n^2}$ (see Lemma 3.27). For the number of affine transformations, we get an additional factor of 2^n. Hence none of the above transformation groups has more than 2^{n^2} elements, and consequently, there are more than $2^{2^n-n^2}$ equivalence classes for each transformation. Thus, the maybe surprisingly small numbers of equivalence classes shown in Figure 1.1 are more an artifact of the small n considered there. The asymptotic behavior is still double exponential.

1.2.4 New Results and Background

In recent years, the problem of whether two given instances of a computational model are "almost" equivalent has been reconsidered from the complexity perspective (see for example [AT96, BR93, BRS98, CK91, Thi98]). A first step is to classify a new problem in the established complexity classes.

To do so, we extend the class **NP** to the *polynomial hierarchy* introduced by Meyer and Stockmeyer [MS72]. This hierarchy classifies sets according to the quantifier prefix needed to express membership in the set, where each quantifier can range over polynomially length bounded strings. The class **NP** is defined as all sets that can be written with a single existential quantifier. For example for the satisfiability problem for boolean formulas, Sat, we have

$$F \in \textsc{Sat} \iff \exists \boldsymbol{a} : \ \boldsymbol{a} \text{ is a satisfying assignment for } F.$$

The remaining predicate, that $\boldsymbol{a}$ satisfies formula F, can be verified efficiently.

The complementary class, **coNP**, is defined the same way, but with a universal quantifier instead of an existential quantifier. For example for the tautology problem for boolean formulas, Taut, we have

$$F \in \textsc{Taut} \iff \forall \boldsymbol{a} : \ \boldsymbol{a} \text{ is a satisfying assignment for } F.$$

The equivalence problems for circuits, formulas, and branching programs can all be expressed with a single universal quantifier: Two, say, circuits C_0, C_1 are equivalent, if *for all* inputs, they evaluate to the same value. Hence these problems are all located in **coNP**.

The next level in the hierarchy involves two quantifiers. The **Σ**-classes are the ones that start with an existential quantifier. That is, $\Sigma_2\mathbf{P}$ is the class of sets that can be expressed as $\exists\forall$. The complementary class that starts with the universal quantifier is denoted $\Pi_2\mathbf{P}$. (An alternative name for **NP** and **coNP** is therefore $\Sigma_1\mathbf{P}$ and $\Pi_1\mathbf{P}$, respectively.) Now it should be clear how to proceed to more complex classes $\Sigma_3\mathbf{P}$, $\Pi_3\mathbf{P}$, ...

For our next observation, consider some transformation group, say permutations. The isomorphism problem for circuits C_0, C_1 asks whether there *exists* a permutation of the variables of C_1 such that C_0 and C_1 become equivalent. Equivalence we can express with an universal quantifier. The isomorphism problem puts another existential quantifier in front. The same holds for the other transformations mentioned above. Therefore, all these problems are located in the class $\Sigma_2\mathbf{P}$.

One of the most fundamental notions in computer science is that of a *reduction*. It provides the technical framework to compare the complexity of seemingly different problems. For sets A and B, we say that A is *reducible to* B, if we can transform an instance x for A efficiently into an instance y for B such that $x \in A \iff y \in B$. That is, whatever algorithm will be developed for B, via the reduction, we can use it for A as well. In this sense, A is computationally at most as difficult as B.

Let $\mathcal{C}$ be a complexity class. A set A is *hard for* $\mathcal{C}$, if every set in $\mathcal{C}$ can be reduced to A. If, in addition, A itself is in $\mathcal{C}$, we call A *complete for* $\mathcal{C}$. It is well known that the equivalence problems for circuits, formulas and branching programs are all complete for **coNP**. If a problem is shown to be complete for such a class, then this is the best classification of its computational complexity we can give right now. Any further questions, like for example for the deterministic time complexity of the set, are connected to the big open questions in the theory of computations (like $\mathbf{P} = \mathbf{NP}$?). A natural question therefore is the following:

are the isomorphism or congruence problems
for formulas or circuits complete for $\Sigma_2\mathbf{P}$*?*

This question was solved by Agrawal and Thierauf [AT96]. It is shown that *none* of these isomorphism or congruence problems are complete for $\Sigma_2\mathbf{P}$, unless the polynomial hierarchy collapses (to $\Sigma_3\mathbf{P}$). The result builds on several powerful techniques that have been developed since the beginning of the eighties. Section 3.1 is devoted to a full proof of this result.

Note that most people in the field conjecture that the classes $\Sigma_k\mathbf{P}$ and $\Pi_k\mathbf{P}$ of the polynomial hierarchy are all different. Again, this is one of the challenging open problems. In particular, a proof would imply that $\mathbf{P} \neq \mathbf{NP}$. So this is an even harder task than "just" separating **P** from **NP**.

In converse, any assumption that implies a collapse of the polynomial hierarchy is taken as evidence that *the assumption is false*. Applying this principle to the above result about the isomorphism problems, none of these problems is expected to be complete for $\Sigma_2\mathbf{P}$.

Thus, loosely speaking, the existential quantifier we get by seeking for an isomorphism doesn't seem to add the full power such a quantifier can have to the corresponding equivalence problem. The most prominent example that supports the latter conjeecture might be the *graph isomorphism problem, GI*. Here, the equivalence problem for graphs, which is in fact an equality problem, is trivially solvable in polynomial time. Therefore GI is in **NP**. But GI is *not* complete for **NP** unless the polynomial hierarchy collapses (to $\Sigma_2\mathbf{P}$) [GMW91, BHZ87] (see also [Sch88]). [1]

In Chapter 4 we consider branching programs. For a restricted class of branching programs, the *read-once branching programs*,where, on each path, each variable is tested at most once, the equivalence problem is easier than for general ones: it can efficiently be solved by a randomized algorithm [BCW80], it is in the complexity class **coRP**. Therefore, putting an existential quantifier in front, the isomorphism problem for read-once branching programs is in $\mathbf{NP} \cdot \mathbf{coRP}$. Motivated by the examples above, we ask whether the problem is hard for **NP**. This question was solved by Thierauf [Thi98] in the following way: it is *not* **NP**-hard, unless the polynomial hierarchy collapses to $\Sigma_2\mathbf{P}$. We present these results in Section 4.1 and 4.2.

There is a great variety of branching programs in the literature. For example *ordered branching programs* are read-once branching programs with the additional restriction that the variable have to appear in a certain order. As for Turing machines, branching programs have been augmented by nondeterminism and probabilism. We will consider these models in detail in the remaining sections of Chapter 4.

[1]We should mention an exception to the above rule. In the special case that the graphs are trees, the isomorphism problem can be solved in logarithmic space, **L**, [Lin92].

Moreover, Jenner, Lange, and McKenzie [JLM97] have shown that it is also complete for **L**. So it seems that for the "small" complexity classes inside **P** things may behave differently. Moreover, the completeness result depends crucially on the representation of the trees (compare with [Bus97]).

Chapter 2

Preliminaries

We assume that the reader is familiar with the basic notions that are usually taught in a course on complexity theory, linear algebra, and probability theory. In particular, for computational models such as *Turing machines*, *boolean circuits* or *boolean formulas*, we refer to some standard textbook as for example [BDG88, BDG91, HU79, Pap94, Sip97] . The *languages* or *sets* we consider are sets of *strings* or *words* over the fixed *alphabet* $\Sigma = \{0, 1\}$. The empty string is denoted by λ. An *n-ary boolean function* $f = f(x_1, \ldots, x_n)$ is a mapping from $\{0,1\}^n$ to $\{0,1\}$.

2.1 Complexity Classes

The class of sets that can be decided by polynomial-time bounded deterministic Turing machines is denoted by **P**. It is considered as the class of problems that can be solved *efficiently*. From a practical point of view, one could object that a running time of order, say, n^{1000} is not very efficient. However, luckily, such high degree polynomials usually do not occur in practice.

For many important problems *no* efficient algorithm is known. An example is the satisfiability problem for boolean formulas, SAT. On the other hand, SAT can be decided efficiently by *nondeterministic* algorithms. **NP** is the class of sets that can be decided by polynomial-time bounded nondeterministic Turing machines.

For any class $\mathcal{C}$, we denote the complement class by $\mathbf{co}\mathcal{C}$. That is $\mathbf{co}\mathcal{C} = \{\overline{C} \mid C \in \mathcal{C}\}$. The tautology problem, TAUT, is an example for a set in **coNP**.

SAT and TAUT can be seen as certain predicates on the *number of satisfying assignments* of a formula. Namely, for a formula with n variables to have at least one, respectively precisely 2^n satisfying assignments. Another interesting problem therefore is to compute the number of satisfying assignments of a given formula. Observe that this is now a *function* and not a decision problem. More general, $\#\mathbf{P}$ is the class of functions that count the number of accepting computations of polynomial-time nondeterministic Turing machines.

There are several equivalent ways to define the *polynomial hierarchy* [MS72]. We have already mentioned the alternating quantifiers in Section 1.2.4.

A set L is in the class $\Sigma_k\mathbf{P}$, the *k-th level of the polynomial hierarchy*, if there is a polynomial p and a set $A \in \mathbf{P}$ such that for all $x \in \Sigma^*$

$$x \in L \iff \exists y_1\, \forall y_2\, \ldots Q_k y_k : \; (x, y_1, \ldots, y_k) \in A,$$

where $Q_k = \exists$ if k is odd, $Q_k = \forall$ if k is even, and the length of all the y_i's is bounded by $p(|x|)$. Classes $\Pi_k\mathbf{P}$ are defined the same way as classes $\Sigma_k\mathbf{P}$, but starting with a universal quantifier.

An equivalent definition is in terms of relativized classes which are based on *oracle Turing machines*.

$$\begin{aligned} \Sigma_1\mathbf{P} &= \mathbf{NP}, \\ \Sigma_{k+1}\mathbf{P} &= \mathbf{NP}^{\Sigma_k\mathbf{P}}, \end{aligned}$$

The complementary classes are $\Pi_k\mathbf{P} = \mathbf{co}\Sigma_k\mathbf{P}$, and the *polynomial hierarchy*, $\mathbf{PH}$, is the union of all the classes: $\mathbf{PH} = \bigcup_{k\geq 1}(\Sigma_k\mathbf{P} \cup \Pi_k\mathbf{P})$. We mention some basic facts about the hierarchy.

1. $\Sigma_k\mathbf{P} \cup \Pi_k\mathbf{P} \subseteq \mathbf{P}^{\Sigma_k\mathbf{P}} \subseteq \Sigma_{k+1}\mathbf{P} \cap \Pi_{k+1}\mathbf{P}$,
2. $\Sigma_k\mathbf{P} = \Pi_k\mathbf{P} \iff \Sigma_k\mathbf{P} = \Sigma_{k+1}\mathbf{P} \iff \mathbf{PH} = \Sigma_k\mathbf{P}$.

It follows from the second fact that if any two levels of the hierarchy coincide, the whole hierarchy collapses to that level. Most people in the field conjecture that the hierarchy does not collapse. Clearly it is hard to have a good intuition of what the difference is between, say $\Sigma_{99}\mathbf{P}$ and $\Sigma_{100}\mathbf{P}$ (who can think 100 quantifiers deep?). For our purposes, however, it is enough to think up to $\Sigma_3\mathbf{P}$ and, maybe, believe that it is different from $\Pi_3\mathbf{P}$. A proof of this conjecture would be a major break through.

PSPACE denotes the class of sets accepted by deterministic polynomial-space bounded Turing machines. All sets in the polynomial hierarchy can be computed in polynomial space, that is $\mathbf{PH} \subseteq \mathbf{PSPACE}$. It is not known whether $\mathbf{P}$ is different from **PSPACE**.

2.2 Reductions

The origins of the reducibility notion go back to recursion theory. There is a big variety of different kinds of reductions in the literature.

In the above definition of the polynomial hierarchy we have already considered classes such as $\mathbf{P}^{\mathbf{NP}}$, the class of sets that can be computed in polynomial time with the help of an **NP**-oracle. This is called a *Turing reduction* to **NP**: for sets A and B, we say that A is *(polynomial-time) Turing reducible to* B (and write $A \leq_T^p B$) if there is some polynomial-time bounded Turing machine M such that $A = L(M^B)$.

A more restricted kind of reducibility is *many-one reducibility*, where one can ask only one query to the oracle: we say that A is *(polynomial-time) many-one reducible to* B (and write $A \leq_m^p B$) if there is some polynomial-time computable function f such that for any $x \in \Sigma^*$, we have

$$x \in A \iff f(x) \in B.$$

For a class $\mathcal{C}$ of sets we say that B *is hard for* $\mathcal{C}$, if $A \leq_m^p B$ for every set $A \in \mathcal{C}$. If, in addition, $B \in \mathcal{C}$, then B is called *complete* for $\mathcal{C}$.

All classes defined in the previous section have complete sets. SAT, the satisfiability problem for boolean formulas is **NP**-complete, and the tautology problem, TAUT, is **coNP**-complete. By using quantified formulas with a bounded number of quantifier alternations we obtain complete sets for the levels of the polynomial hierarchy. Without restriction on the number of quantifier alternations, the satisfiability problem for quantified boolean formulas, QBF, is **PSPACE**-complete.

2.3 Probabilistic Complexity Classes

We will also consider *randomized algorithms*. The basic idea is to replace the existential quantifier in nondeterministic computations by probabilistic requirements. From a configuration of a nondeterministic Turing machine, the machine can have several possibilities (w.l.o.g. we can assume two possibilities) for the next move. In the nondeterministic case, we ask whether at least one of these moves leads to an accepting configuration. In the case of a *probabilistic Turing machine* M, we consider each nondeterministic branch as a probability experiment, where each of the two possible moves can be chosen with probability 1/2. Then M can be seen as a random variable and consequently we write $\mathbf{Pr}[M(x) \text{ accepts}]$ for the probability that M accepts input x.

Probabilistic algorithms are distinguished according to what kind of errors they make. A randomized algorithm is called a *Monte Carlo algorithm* if it errs only with small probability.

Definition 2.1 (Bounded-error probabilistic polynomial time) **BPP** *is the class of sets L such that there exists a probabilistic polynomial-time bounded Turing machine M such that for every x*

$$\begin{aligned} x \in L & \implies & \mathbf{Pr}[M(x) \text{ accepts}] & \geq 3/4, \\ x \notin L & \implies & \mathbf{Pr}[M(x) \text{ accepts}] & \leq 1/4. \end{aligned}$$

In other words, the error probability of M is bounded by $1/4$. The error is *two-sided*: M can err for $x \in L$ as well as for $x \notin L$. The correctness of M can be *amplified* by performing several independent iterations of M on the same input and taking as output the majority of the outputs of these iterations. Each iteration is a Bernoulli experiment with parameter $p = \mathbf{Pr}[M(x) \text{ outputs correctly}] \geq 3/4$. Therefore, when performing t independent iterations, the corresponding random variable is binomially distributed. Thus we expect to get tp correct answers, while the majority vote checks which answer was given more than $t/2$ times. Since p is bounded away from $1/2$, the majority vote errs only when the results of the independent iterations deviate more than a factor $1/2p < 1$ from their expected value. By the Chernoff bounds for the binomial distribution we get an exponentially small error, 2^{-ct}, for this event, for some constant $c > 0$. In fact the same argument works if we start with error parameters $\frac{1}{2} + \frac{1}{p(n)}$ and $\frac{1}{2} - \frac{1}{p(n)}$ instead of $3/4$ and $1/4$ in Definition 2.1, for some polynomial p and $|x| = n$.

Below we give an elementary proof how to bound the error probability of the majority vote, i.e. without using the Chernoff bound (see [Sch85]).

Lemma 2.2 *Let E be some event that occurs with probability at least $p > 1/2$ and let X_t be the random variable that counts how often E occurs in t independent trials. Then*

$$\mathbf{Pr}[X_t \geq t/2] \quad \geq \quad 1 - 2^{-ct}$$

for $c = -\log(2\sqrt{p(1-p)})$ where $p < 1$.

Proof X_t is binomially distributed: $\mathbf{Pr}[X_t = k] = \binom{t}{k} p^k \, (1-p)^{t-k}$. Therefore,

$$\begin{aligned}
\mathbf{Pr}[X_t < t/2] &\leq \sum_{k=0}^{(t-1)/2} \binom{t}{k} p^k \, (1-p)^{t-k} \\
&= (1-p)^t \sum_{k=0}^{(t-1)/2} \binom{t}{k} \left(\frac{p}{1-p}\right)^k \\
&\leq (1-p)^t \left(\frac{p}{1-p}\right)^{t/2} \sum_{k=0}^{(t-1)/2} \binom{t}{k} \\
&\leq (p\,(1-p))^{t/2}\, 2^{t-1} \\
&= \frac{1}{2} (4\,p\,(1-p))^{t/2} \\
&\leq 2^{-ct},
\end{aligned}$$

for some constant $c > 0$. The last inequality holds because $p(1-p) < 1/4$ for $p > 1/2$. □

If the original **BPP**-machine M uses r random bits to decide an input x, then we need tr random bits for the majority vote on t iterations of M on input x. Since the production of random bits is expensive (if not impossible), one is seeking for amplification methods that consume less random bits. The currently best construction is due to Zuckerman [Zuc96] (see [GZ97]). It uses only $r + c't$ random bits for any constant $c' > 1$, and achieves an error probability bounded by 2^{-t}.

Often a probabilistic algorithm has only *one-sided error*, for example such that the algorithm errs only on negative instances: if it accepts, then this is always correct. Only in the case that the algorithm rejects there is a small chance that the answer is wrong.

Definition 2.3 (Randomized polynomial time) **RP** *is the class of sets L such that there exists a probabilistic polynomial-time bounded Turing machine M such that for every x*

$$\begin{aligned}
x \in L &\implies \mathbf{Pr}[M(x) \text{ accepts}] \geq 1/2, \\
x \notin L &\implies \mathbf{Pr}[M(x) \text{ accepts}] = 0.
\end{aligned}$$

Amplification is much easier than in the case of two-sided error: again we consider t independent runs of M. Since M never accepts an input x that is not in L, we will accept x as soon as M accepted on at least one of the t independent runs. That way, we still never accept an x not in L and will err only with probability 2^{-t} if x is in L.

As an **RP**-algorithm can err only on positive instances for a set L, a **coRP**-algorithm can err only on negative instances. If L belongs to both **RP** and **coRP**, it can be solved by *zero-sided error* randomized algorithms. Such algorithms are sometimes called *Las Vegas algorithms*.

Definition 2.4 (Zero-error probabilistic polynomial time) $\mathbf{ZPP} = \mathbf{RP} \cap \mathbf{coRP}$.

From a practical point of view randomized algorithms are an attractive alternative to standard deterministic algorithms, because often they are much simpler and faster than their deterministic counterparts (if there are any). The error can be reduced to an acceptable size without slowing down the algorithm too much.

It is an open problem whether one can compute *more* by using randomization than without, i.e., whether **P** is different from **ZPP**, **RP**, or **BPP**. On the other hand, whether one can solve an **NP**-complete problem within **BPP** is open as well. However, the latter seems to be very unlikely: Karp and Lipton [KL82] showed that this would imply a collapse of the polynomial hierarchy to the second level, $\mathbf{PH} = \Sigma_2\mathbf{P}$.

2.4 Probabilistic Quantifiers

The polynomial hierarchy is defined by alternating existential and universal quantifiers. Below we use **NP** and **coNP** as *operators* that replace the quantifiers.

$\mathbf{NP} \cdot \mathcal{C}$ is the class of sets L such that there exists a set $A \in \mathcal{C}$ and a polynomial p such that for every x, we have

$$x \in L \iff \exists y \in \Sigma^{p(|x|)} : (x, y) \in A.$$

co$\mathbf{NP} \cdot \mathcal{C}$ is defined analogously, but with a universal quantifier instead. Then we can write for example

$$\begin{aligned} \Sigma_2\mathbf{P} &= \mathbf{NP} \cdot \mathbf{coNP}, \\ \Sigma_3\mathbf{P} &= \mathbf{NP} \cdot \mathbf{coNP} \cdot \mathbf{NP}, \end{aligned}$$

and so on.

In the same way, we can use other complexity classes as operators as well. Via probabilistic complexity classes we get a notion of *probabilistic quantifiers*.

$\mathbf{BP} \cdot \mathcal{C}$ is the class of sets L such that there exists a set $A \in \mathcal{C}$ and a polynomial p such that for every x

$$\begin{aligned} x \in L &\implies \mathbf{Pr}[(x,y) \in A] \geq 3/4, \\ x \notin L &\implies \mathbf{Pr}[(x,y) \in A] \leq 1/4, \end{aligned}$$

where y is chosen uniformly at random from $\Sigma^{p(|x|)}$.

The probabilistic quantifier generalizes the step when going from **P** to **BPP** to the class $\mathcal{C}$. Interesting classes we obtain that way are for example $\mathbf{BP} \cdot \mathbf{NP}$, $\mathbf{BP} \cdot \mathbf{coNP}$, ... Schöning [Sch89] showed that $\mathbf{BP} \cdot \Sigma_k\mathbf{P} \subseteq \Pi_{k+1}\mathbf{P}$. He also showed that within $\mathbf{BP}\cdot\Sigma_k\mathbf{P}$ one cannot decide a $\Sigma_{k+1}\mathbf{P}$-complete problem, unless the polynomial hierarchy collapses:

Theorem 2.5 *For all $k \geq 1$ we have*

1) $\mathbf{BP} \cdot \Sigma_k\mathbf{P} \subseteq \Pi_{k+1}\mathbf{P}$,
2) $\Pi_k\mathbf{P} \subseteq \mathbf{BP} \cdot \Sigma_k\mathbf{P} \implies \mathbf{PH} = \Sigma_{k+1}\mathbf{P}$.

2.5 Interactive Proofs

An intuitive way of viewing **NP** is as the class of languages that have *short proofs of membership*. That is, a "proof" that x is in L is a string y such that $(x,y) \in A$, for some appropriate set $A \in \mathbf{P}$ and y of length polynomial in the length of x. Unless $\mathbf{P} = \mathbf{NP}$ we cannot compute such a proof in polynomial time in general. A slight tweak of this view on **NP** is as follows: a powerful *prover* can provide a proof of membership y to a polynomial-time *verifier*, who can check that y is indeed a proof.

This setting was generalized to *interactive proofs* by Goldwasser, Micali, and Rackoff [GMR89] . They allowed several rounds of communication between the prover and the verifier (not just 1 message as above). Furthermore, the verifier can use randomization (instead of being deterministic as above) and has to be convinced by overwhelming statistical evidence.

Definition 2.6 *An* interactive proof system *for a set L consists of a* prover *P and* verifier *V. The verifier is a randomized polynomial-time algorithm that can communicate with the prover. The prover can make arbitrary computations. After following some communication protocol, the verifier finally has to accept or reject a given input such that*

$$\begin{aligned} x \in L &\implies \exists \textit{ prover } P: \ \mathbf{Pr}[(V,P)(x) \textit{ accepts}] &= 1, \\ x \notin L &\implies \forall \textit{ prover } P: \ \mathbf{Pr}[(V,P)(x) \textit{ accepts}] &\leq 1/2, \end{aligned}$$

where the probability is taken over the random choices of the verifier.

IP *denotes the class of sets that have an interactive proof system.* **IP**$[k]$ *is the subclass of* **IP** *where the verifier and the prover exchange at most* k *messages.*

Clearly **NP** $\subseteq$ **IP**, in fact **IP**$[1]$ already captures **NP**. An obvious question is whether there are any languages *not* in **NP** that have interactive proofs. Consider **coNP**. In order to convince a verifier that a string is in a **coNP**-set it seems as one has to send almost all strings that could potentially prove the opposite. But these are exponentially many and hence this is not a *short* proof.

A first step to solve this question was already done in the first papers [GMR89, GMW91]. There it is shown that that the *graph non-isomorphism problem*, GNI, and the *quadratic non-residue* problem have constant round interactive proofs. Both problems are in **coNP** and not known to be in **NP**. As an example, we give the protocol for GNI in Figure 2.1. The input are two graphs (G_0, G_1) that each have n nodes.

- [V:] The verifier randomly picks $i \in \{0,1\}$ and a permutation π on $\{1,\dots,n\}$, and computes the graph $H = \pi(G_i)$ In other words, H is a random isomorphic copy of a randomly selected input graph. The verifier sends H to the prover.
- [P:] The prover is expected to tell the verifier which input graph was used to construct H. Therefore she sends $j \in \{0,1\}$ to the verifier.
- [V:] Finally, the verifier accepts if the prover gave the correct answer, i.e., if $i = j$, and rejects otherwise.

Fig. 2.1: Interactive protocol for the graph non-isomorphism problem

Theorem 2.7 *The protocol shown in Figure 2.1 constitutes an interactive proof system for* GNI*. Therefore* GNI $\in$ **IP**$[2]$.

Proof When the input graphs are *not* isomorphic, the prover can find out from which of G_0 or G_1 the graph H was obtained by the verifier, and can therefore make the verifier accept with probability one (recall that the computational power of the prover is not limited, therefore she can solve GNI).

On the other hand, if the graphs are isomorphic, H can be constructed from both, G_0 and G_1. Therefore no prover can find out the graph that was chosen by the verifier to construct H. Hence, the answer of any prover is correct with probability at most 1/2. $\square$

For the correctness of the above protocol it is important that the random bits used by the verifier to choose i and π are *private*: the prover does not know them. The corresponding model with *public* random bits was introduced by Babai and Moran [BM88] and called *Arthur-Merlin games* with Arthur corresponding to the verifier and Merlin to the prover. That is, Arthur-Merlin games are defined exactly the same way as interactive proof system but with the random bits accessible by both, Arthur and Merlin.

$\mathbf{AM}[k]$ denotes the class of sets that have an Arthur-Merlin game where k many messages can be send (starting with Arthur). Recall that **IP** denotes interactive proof systems with an unbounded number of rounds. In contrast, **AM** is defined to be $\mathbf{AM}[2]$. Symmetrically, $\mathbf{MA}[k]$ and $\mathbf{MA} = \mathbf{MA}[1]$ denote the same classes but with Merlin starting the communication.

MA is the smallest probabilistic extension of **NP** with respect to interactive proofs: Merlin sends a proof to Arthur, who can check it probabilistically (instead of deterministically as in the case of **NP**). We have

$$\mathbf{NP} \cdot \mathbf{BPP} \ \subseteq \ \mathbf{MA} \ \subseteq \ \mathbf{AM}.$$

Note that Arthur must not be a **BPP**-predicate: only on the instances provided by Merlin Arthur must have bounded error. It is actually not known whether the first inclusion is strict or not. The second inclusion is simply an exchange of quantifiers which works here because one can amplify the probability of correctness in **MA**.

There was a series of surprising results about the relationships and the power of these classes. The ability of having *private* random bits seems to be quite crucial in the above protocol. However, Goldwasser and Sipser [GS89] showed that for any interactive proof system with k messages exchanged (k can be a function depending on the input length), there is an Arthur-Merlin game with $k+2$ messages that accepts the same set. In other words

$$\mathbf{IP}[k] \subseteq \mathbf{AM}[k+2].$$

Thus a private coin is not much better than a public coin.

Furthermore Babai and Moran [BM88] showed that any constant round Arthur-Merlin game can be turned into one with just two messages sent: $\mathbf{AM}[k] = \mathbf{AM}$, for any $k \geq 2$. Taking these results together, we have

Theorem 2.8 $\mathbf{IP}[k] = \mathbf{AM}$ *for any* $k \geq 2$.

In particular, since there are two messages sent in the above protocol for GNI, we have GNI $\in \mathbf{AM}$.

AM can be characterized in terms of probabilistic quantifiers (see Section 2.4).

Proposition 2.9 $\mathbf{AM} = \mathbf{BP} \cdot \mathbf{NP} \subseteq \Pi_2\mathbf{P}$.

Proof The claim is trivial when we allow **AM** to have two-sided error as well. In this case, the **BP**-quantifier corresponds to Arthur's move which consists of simply sending random bits. For most of them, Merlin has to give an answer such that Arthur accepts. Merlin's move together with Arthur's evaluation corresponds to the **NP** set.

The proof that $\mathbf{BP} \cdot \mathbf{NP}$ can be turned to one-sided error, i.e., that $\mathbf{BP} \cdot \mathbf{NP} = \mathbf{coRP} \cdot \mathbf{NP}$ is not as obvious. The idea due to Sipser [Sip83] is to use universal hashing to approximate the acceptance ratio of Arthur. This has the property that it detects a high acceptance ratio with probability 1. We complete the proof at the end of Section 3.1.5 on page 50.

The inclusion in $\Pi_2\mathbf{P}$ follows because $\mathbf{coRP} \subseteq \mathbf{coNP}$ and hence $\mathbf{coRP} \cdot \mathbf{NP} \subseteq \mathbf{coNP} \cdot \mathbf{NP} = \Pi_2\mathbf{P}$. □

From Theorem 2.8 and Proposition 2.9 we learn that the computational power of constant round interactive proofs just slightly extend **NP**. This intuition was also supported by Boppana, Håstad, and Zachos [BHZ87] who showed that **coNP** does not have constant round interactive proofs, unless the polynomial hierarchy collapses. Note that Theorem 2.5 generalizes this result.

Theorem 2.10 $\mathbf{coNP} \subseteq \mathbf{BP} \cdot \mathbf{NP} \Longrightarrow \mathbf{PH} = \Sigma_2\mathbf{P}$.

Recall that the graph non-isomorphism problem, GNI, is in **coNP** and in $\mathbf{BP} \cdot \mathbf{NP}$, but not known to be in **NP**. By Theorem 2.10, GNI cannot be **coNP**-complete, unless the polynomial hierarchy collapses. Therefore the graph isomorphism problem, GI, cannot be **NP**-complete, unless the polynomial hierarchy collapses.

Corollary 2.11 GI *cannot be* **NP***-complete, unless* $\mathbf{PH} = \Sigma_2\mathbf{P}$.

Theorem 2.8 relativizes. For example the class $\mathbf{BP} \cdot \Sigma_2\mathbf{P}$ is the same as the class of sets that have constant round interactive proof systems, where the verifier has access to an **NP**-oracle (the prover does not need such an oracle). We show in Chapter 3 that there is such a proof system for the *non-isomorphism problem for boolean formulas*, FNI. Since $\mathrm{FNI} \in \Pi_2\mathbf{P}$, we can argue along the same lines as for GI above (but using Theorem 2.5) and obtain the following result (this is repeated as Corollary 3.25 in Section 3.1.4):

Theorem 2.12 *The* isomorphism problem for boolean formulas*,* FI*, cannot be* $\Sigma_2\mathbf{P}$*-complete unless the polynomial hierarchy collapses to* $\Sigma_3\mathbf{P}$*.*

What about the computational power of interactive proofs with an *unbounded* number of communications? Fortnow and Sipser [FS88] constructed an oracle A such that $\mathbf{coNP}^A \not\subseteq \mathbf{IP}^A$. Proofs that show an inclusion of one complexity class in another usually *relativize*. That is, the same proof goes through in the presence of an oracle. The result of Fortnow and Sipser lead to the conjecture that **coNP**-complete languages do *not* have (unbounded round) interactive proof systems. Therefore it was a big surprise when Lund, Fortnow, Karloff, and Nisan [LFKN92] came up with interactive proof systems not only for **coNP**, but even for #**P**. Then Shamir [Sha92] noticed that their technique works up to **PSPACE**, thereby precisely characterizing the computational power of **IP**.

Theorem 2.13 $\mathbf{IP} = \mathbf{PSPACE}$*.*

2.6 Isomorphisms and Other Transformations

Boolean functions are not represented *uniquely* by formulas or circuits. For example the formulas

$$1,\ x \vee \overline{x},\ (x \vee \overline{x}) \wedge (x \vee \overline{x}), \ldots$$

all represent the same function. The problem to decide whether two given representations describe in fact the same boolean function is called the *equivalence problem* of that model. In the following we only mention boolean formulas. But we could use boolean circuits as well.

Two formulas F and G are equivalent if they are defined over the same set of variables and evaluate to the same value for *all* assignments to their variables. By FE we denote the equivalence problem for formulas. Equivalence we denote by $\equiv$. Thus we can write

$$\mathrm{FE} \;=\; \{\,(F,G) \mid F \equiv G\,\}.$$

FE $\in$ **coNP** and Taut many-one reduces to FE via the reduction $F \mapsto (F,1)$. Hence, FE is **coNP**-complete.

Two formulas F and G with variables $\{x_1,\ldots,x_n\}$ are *isomorphic*, denoted by $F \cong G$, if there exists a permutation φ on $\{x_1,\ldots,x_n\}$, such that $F \equiv G \circ \varphi$, where $G \circ \varphi$ is the formula obtained from G by permuting the variables according to φ. In this case, we call φ an *isomorphism* *between F and G*. $\mathbf{Iso}(F, G)$ denotes the set of isomorphisms between F and G. The *formula isomorphism problem* is

$$\mathrm{FI} \quad = \quad \{\, (F, G) \mid F \cong G \,\}.$$

The corresponding *non-isomorphism* problem is denoted by FNI.

A related problem is the *boolean automorphism problem*, FA. A permutation α of the variables of formula F is called an *automorphism*, if $F \equiv F \circ \varphi$. By $\mathbf{Aut}(F)$ we denote the set of automorphisms of F.

Every formula has a trivial automorphism: the identity permutation, *id*. The automorphism problem asks whether a formula has a *non-trivial* automorphism.

$$\mathrm{FA} \quad = \quad \{\, F \mid \|\mathbf{Aut}(F)\| > 1 \,\}.$$

From the definition we see immediately that FA and FI are in $\Sigma_2\mathbf{P}$.

A *negation mapping* on n variables is a function ν such that $\nu(x_i) \in \{x_i, \overline{x}_i\}$ for $1 \leq i \leq n$. Two formulas F and G are *congruent*, if there exists a permutation φ and a negation mapping ν on $\{x_1, \ldots, x_n\}$, such that $F = G \circ \nu \circ \varphi$. The *formula congruence problem*, FC, is the set of pairs of formulas that are congruent,

$$\mathrm{FC} \quad = \quad \{\, (F, G) \mid F \text{ congruent } G \,\}.$$

Finally we define the most general transformations. Formulas F and G are *affine equivalent* if there exists a non-singular $n \times n$ matrix $\boldsymbol{A}$ and a $1 \times n$ vector $\boldsymbol{c}$ over GF(2), the Galois field with two elements, such that for every $\boldsymbol{x} = (x_1, \ldots, x_n)$, we have $F(\boldsymbol{x}) \equiv G(\boldsymbol{x}\boldsymbol{A} + \boldsymbol{c})$ (here addition and multiplication is over GF(2)). The formulas are *linear equivalent* if they are affine equivalent with the vector c being zero.

Chapter 3

Boolean Formulas and Circuits

In this chapter we study the isomorphism problem for boolean formulas, FI. We extend the results to circuits and also to the more general transformations defined in Section 2.6.

As an upper bound on its complexity we have FI $\in \Sigma_2\mathbf{P}$. As a lower bound, we will show in Section 3.2 that it is **coNP**-hard. But FI is not known to be in **coNP**. An obvious question is whether it is complete for $\Sigma_2\mathbf{P}$. This was posed as an open problem by Borchert, Ranjan, and Stephan [BRS98]. An answer was given by Agrawal and Thierauf [AT96] as follows: FI is *not* complete for $\Sigma_2\mathbf{P}$, unless the polynomial hierarchy collapses to its third level. In Section 3.1 we give a full proof of this result.

Almost all natural problems are complete for some of the established complexity classes (under some suitable type of reduction). Problems that do not fall into one of these categories, so called *intermediate problems*, are therefore very interesting. Right now it seems as FI is such an intermediate problem between **coNP** and $\Sigma_2\mathbf{P}$. Another such problem seems to be the graph isomorphism problem, GI, as we already outlined in Section 2.5. In Section 3.2 we compare the complexity of FI with other problems in more detail.

The formula isomorphism problem shares many similarities with the graph isomorphism problem, GI (see [Hof82] and [KST93] for a comprehensive treatment of the graph isomorphism problem). Many of the results for GI carry over to FI with similar proofs, although with some crucial differences.

3.1 An Interactive Proof for FNI

We show that there is a one round interactive proof system for the boolean formula non-isomorphism problem, FNI, where the verifier has access to an **NP**-oracle, i.e., FNI $\in \mathbf{IP}[2]^{\mathbf{NP}} = \mathbf{BP} \cdot \Sigma_2\mathbf{P}$. By the discussion in Section 2.5 (on page 20), it follows that FI cannot be $\Sigma_2\mathbf{P}$-complete unless the polynomial hierarchy collapses to $\Sigma_3\mathbf{P}$.

Our interactive proof is based on the one for GNI presented in Section 2.5. However, unfortunately the analogous protocol for FNI does not work. To see this, consider the protocol on input of two formulas (F_0, F_1), where

$$\begin{aligned} F_0 &= x_1 \wedge (\overline{x}_1 \vee \overline{x}_2), \quad \text{and} \\ F_1 &= \overline{x}_1 \wedge x_2. \end{aligned}$$

Note that F_0 and F_1 are isomorphic (exchange x_1 and x_2). The verifier randomly picks $i \in \{0, 1\}$, obtains a formula G by randomly permuting the variables of F_i and sends it to the prover. However, even though F_0 and F_1 are isomorphic, the prover can easily detect from which one G has been obtained, because of the *syntactic structure of* G: any permutation of F_0 will have three literals and any permutation of F_1 will have two literals.

It seems as what we need is a *normal form* for equivalent boolean formulas that can be computed by the verifier. Then the verifier could map the formula G in the above protocol to its normal form G', and the prover could not distinguish any more whether G' was obtained from F_0 or F_1 if the formulas are isomorphic.

Clearly, we cannot simply use the disjunctive or conjunctive normal form because this might lead to exponentially longer formulas. Another obvious candidate for a normal form is the *minimal equivalent boolean formula* (under some suitable total order). However, it requires a $\Sigma_2\mathbf{P}$-oracle to compute this [Uma98], and our verifier only has an **NP**-oracle available.

To overcome this difficulty, we compute what can be called a *randomized normal form* of a formula. It is obtained from an algorithm in learning theory by Bshouty *et al.* [BCG$^+$96]. We start by providing an informal description of the learning scenario and then give a reformulation in pure complexity theoretic terms.

3.1.1 A Randomized Normal Form

Bshouty *et al.* [BCG+96] show that boolean formulas can be learned by a probabilistic polynomial-time algorithm with equivalence queries and access to an **NP**-oracle.

The scenario is roughly as follows. Let F be a boolean formula given in a *black box*. A probabilistic polynomial-time machine, the *learner*, has to compute a formula that is equivalent to F, but without seeing F. The learner can use an **NP**-oracle, and furthermore ask *equivalence queries* to a *teacher* who knows F. That is, the learner can send a formula G to the teacher. If F and G are equivalent, the learner has succeeded in learning F and the teacher will answer 'yes'. Otherwise, the teacher will send a *counterexample* to the learner, namely an assignment $\boldsymbol{a}$ such that $F(\boldsymbol{a}) \neq G(\boldsymbol{a})$. The learner *succeeds in learning* F, if he outputs, with high probability, a formula that is equivalent to F. The learner might sometimes fail to learn F, but only with small probability. In this case, he makes no output. The result of Bshouty *et al.* [BCG+96] says that boolean formulas can be learned in this setting.

How can we use this for our normal form? The crucial observation in the learning process is that the output of the learner does *not* depend on the specific syntactic form of the input formula F: because of the black box approach, the learner has exactly the same behavior on every formula F' given as input that is equivalent to F. *Hence, we take the output of the learner as our normal form.*

Note that this is not a normal form in the classical sense, because the learner produces possibly different equivalent formulas for different random choices. However, on each random path the output remains the same on any F' as input that is equivalent to F. This will suffice for our purposes.

We want to reformulate the result in complexity theoretic terms. Our first step is to throw out the teacher. She has to decide the equivalence of formulas and to compute counterexamples. The latter can easily be done within $\mathbf{P}^{\mathbf{NP}}$ by a method called *prefix-search*: given two formulas F and G, extend, bit by bit, a partial assignment to the variables that is a prefix of an assignment where F and G differ. Each bit can be obtained with a query to an **NP**-oracle. A more formal description of the algorithm in shown in Figure 3.1.

Consider COUNTEREXAMPLE on input of two formulas F and G that are not equivalent. In each iteration of the for-loop, the initial assignment $(a_1, \ldots, a_{i-1})$ is extended such that the formulas on the remaining, unassigned variables are still not equivalent. (In line 2 we use the **NP**-oracle

```
COUNTEREXAMPLE ($F(x_1,\dots,x_n), G(x_1,\dots,x_n)$)
1  for $i \leftarrow 1$ to $n$ do
2      if $F(a_1,\dots,a_{i-1},0,x_{i+1},\dots,x_n) \not\equiv G(a_1,\dots,a_{i-1},0,x_{i+1},\dots,x_n)$
3      then $a_i \leftarrow 0$
4      else $a_i \leftarrow 1$
5  return $\boldsymbol{a} = (a_1,\dots,a_n)$
```

Fig. 3.1: COUNTEREXAMPLE computes an assignment where formulas F and G differ.

to test the non-equivalence of two boolean formulas.) Therefore we have $F(\boldsymbol{a}) \neq G(\boldsymbol{a})$ for the output $\boldsymbol{a}$ of COUNTEREXAMPLE.

In summary, we get *one* probabilistic algorithm that, with the help of an **NP**-oracle, simulates the learner *and* the teacher. The output is, with high probability, a formula equivalent to the input formula, and, with small probability, there is no output. Moreover, the output does not depend on the syntax of the input formula. More precisely, we can restate the result of Bshouty *et al.* [BCG+96] as follows.

Theorem 3.1 *There is a probabilistic polynomial-time algorithm* RANDOMIZED-NORMAL-FORM (F) *that has access to an* **NP***-oracle and, on input of a boolean formula F, has the following properties:*

- RANDOMIZED-NORMAL-FORM (F) *outputs a boolean formula that is equivalent to F with probability at least $3/4$, and makes no output otherwise,*
- *if F' is a formula equivalent to F, then, for any choice of the random bits used by* RANDOMIZED-NORMAL-FORM*, it makes the same output on input F as on input F'.*

The basic idea of the algorithm is fairly simple. Let F be a fixed input formula in n variables, $|F| = m$. Suppose we already got some counterexamples

$$\mathcal{C} \quad = \quad \{\, (\boldsymbol{a}_i, b_i) \mid F(\boldsymbol{a}_i) = b_i, \text{ for } i = 1,\dots,k \,\}.$$

Furthermore, let CONS($\mathcal{C}$) be the set of formulas H in n variables of size at most m that are *consistent with F on $\mathcal{C}$*. That is

$$\textsc{Cons}(\mathcal{C}) \quad = \quad \{\, H \mid |H| \leq m \text{ and } \forall (\boldsymbol{a}, b) \in \mathcal{C} : \ H(\boldsymbol{a}) = b \,\}.$$

Initially $\mathcal{C}$ is empty and CONS($\mathcal{C}$) contains all formulas up to the size of F. The goal is to eliminate some fraction of the formulas in CONS($\mathcal{C}$) that are not equivalent to F.

Let $\textsc{Cons}(\mathcal{C}) = \{H_1, \ldots, H_T\}$ for some $T \geq 1$. Consider the formula G that is defined as the majority , on a given assignment, outputs of the H_i's:

$$G \quad = \quad \text{majority}(H_1, \ldots, H_T).$$

G can be written as a boolean formula, i.e., with two-ary boolean operations, of size polynomial in the size of the input formulas.

Suppose $G \not\equiv F$ and let $\boldsymbol{a} \in \{0,1\}^n$ be a counterexample, so that $G(\boldsymbol{a}) \neq F(\boldsymbol{a})$. By the construction of G, more than half of the formulas in $\textsc{Cons}(\mathcal{C})$ must differ from F on $\boldsymbol{a}$. Therefore, when we add the pair $(\boldsymbol{a}, F(\boldsymbol{a}))$ to $\mathcal{C}$, the size of $\textsc{Cons}(\mathcal{C})$ decreases by a factor of at least $1/2$.

Initially $\textsc{Cons}(\mathcal{C})$ has at most $2^{|F|+1}$ elements. Hence, after at most $|F| + 1$ iterations of this procedure, we end up with a formula G that is equivalent to F. Note that the syntactic form of F does not play a role in this procedure.

However, we have not yet proven Theorem 3.1. This is because $\textsc{Cons}(\mathcal{C})$ has exponential size in the beginning and therefore our procedure has exponential running time. Note that G is constructed from *all* formulas in $\textsc{Cons}(\mathcal{C})$.

Here is an idea to get around this problem: recall that G is a majority vote on these functions. We know from statistics that when we do the vote just on a small (but large enough) random sample, we obtain a fairly good approximation of the total vote. Note that it is enough to eliminate some constant fraction, say $1/4$, of the functions in $\textsc{Cons}(\mathcal{C})$, not necessarily $1/2$ as above. This slows down the algorithm only by a constant factor, $1/\log(4/3) < 2.5$ in the case of $1/4$.

Definition 3.2 $G = \text{majority}(H_1, \ldots, H_t)$ *is called* good for $\textsc{Cons}(\mathcal{C})$ with respect to F, *if any counterexample to F eliminates at least* $1/4$ *of the formulas in* $\textsc{Cons}(\mathcal{C})$

A sample of size $t = 10m$ provides already a good approximation to the total majority vote, with high probability.

Lemma 3.3 (Sampling Uniformly) *Let $H_1, \ldots, H_t$ be chosen independently and uniformly at random from* $\textsc{Cons}(\mathcal{C})$ *and define formula* $G = \text{majority}(H_1, \ldots, H_t)$, *where $t \geq 10m$. Then*

$$\mathbf{Pr}[G \textit{ is good}] \quad \geq \quad 1 - 2^{-m}.$$

Proof Let $(\boldsymbol{a}, b)$ be a counterexample that eliminates less than $1/4$ of the functions in $\textsc{Cons}(\mathcal{C})$. Hence, less than $1/4$ of the functions take value

$1 - b$ on $\boldsymbol{a}$. In other words, when we select some $H \in \textsc{Cons}(\mathcal{C})$ uniformly at random, we have $\mathbf{Pr}[H(\boldsymbol{a}) = 1 - b] \leq 1/4$. From Lemma 2.2 we get that

$$\mathbf{Pr}[G(\boldsymbol{a}) = 1 - b] \;\; \leq \;\; 2^{-ct},$$

for $c = \log(2/\sqrt{3})$. Taking the union over all the 2^n possibilities for $\boldsymbol{a}$, we have

$$\begin{aligned} \mathbf{Pr}[\exists \boldsymbol{a} : \; G(\boldsymbol{a}) = 1 - b] &\leq 2^{-ct+n} \\ &\leq 2^{-m}, \end{aligned}$$

for $t \geq (n+m)/c$. Now the claim follows since $(n+m)/c \leq 2m/c < 9.7m$. □

The only point that remains to explain is how to do the uniform sampling from $\textsc{Cons}(\mathcal{C})$. This turns out to be a nontrivial problem. In fact, we are not able to do so just with an **NP**-oracle. One reason for this is that we don't have the sample space, $\textsc{Cons}(\mathcal{C})$, in hand. it is just implicitly given (by the definition). Another, more technical reason is that because we just allow coin tosses as our basic probability experiment, it is impossible to sample uniformly, for example, a set of three elements.

However, if one allows a generating algorithm to make sometimes *no* output (with small probability), then Bellare, Goldreich, and Petrank [BGP98] showed how to get a uniform generator with an **NP**-oracle. They build on and improve a result by Jerrum, Valiant, and Vazirani [JVV86], who showed how to sample *almost uniformly*. We will stick to the latter result because it suffices for our purpose and is somewhat easier to derive. In more detail, we will see that there is a probabilistic polynomial-time Turing machine ALMOST-UNIFORM-SAMPLING that, with the help of an **NP**-oracle, does the following:

ALMOST-UNIFORM-SAMPLING$(F, \mathcal{C}, \epsilon)$

F is a formula, $\mathcal{C}$ is a collection of counterexamples and ϵ a tolerance parameter, where $1/\epsilon$ is bounded by some polynomial in $m = |F|$.
The output probability $\mathbf{Pr}[H]$ for a formula H of size at most m is as follows:

- if $H \notin \textsc{Cons}(\mathcal{C})$ then $\mathbf{Pr}[H] = 0$,
- if $H \in \textsc{Cons}(\mathcal{C})$ then

$$\frac{1}{(1+\epsilon)\|\textsc{Cons}(\mathcal{C})\|} \leq \mathbf{Pr}[H] \leq \frac{1+\epsilon}{\|\textsc{Cons}(\mathcal{C})\|}.$$

The running time is polynomial in m, $|\mathcal{C}|$, and $1/\epsilon$.

In other words, ALMOST-UNIFORM-SAMPLING samples within a factor $(1+\epsilon)$ uniformly from the formulas consistent with the counterexamples. Note that there might be *no* output on some computations of ALMOST-UNIFORM-SAMPLING. However, if $\|\mathrm{CONS}(\mathcal{C})\| > 0$ we get an output with probability at least $\frac{1}{1+\epsilon}$. Therefore the probability that there is no output is bounded by $1 - \frac{1}{1+\epsilon} \leq \epsilon$.

Before we describe the algorithm in detail in Section 3.1.2, we finish the description of our randomized normal form. First we reformulate Lemma 3.3 for the case that we sample only almost uniformly. The point in Lemma 3.3 was that when a counterexample eliminates some fraction of the formulas in CONS($\mathcal{C}$), then this fraction is *precisely* the probability of selecting such a function from CONS($\mathcal{C}$) under uniform distribution. Now, under almost uniform distribution, we might oversample this bad part of CONS($\mathcal{C}$). However, only by a factor $(1 + \epsilon)$. So all we do is to increase slightly the size t of our sample in order to incorporate the $(1+\epsilon)$-deviation from the uniform distribution.

Lemma 3.4 (Sampling Almost Uniformly) *Let $H_1, \ldots, H_t$ be chosen independently and almost uniformly at random from* CONS($\mathcal{C}$) *with tolerance parameter $\epsilon \leq 1/8$ and let $G = \mathrm{majority}(H_1, \ldots, H_t)$, where $t \geq 14m$. Then*

$$\mathbf{Pr}[G \textit{ is good}] \quad \geq \quad 1 - 2^{-m}.$$

Proof Let $(\boldsymbol{a}, b)$ be a counterexample that eliminates less than 1/4 of the formulas in CONS($\mathcal{C}$), so that less than 1/4 of the formulas take value $1 - b$ on $\boldsymbol{a}$.

When sampling almost uniformly from CONS($\mathcal{C}$), we have

$$\mathbf{Pr}[H] \leq \frac{(1+\epsilon)}{\|\mathrm{CONS}(\mathcal{C})\|}.$$

Observe that for $\epsilon = 0$ we get the probability under uniform distribution, $1/\|\mathrm{CONS}(\mathcal{C})\|$. Therefore, when choosing formulas under almost uniform distribution, we can oversample at most by a factor $(1 + \epsilon)$. Hence, we have $\mathbf{Pr}[H(\boldsymbol{a}) = 1 - b] \leq (1 + \epsilon)/4$. If $\epsilon \leq 1/8$ then $(1 + \epsilon)/4 < 1/2$ and hence, we can apply Lemma 2.2 and get

$$\mathbf{Pr}[G(\boldsymbol{a}) = 1 - b] \quad \leq \quad 2^{-ct},$$

for some constant $c \leq 4 - \log 3\sqrt{23}$. Thus

$$\begin{aligned}\mathbf{Pr}[\exists \boldsymbol{a} :\ G(\boldsymbol{a}) = 1 - b] &\le 2^{-ct+n}\\ &\le 2^{-m},\end{aligned}$$

for $t \ge (n+m)/c$. Since $(n+m)/c \le 2m/c < 13.1m$, this proves the lemma. □

Now we have the tools to implement the randomized normal form generator: it follows the basic idea of doing a majority vote but uses an almost uniform sampling strategy to approximate the majority vote of all formulas on a small sample. So let F be some formula of length $|F| = m$. As tolerance parameter we choose $\epsilon = 1/m^3$. Hence we have $\epsilon \le 1/8$ for $m \ge 2$. The algorithm is shown in Figure 3.2.

```
RANDOMIZED-NORMAL-FORM ($F$)
1   $\mathcal{C} \leftarrow \emptyset$
2   repeat at most $\lceil 1/\log(4/3)m \rceil$ times
3       invoke $t = 14m$ times ALMOST-UNIFORM-SAMPLING($F, \mathcal{C}, 1/m^3$)
        to obtain $H_1, \dots, H_t \in$ CONS($\mathcal{C}$)
4       $G \leftarrow$ majority($H_1, \dots, H_t$)
5       if $F \equiv G$ then return $G$
6       else
7           $\boldsymbol{a} \leftarrow$ COUNTEREXAMPLE($F, G$)
8           $\mathcal{C} \leftarrow \mathcal{C} \cup (\boldsymbol{a}, F(\boldsymbol{a}))$
```

Fig. 3.2: RANDOMIZED-NORMAL-FORM computes a normal form of formula F.

Note first that the only outputs of RANDOMIZED-NORMAL-FORM are formulas G equivalent to F. However, the output must *not* be a formula from CONS($\mathcal{C}$) and, in particular, can be slightly larger than F.

We bound the probability that there is no output. By Lemma 3.4, each iteration of the repeat-loop starting in line 2 eliminates less than 1/4 of the elements of CONS($\mathcal{C}$) with probability at most 2^{-m}. Additionally, ALMOST-UNIFORM-SAMPLING can fail to produce a formula from CONS($\mathcal{C}$) with probability at most $\epsilon = 1/m^3$. Therefore, the probability that RANDOMIZED-NORMAL-FORM terminates without finding a formula G equivalent to F is bounded by $O(m^2)/m^3 = O(1/m)$ which is less than 1/4 for large enough m. This proves Theorem 3.1 except for the almost uniform sampling method.

3.1.2 An Almost Uniform Generator

In this section we show how the algorithm ALMOST-UNIFORM-SAMPLING works that we used in the preceding section. That is, how to generate strings from a set in an *approximately* uniform way.

Definition 3.5 *Let $a, b, r \in \mathbf{R}$, $r > 0$. We say that b* approximates a within ratio r, *if*

$$\frac{b}{r} \leq a \leq br.$$

For functions $f, g : \Sigma^* \mapsto \mathbf{N}$ and $r : \mathbf{N} \mapsto \mathbf{R}^+$, we say that g approximates f within ratio r, *if $g(x)$ approximates $f(x)$ within ratio $r(|x|)$ for all x.*

We construct a generator for an arbitrary **NP**-set. Every **NP**-set L can be expressed as

$$x \in L \iff \exists y \in \Sigma^{q(|x|)} : (x, y) \in A,$$

where q is some polynomial and $A \in \mathbf{P}$. By $\#A(x)$ we denote the number of y's such that $(x, y) \in A$.

Definition 3.6 *A probabilistic Turing machine M is called an* almost uniform generator for A with tolerance ϵ, *if, for every $x \in \Sigma^*$ and $y \in \Sigma^{q(|x|)}$ the following holds:*

- *if $(x, y) \notin A$ then $\mathbf{Pr}[M(x) = y] = 0$,*
- *if $(x, y) \in A$ then $\mathbf{Pr}[M(x) = y]$ approximates the uniform distribution, $1/\#A(x)$, within ratio $1 + \epsilon$. That is,*

$$\frac{\mathbf{Pr}[M(x) = y]}{1 + \epsilon} \leq \frac{1}{\#A(x)} \leq \mathbf{Pr}[M(x) = y](1 + \epsilon).$$

Jerrum, Valiant, and Vazirani [JVV86] showed that there is an almost uniform generator that works in polynomial time in $|x|$ and $1/\epsilon$ with the help of an **NP**-oracle, for every set $A \in \mathbf{P}$.

Note that our algorithm ALMOST-UNIFORM-SAMPLING in Section 3.1.1 follows from the existence of such a generator: define the set A on formulas F, counterexamples $\mathcal{C}$, and formulas H such that

$$((F, \mathcal{C}), H) \in A \iff H \in \text{CONS}(\mathcal{C}).$$

Then we have $A \in \mathbf{P}$ and an almost uniform generator will do the desired sampling.

Like Almost-Uniform-Sampling, an almost uniform generator can have *no* output on some computations as well. By the same argument again, this happens with probability at most ϵ.

A first, naive idea for the generator could be to simply guess a string y uniformly at random and output it, if $(x, y) \in A$. However, if there are just a few y's, maybe just one, such that $(x, y) \in A$, then the output probability of the naive generator will be exponentially small. But then condition (ii) in Definition 3.6 can only be fulfilled for large ϵ whereas we need ϵ to be small.

Let x be fixed now, and let $m = q(|x|)$ be the length of the witnesses y for x. Let us consider the naive generator in more detail. To guess a string y of length m uniformly at random a prefix w of y is extended by one bit at a time: with probability $1/2$ to $w0$ and with probability $1/2$ to $w1$. Hence we can view this process as walking down a complete binary tree of depth m. The nodes in depth k correspond to the prefixes w of length k which all have the same probability to be reached, 2^{-k}. The problem with our naive generator arises from the fact that it proceeds with equal probability to parts of the tree where maybe no output is made, as well as to parts where many outputs are made. We could avoid that problem if we could get some information about *how many witnesses there are in a subtree*, because then we could give preference to the subtree that has more witnesses and, in particular, completely ignore subtrees with no witnesses. The information we need is given by the following function.

$$\#A(x, w) \quad = \quad \|\{\, (x, ww') \mid |ww'| = q(|x|) \text{ and } (x, ww') \in A \,\}\|$$

$\#A \in \#\mathbf{P}$, since we count the number of witnesses of an $\mathbf{NP}$-set, namely of the *prefix version* of L. For simplicity we first show how to get an almost uniform generator when we have $\#A$ as an oracle available. Then we will argue that an $\mathbf{NP}$-oracle can be used as well.

The basic generator shown in Figure 3.3 incorporates the idea described above. The inputs are x and a prefix w of a witness for x, $|w| \leq q(|x|) = m$. The initial call is with (x, λ).

Proposition 3.7 *On input (x, w),* Basic-Generator *produces every witness $y = ww'$ uniformly with probability precisely $1/\#A(x, w)$, if there is a witness.*

Proof The proof is by induction on the length of w (downwards). The claim is trivial for $|w| = m$. Suppose $|w| < m$ and let $y = w0w'$ be a witness. To output y, the algorithm has to choose 0 as the next bit which is done with probability p, and then guess w' which is done with probability $1/\#A(x, w0)$ by the induction hypothesis. Therefore the output

```
Basic-Generator $(x, w)$
1  **if** $\#A(x, w) > 0$ **then**
2      **if** $|w| = m$ **then output** $w$
3      **else**
4          $p \leftarrow \#A(x, w0)/\#A(x, w)$
5          **either** with probability $p$ **do** Basic-Generator $(x, w0)$
6          **or** with probability $1 - p$ **do** Basic-Generator $(x, w1)$
```

Fig. 3.3: Basic-Generator computes witnesses under uniform distribution.

probability of y is $p/\#A(x, w0) = 1/\#A(x, w)$ by the definition of p. The case of $y = w1w'$ is analogous. □

However, there is a problem with Basic-Generator when we try to implement it on a probabilistic Turing machine: there we cannot branch according to an arbitrary rational probability p as it is done in line 4 and 5 in Basic-Generator. For example, if we take the standard Turing machine model, then every probabilistic branch point has precisely two successors. But then it is impossible to branch for example with probability $p = 1/3$.

For what values of p is it possible to branch? Consider a complete binary tree, say of depth d. There are 2^d leaves. Label each leaf with the 0-1-path that leads to it from the root. That is, interpreted as a binary number, the kth node from the left has label k, for $k = 0, \ldots, 2^d - 1$. Therefore we can use any rational number of the form $k/2^d$ for some k and d as a branching probability: just divide the tree into the leaves with value smaller or equal, respectively larger than k.

The solution for our problem is simply to *approximate* the value p that we get in line 3 in Basic-Generator by a value of the form $k/2^d$. The parameter d subdivides the unit interval $[0, 1]$ into 2^d subintervals, each of length $1/2^d$. Then we choose k such that

$$\frac{k}{2^d} \le p < \frac{k+1}{2^d}$$

and take $\alpha(p, d) = k/2^d$ as our approximation for p. Clearly, $\alpha(p, d)$ is off from p by less than the interval size, $1/2^d$.

Lemma 3.8 *Let $2^{-d} \le p \le 1$. Then $\alpha(p, 2d)$ approximates p within ratio $1 + 2^{-d}$.*

Proof Let $\alpha(p, 2d) = k/2^{2d}$. Since $p \geq 2^{-d}$, we have that $k \geq 2^d$. Therefore

$$\frac{k+1}{2^{2d}} = \frac{k}{2^{2d}}\frac{k+1}{k} \leq \frac{k}{2^{2d}}(1 + 2^{-d}).$$

□

Since we can choose d in Lemma 3.8 to be a polynomial in m, we get an approximation ratio for the rational probabilities exponentially close to 1. Now consider Basic-Generator when modified to branch with approximations for the probabilities p and $1 - p$ in line 4 and 5: even if the error accumulates after some iterations, it will still stay exponentially close to 1. Therefore this solves our problem as long as we can use a #**P**-oracle.

Proposition 3.9 *Let $A \in \mathbf{P}$ as above. There is a probabilistic polynomial-time Turing machine equipped with a #**P**-oracle that is an almost uniform generator with exponentially small tolerance parameter.*

To get rid of the #**P**-oracle, we show that #**P**-functions can be approximated with high probability in polynomial time with the help of an **NP**-oracle. The result is essentially based on a technique from Sipser [Sip83] and the application to this setting from Stockmeyer [Sto85].

There is a probabilistic polynomial-time Turing machine Approximate-Count that, with the help of an **NP**-oracle, does the following:

Approximate-Count(x, w, ϵ, e)

$x, w \in \Sigma^*$, $|w| \leq m = q(|x|)$, $\epsilon \geq 0$, and $e \geq 1$.
For the output N we have

$$\mathbf{Pr}[N \text{ approximates } \#A(x, w) \text{ within ratio } 1 + \epsilon] \quad \geq \quad 1 - 2^{-e}.$$

The running time is polynomial in $|x|$, $1/\epsilon$, and e.

We describe Approximate-Count in the next section. Let us first see how to apply it for almost uniform generation. For this we modify Basic-Generator by approximating the function $\#A$ with Approximate-Count within ratio $1+\delta$ for some $\delta \leq 1$ to be determined later. Rational probabilities are approximated via function α as explained above. So let $x \in \Sigma^*$ and w be some prefix of a witness y for x, i.e., such that $(x, y) \in A$, and let ϵ be a tolerance parameter. The algorithm is shown in Figure 3.4. The initial call is with input (x, λ, ϵ).

Theorem 3.10 Almost-Uniform-Generator *is an almost uniform generator with tolerance ϵ, for $\epsilon < 1$ and $1/\epsilon$ bounded by some polynomial.*

```
Almost-Uniform-Generator $(x, w, \epsilon)$
if $|w| = m$ and $(x, w) \in A$ then output $w$
else
    $N_0 \leftarrow$ Approximate-Count$(x, w0, \delta, 2m)$
    $N_1 \leftarrow$ Approximate-Count$(x, w1, \delta, 2m)$
    if $N_0 + N_1 > 0$ then
        $p \leftarrow N_0/(N_0 + N_1)$
        $\alpha_0 \leftarrow \alpha(p, 2(m+2))$; $\alpha_1 \leftarrow \alpha(1-p, 2(m+2))$
        either with probability $\alpha_0$ do
            Almost-Uniform-Generator $(x, w0, \epsilon)$
        or with probability $\alpha_1$ do
            Almost-Uniform-Generator $(x, w1, \epsilon)$
```

Fig. 3.4: Almost-Uniform-Generator computes witnesses under almost uniform distribution.

Proof Algorithm Almost-Uniform-Generator starts by invoking Approximate-Count to get approximations for $\#A(x, w0)$ and $\#A(x, w1)$. With probability at most 2^{-2m}, Approximate-Count fails to do so. Since it is called at most $2m$ times, there is a $2m2^{-2m}$ fraction of the outputs of Almost-Uniform-Generator where we have no control on. Since our tolerance is $1/\mathbf{poly}$, we can safely ignore that part.

Next let us consider the part where Approximate-Count provides the desired approximations, i.e., N_0 and N_1 approximate $\#A(x, w0)$ and $\#A(x, w1)$ respectively within ratio $1 + \delta$. Then $N_0 + N_1$ approximates $\#A(x, w)$ within ratio $1 + \delta$, and thus p_0 approximates $\#A(x, w0)/\#A(x, w)$ within ratio $(1 + \delta)^2$.

Since $\#A(x, w0)/\#A(x, w) \geq 2^{-m}$, we have $p_0 \geq 2^{-m-2}$ for $\delta \leq 1$, and hence, by Lemma 3.8, α_0 approximates p_0 within ratio $1 + 2^{-m-2} \leq 1 + \delta$, for large enough m. Therefore α_0 approximates $\#A(x, w0)/\#A(x, w)$ within ratio $(1 + \delta)^3$. The same holds for α_1 and $\#A(x, w1)/\#A(x, w)$.

Since $\alpha_0 + \alpha_1$ can be less than 1, there can be a certain fraction of computations where Almost-Uniform-Generator makes *no* output. However, under the condition that there is an output, the output probability $\mathbf{Pr}[y \mid \text{there is an output}]$ of a witness y approximates the uniform distribution, $1/\#A(x)$, within ratio $(1+\delta)^{3m}$. This follows from the proof of Proposition 3.7, just incorporate the approximation ratio in each iteration of Almost-Uniform-Generator.

To bound the probability that Almost-Uniform-Generator makes no output, we first consider one iteration:

$$\begin{aligned}
1-(\alpha_0+\alpha_1) &\leq 1-\left(\frac{\#A(x,w0)}{\#A(x,w)(1+\delta)^3}+\frac{\#A(x,w1)}{\#A(x,w)(1+\delta)^3}\right)\\
&= 1-\frac{1}{(1+\delta)^3}\\
&\leq \delta\,\frac{3+3\delta+\delta^2}{(1+\delta)^3}\\
&\leq 3\delta.
\end{aligned}$$

Therefore, the probability of getting no output is bounded by $3m\delta$.

Since

$$\mathbf{Pr}[y] \;=\; \mathbf{Pr}[y \mid \text{there is an output}]\cdot\mathbf{Pr}[\text{there is an output}],$$

$\mathbf{Pr}[y]$ approximates the uniform distribution within ratio $(1+\delta)^{3m}/(1-3m\delta)$. We want the ratio to be at most $1+\epsilon$. Define $\delta=\epsilon/(3m^2)$. Then we have

$$\begin{aligned}
\frac{(1+\delta)^{3m}}{1-3m\delta} &= \frac{\left((1+\delta)^{\frac{1}{\delta}}\right)^{\frac{\epsilon}{m}}}{1-\frac{\epsilon}{m}}\\
&\leq \frac{e^{\frac{\epsilon}{m}}}{1-\frac{\epsilon}{m}}\\
&\leq 1+\epsilon.
\end{aligned}$$

To see the last inequality, note that it is equivalent with

$$e \;\leq\; (1+\epsilon)^{\frac{m}{\epsilon}}\left(1-\frac{\epsilon}{m}\right)^{\frac{m}{\epsilon}}.$$

But this holds, since $\left(1-\frac{\epsilon}{m}\right)^{\frac{m}{\epsilon}}\geq\frac{1}{e}$ and $(1+\epsilon)^{\frac{m}{\epsilon}}$ gets larger than any constant for large enough m. □

3.1.3 Approximate Counting

An **NP**-problem asks for the *existence* of a solution whereas a #**P**-problem asks to *count* the number of solutions. Clearly, counting cannot be easier than deciding the existence. But is it harder? Toda [Tod91] showed that #**P** is surprisingly powerful: it can handle the whole polynomial hierarchy, $\mathbf{PH}\subseteq\mathbf{P}^{\#\mathbf{P}}$. Hence we cannot compute #**P** within the polynomial hierarchy, unless the hierarchy collapses. (This is because #**P** has complete functions and therefore, if #**P** can be computed within the polynomial hierarchy, this could be done within some finite level. But then the hierarchy would collapse precisely to that level by the result of Toda.)

However, one can *approximate* #**P**-functions within the polynomial hierarchy. The crucial step was done by Sipser [Sip83], who showed how to apply *universal hashing* to get a rough estimate on the value of a #**P**-functions. This estimate was good enough to distinguish whether the value is large or small, like in the case when one counts the number of accepting computations of a **BPP**-algorithm. It followed that $\mathbf{BPP} \subseteq \Sigma_2\mathbf{P}$ (see Theorem 3.32). Then Stockmeyer [Sto85] fine tuned the technique and showed that any #**P**-function can be approximated within ratio 1/**poly**.

Let $f \in \#\mathbf{P}$. We can write f as

$$f(x) \;=\; \|\{\, y \in \Sigma^{q(|x|)} \mid (x,y) \in A \,\}\|,$$

where q is some polynomial and $A \in \mathbf{P}$. In other words, for $m = q(|x|)$, let

$$Y \;=\; Y_x \;=\; \{\, y \in \Sigma^m \mid (x,y) \in A \,\}$$

the set of witnesses of x. Our goal is to approximate the size of $Y \subseteq \Sigma^m$. We start by describing the technique of Sipser.

Hashing is a technique to store and find elements quickly. The main ingredients are *hash functions* that map *keys* (elements) from a *universe* U (a finite set) to some *hash table* T (a finite set). Usually T is much smaller than U.

Suppose we want to store a set $S \subseteq U$. This is done by storing element $x \in S$ at $h(x) \in T$. A hash function h should have several properties to be useful: first of all, h should be very easy to compute. Second, h should avoid *collisions*, that is, to map two elements of S to the same place in T. In other words, h should be injective on S. However, if we fix a hash function h and vary the set S, then h will be highly non-injective for many S (recall that U is much larger than T).

A way to improve this is to use a *collection* of hash functions $\mathcal{H}$ instead of just one hash function and choose a function from $\mathcal{H}$ at random each time a set should be hashed. The idea thereby is to become independent of a particularly bad hash function for a set S.[1]

Suppose we map elements from U to T at random. The probability that two elements $y \neq z$ are mapped to the same place in T is $1/\|T\|$. Hence, we expect to end up with mapping $\|U\|/\|T\|$ elements of U to one element of T. Carter and Wegman [CW79] called a family of functions *universal* if it behaves like a random mapping.

[1]Clearly, one cannot completely avoid collisions. Therefore, techniques how to handle collisions are an important topic in practice.

Definition 3.11 *A collection of hash functions $\mathcal{H}$ from U to T is called* universal, *if for all $y \neq z \in U$*

$$\mathbf{Pr}[h(y) = h(z)] \quad = \quad \frac{1}{\|T\|},$$

where h is chosen uniformly at random from $\mathcal{H}$. In other words, for every pair $y \neq z$, the same fraction $\|\mathcal{H}\|/\|T\|$ of functions from $\mathcal{H}$ lead to a collision in T.

We give an example for a class of universal hash functions. Let $U = \Sigma^m$ and $T = \Sigma^t$. We consider Σ^m and Σ^t as vector spaces over GF(2), the two element field. As hash functions we take the set of linear functions from Σ^m to Σ^t, that is, $\mathcal{H}$ is the set of $t \times m$ 0-1-matrices, $\mathcal{M}(t, m)$.

Lemma 3.12 *$\mathcal{M}(t, m)$ is a universal class of hash functions.*

Proof Let $\boldsymbol{y} \neq \boldsymbol{z} \in \Sigma^m$. We show that $\mathbf{Pr}[H\boldsymbol{y} = H\boldsymbol{z}] = 2^{-t}$ for a randomly chosen $H \in \mathcal{M}(t, m)$.

Since $\mathcal{M}(t, m)$ consists of linear functions and since GF(2) is the underlying field,

$$\begin{aligned} H\boldsymbol{y} = H\boldsymbol{z} \quad &\Longleftrightarrow \quad H\boldsymbol{y} + H\boldsymbol{z} = \mathbf{0} \\ &\Longleftrightarrow \quad H(\boldsymbol{y} + \boldsymbol{z}) = \mathbf{0}. \end{aligned}$$

Note that $\boldsymbol{y} + \boldsymbol{z} \neq \mathbf{0}$ since $\boldsymbol{y} \neq \boldsymbol{z}$. Therefore it suffices to show that

$$\mathbf{Pr}[H\boldsymbol{y} = \mathbf{0}] \quad = \quad 2^{-t} \tag{3.1}$$

for $\boldsymbol{y} \neq \mathbf{0}$ and a randomly chosen $H \in \mathcal{M}(t, m)$.

The product $H\boldsymbol{y}$ is a vector in Σ^t obtained by multiplying the t rows of H with $\boldsymbol{y}$, i.e., by building t inner products in Σ^m. Since each 0-1-entry of H is chosen independently at random, the rows of H are independent too. Therefore, in order to prove equation (3.1), it is enough to show that each inner product is 0 with probability precisely 1/2.

Let $\boldsymbol{y} = y_1 \cdots y_m \in \Sigma^m$ such that $\boldsymbol{y} \neq \mathbf{0}$. For $b = 0, 1$, consider the vectors that have inner product b with $\boldsymbol{y}$ (over GF(2)):

$$I_b \quad = \quad \{\, \boldsymbol{a} \in \Sigma^m \mid \boldsymbol{a} \cdot \boldsymbol{y} = b \,\}$$

Fix a position i such that $y_i = 1$. Consider any $\boldsymbol{a} \in I_0$. Define $\boldsymbol{a}'$ to coincide with $\boldsymbol{a}$ except for the ith bit which is flipped,

$$\boldsymbol{a}' \quad = \quad a_1 \cdots a_{i-1}(1 - a_i)a_{i+1} \cdots a_m.$$

Then $\boldsymbol{a}' \in I_1$.

It follows that the function that maps $\boldsymbol{a}$ to $\boldsymbol{a}'$ as explained above is a bijection between I_0 and I_1. Hence $\|I_0\| = \|I_1\| = 2^{m-1}$. So if we choose $\boldsymbol{a} \in \Sigma^m$ uniformly at random we get

$$\mathbf{Pr}[\boldsymbol{a} \cdot \boldsymbol{y} = 0] \quad = \quad 1/2,$$

which is what we wanted to show. □

Now consider a subset $Y \subseteq \Sigma^m$ that we want to hash with $H \in \mathcal{M}(t,m)$. H might not be injective on whole Y, but maybe on a part of Y.

Definition 3.13 *Let $Y \subseteq \Sigma^m$ and $\boldsymbol{y} \in Y$. $H \in \mathcal{M}(t,m)$* separates $\boldsymbol{y}$ within Y*, if $H\boldsymbol{z} \neq H\boldsymbol{y}$ for all $\boldsymbol{z} \in Y \setminus \{\boldsymbol{y}\}$.*

Next consider a (small) collection $\mathcal{H} \subseteq \mathcal{M}(t,m)$ of hash functions. No single $H \in \mathcal{H}$ might be injective on Y. We consider the case that each $\boldsymbol{y} \in Y$ can at least be separated within Y by *some* $H \in \mathcal{H}$.

Definition 3.14 *A collection $\mathcal{H} \subseteq \mathcal{M}(t,m)$* separates Y*, if for each $\boldsymbol{y} \in Y$ there is a $H \in \mathcal{H}$ that separates $\boldsymbol{y}$ within Y. This is expressed by predicate* $\textsc{Separate}_Y$*:*

$$\textsc{Separate}_Y(\mathcal{H}) \quad \Longleftrightarrow \quad \forall \boldsymbol{y} \in Y\ \exists H \in \mathcal{H}\ \forall \boldsymbol{z} \in Y \setminus \{\boldsymbol{y}\} :\ H\boldsymbol{y} \neq H\boldsymbol{z}.$$

When we take $Y = Y_x$ as a set of witnesses for some x as explained above and if furthermore $\mathcal{H}$ contains only polynomially many functions, then $\textsc{Separate}_Y(\mathcal{H})$ can be decided in **coNP**. This is because for small $\mathcal{H}$ the existential quantifier can be pulled into the formula by standard techniques.

If the set Y is small, a randomly chosen collection of hash function is quite likely to separate every $\boldsymbol{y} \in Y$. But certainly this probability decreases when Y gets larger. On the other hand, we increase our chances to separate Y when we take more functions into our collection $\mathcal{H}$. The precise relationship is given by Sipser's Coding Lemma.

Lemma 3.15 (Coding Lemma) *Let $Y \subseteq \Sigma^m$. For a collection $\mathcal{H}$ of k hash functions chosen uniformly at random from $\mathcal{M}(t,m)$*

$$\mathbf{Pr}[\textsc{Separate}_Y(\mathcal{H})] \quad \geq \quad 1 - \frac{\|Y\|^{k+1}}{2^{kt}}.$$

Proof We bound the probability to *not* separate, i.e., to have a collision. From Lemma 3.12 we know that for any fixed $\boldsymbol{y} \neq \boldsymbol{z}$, we have $\mathbf{Pr}[H\boldsymbol{y} = H\boldsymbol{z}] = 2^{-t}$ for a randomly chosen H. Therefore, for fixed $\boldsymbol{y}$,

$$\mathbf{Pr}[\exists \boldsymbol{z} \in Y \setminus \{\boldsymbol{y}\} : \ H\boldsymbol{y} = H\boldsymbol{z}] \ \leq \ \|Y\| 2^{-t}.$$

The probability that we have a collision for every $H \in \mathcal{H}$ is

$$\mathbf{Pr}[\forall H \in \mathcal{H} \ \exists \boldsymbol{z} \in Y \setminus \{\boldsymbol{y}\} : \ H\boldsymbol{y} = H\boldsymbol{z}] \ \leq \ \left(\|Y\| 2^{-t}\right)^k.$$

Finally, we bound the probability that there is some $\boldsymbol{y} \in Y$ that is not separated:

$$\mathbf{Pr}[\exists \boldsymbol{y} \ \forall H \in \mathcal{H} \ \exists \boldsymbol{z} \in Y \setminus \{\boldsymbol{y}\} : \ H\boldsymbol{y} = H\boldsymbol{z}] \ \leq \ \|Y\|^{k+1} 2^{-kt}.$$

□

Suppose Y has at most half the size of Σ^t, i.e., $\|Y\| \leq 2^{t-1}$. Then a collection of t hash functions is quite likely to separate Y.

Corollary 3.16 *If $\|Y\| \leq 2^{t-1}$ then for a collection $\mathcal{H}$ of t hash functions chosen uniformly at random from $\mathcal{M}(t, m)$*

$$\mathbf{Pr}[\textsc{Separate}_Y(\mathcal{H})] \ \geq \ 1/2$$

Proof The claim follows from the Coding Lemma since $\|Y\|^{k+1} 2^{-kt} \leq 2^{(t-1)(t+1)-t^2} = 1/2$, □

On the other hand, if Y is slightly bigger than Σ^t, then *no* collection of t hash functions can separate Y.

Lemma 3.17 *If $\|Y\| > t2^t$ then no collection of t hash functions from $\mathcal{M}(t, m)$ separates Y.*

Proof Let $H_1, \ldots, H_t \in \mathcal{M}(t, m)$ be a family of hash functions that separates Y. By definition, for each $\boldsymbol{y} \in Y$ there exists an index i such that $H_i\boldsymbol{y} \neq H_i\boldsymbol{z}$ for all $\boldsymbol{z} \neq \boldsymbol{y}$. Thus the function $\boldsymbol{y} \in Y \mapsto (H_i\boldsymbol{y}, i) \in \Sigma^t \times \{1, \ldots, t\}$ is injective, and hence $\|Y\| \leq t2^t$. □

Corollary 3.18 *Let $\emptyset \neq Y \subseteq \Sigma^m$ and let t_0 be the smallest t such that there exists a collection of t hash functions from $\mathcal{M}(t, m)$ that separates Y, then*

$$2^{t_0 - 2} \ < \ \|Y\| \ \leq \ t_0 2^{t_0}.$$

Proof The upper bound follows from Lemma 3.17. For the lower bound, we use Corollary 3.16 (in the contrapositive): since no collection of $(t_0 - 1)$ hash functions separates Y, we have $\|Y\| > 2^{(t_0-1)-1} = 2^{t_0-2}$. □

Corollary 3.18 gives an approximation of the size of Y within ratio $4t_0$. Since $t_0 \leq \log\|Y\| + 2 \leq 3\log\|Y\|$, the approximation ratio is bounded by $12\log\|Y\| \leq 12m$.

So all we have to do is to search for the smallest t that separates Y. Recall that $\textsc{Separate}_Y$ is a **coNP**-predicate in our application. Therefore, the question whether for a given t

$$\exists H_1, \ldots, H_t \in \mathcal{M}(t,m) \ \ \textsc{Separate}_Y(H_1, \ldots, H_t)$$

is a Σ_p^2-predicate. Hence, we can approximate a #**P**-function in deterministic polynomial time with the help of a Σ_2**P**-oracle, at least within ratio $12\log\|Y\|$.

It is not hard to get a better approximation ratio: instead of approximating $\|Y\|$, we start with approximating the size of the cartesian product Y^r, for some $r \geq 1$. Note that $Y^r \subseteq \Sigma^{mr}$ and $\|Y^r\| = \|Y\|^r$. Hence, we get an approximation for $\|Y^r\|$ within ratio $12mr$, and therefore an approximation for $\|Y\|$ within ratio $(12mr)^{1/r}$.

The approximation ratio should be close to 1. So let $p(m)$ be some polynomial. We want $(12mr)^{1/r}$ to be at most $1 + 1/p$. This is equivalent to

$$12mr \quad \leq \quad \left(1 + \frac{1}{p}\right)^r. \tag{3.2}$$

We choose $r = pm$. Then we have

$$\begin{aligned}\left(1 + \frac{1}{p}\right)^r \quad &= \quad \left(\left(1 + \frac{1}{p}\right)^p\right)^m \\ &\geq \quad 2^m,\end{aligned}$$

for $p \geq 1$. Therefore inequality (3.2) holds for large enough m. We have proven the following theorem:

Theorem 3.19 *Let $f \in$ #**P**. There is a (deterministic) Turing machine M equipped with an Σ_2**P**-oracle such that $M(x,\epsilon)$ approximates $f(x)$ within ratio $1+\epsilon$.*

The running time of M is polynomial in x and $1/\epsilon$.

Recall that we have to do counting with an **NP**-oracle only for our almost uniform generator from the last section. So Theorem 3.19 still does not solve our problem. But we are on the way!

The idea how to get along with an **NP**-oracle is provided by Corollary 3.16: let the approximation algorithm guess the hash functions itself uniformly at random [2] and then check whether they separate Y. That way we might not necessarily find the minimum t, but by Corollary 3.16 we are off by at most 1 with probability at least 1/2. The algorithm is shown in Figure 3.5.

```
APPROXIMATE (x)
1  $t \leftarrow 0$
2  **repeat**
3      $t \leftarrow t + 1$
4      **randomly choose** $H_1, \dots, H_t \in \mathcal{M}(t, m)$
5  **until** SEPARATE$_Y(H_1, \dots, H_t)$ **or** $t = m$
6  **return** $t2^t$
```

Fig. 3.5: APPROXIMATE approximates the size of set Y_x.

We could do a binary search as well which would be more efficient. The above way is slightly easier to analyze. By Corollary 3.18, we can take any number in the interval $[2^{t-2} + 1, t2^t]$ as an approximation value in line 6. We take the largest value so that we never underestimate the correct value.

Theorem 3.20 *Let $Y \subseteq \Sigma^m$ be the set of witnesses of x.* APPROXIMATE(x) *approximates $\|Y\|$ within ratio $12 \log \|Y\|$ with probability at least* 1/2*.*

Proof Let s be such that $2^{s-1} < \|Y\| \leq 2^s$. When variable t in APPROXIMATE has value $t = s + 1$ in line 4, the assumption of Corollary 3.16 is fulfilled and we have $\mathbf{Pr}[\text{SEPARATE}_Y(H_1, \dots, H_t)] \geq \frac{1}{2}$.

For $t = s + 1$ we also have $2^{t-2} < \|Y\| \leq t2^t$, the approximation ratio is bounded by $12 \log \|Y\|$. □

We can again improve the approximation ratio with the cartesian product trick as outlined above. We have proven the following theorem:

[2] Since our hash functions are 0-1-matrices, this time we get no problem with choosing *uniformly at random*: simply choose each matrix entry with probability 1/2 to be 0 or 1.

Theorem 3.21 *Let $f \in \#\mathbf{P}$. There is a probabilistic Turing machine M equipped with an $\mathbf{NP}$-oracle such that on input (x, ϵ),*

$$\mathbf{Pr}[M(x,\epsilon) \textit{ approximates } f(x) \textit{ within ratio } 1+\epsilon] \quad \geq \quad 1/2.$$

The running time of M is polynomial in x and $1/\epsilon$.

The approximation algorithm provided by Theorem 3.21 can now almost be used for the almost uniform generator in the last section: the only catch is that we should have a higher success probability than $1/2$, namely $2^{-p(n)}$, for some polynomial p. But this is easy to achieve now: simply define machine M_k to iterate machine M in Theorem 3.21 k times and output the minimum of all outputs of M in these iterations. (Recall that the outputs of M never underestimate $f(x)$.) Then $M_k(x, \epsilon)$ approximates $f(x)$ within ratio $1 + \epsilon$ with probability at least $1 - 2^{-k}$.

Corollary 3.22 *Let $f \in \#\mathbf{P}$ and p be some polynomial. There is a probabilistic Turing machine M equipped with an $\mathbf{NP}$-oracle such that on input (x, ϵ),*

$$\mathbf{Pr}[M(x,\epsilon) \textit{ approximates } f(x) \textit{ within ratio } 1+\epsilon] \quad \geq \quad 1 - 2^{-p(|x|)}.$$

The running time of M is polynomial in x and $1/\epsilon$.

Corollary 3.22 completes the proof of Theorem 3.1.

Another way to amplify the success probability follows from the Coding Lemma: increase the number of hash functions chosen in Corollary 3.16. This amplifies the probability of predicate SEPARATE$_Y$ to be true which is precisely what we want. This also increases the approximation ratio of our initial estimate. But this can be handled again by using the cartesian product as we did above.

3.1.4 The Protocol

We have provided all the tools needed to describe the protocol for the formula non-isomorphism problem, FNI. The verifier selects one of the two input formulas at random and randomly permutes its variables. Let F be the resulting formula. Now, instead of directly sending F to the prover, the verifier first uses RANDOMIZED-NORMAL-FORM to obtain, with high probability, a formula G equivalent to F, and then sends G to the prover. The the prover is asked to find out which input formula was used to obtain G. We give the full protocol below.

Theorem 3.23 FNI $\in \mathbf{IP}[2]^{\mathbf{NP}}$.

Proof The IP-protocol of Figure 3.6 accepts FNI. The inputs are two formulas (F_0, F_1), both in variables $x_1, \ldots, x_n$ and w.l.o.g. of the same length.

- [V:] The verifier randomly picks $i \in \{0,1\}$ and a random permutation φ on n variables. Let $G = F_i \circ \varphi$. Now, the verifier computes a randomized normal form H of G by using the algorithm RANDOMIZED-NORMAL-FORM(G) from Section 3.1.1 and sends H to the prover. On those paths where the algorithm does not make an output, the verifier directly accepts.
- [P:] The prover answers by sending $j \in \{0,1\}$ to the verifier.
- [V:] Finally, the verifier accepts if the prover gave the correct answer, i.e., if $i = j$, and rejects otherwise.

Fig. 3.6: Interactive protocol for FNI.

We show that the protocol shown in Figure 3.6 is correct. If F_0 is *not* isomorphic to F_1, a prover can determine which of F_0 and F_1 formula H is isomorphic to, and tell it to the verifier. Also, on the random paths where no equivalent formula is produced the verifier accepts. Therefore, the verifier accepts with probability one.

Now consider the case when F_0 is isomorphic to F_1. Assume the verifier picks $i = 0$ (the case $i = 1$ is analogous). Then the verifier constructs $G = F_0 \circ \varphi$ for some randomly chosen permutation φ. Since F_0 is isomorphic to F_1, G is isomorphic to F_1 too. Hence there is some permutation φ' such that formula $G' = F_1 \circ \varphi'$ is equivalent to G.

The next step of the verifier is to apply the algorithm of Theorem 3.1 to G and to obtain the equivalent formula H (or no formula). Now observe that the random bits of the verifier that lead to the construction of H on input G would also lead to the construction of H on input G'. In other word, any formula H has the same probability to be sent to the prover, irrespective of whether G was obtained from F_0 or F_1. Therefore the answer of any prover will be correct with probability 1/2.

In summary, the verifier accepts with probability 1/2 on those computations where a formula H is produced and on all computations with no output. The latter occurs with probability at most 1/4. Therefore, the verifier will accept with probability at most $1/2 + 1/4 = 3/4$.

The definition of an interactive proof system requires an error bound of at most 1/2. There is a standard trick to achieve this now: execute the above protocol three times in parallel and accept only if all three executions lead to acceptance. This doesn't change the case when F_0 is not isomorphic to F_1. In the other case, the error decreases to $(3/4)^3 < 1/2$. This proves the theorem. □

As explained in Section 2.5, for any interactive proof system with constantly many messages there is an equivalent Arthur-Merlin game with two messages, and this holds as well in our relativized setting, i.e., with an **NP**-oracle.

Corollary 3.24 FNI $\in \mathbf{AM}^{\mathbf{NP}} = \mathbf{BP} \cdot \Sigma_2\mathbf{P}$.

Finally, by applying Theorem 2.5 we get the **main result** of this chapter.

Corollary 3.25 FI *is not* $\Sigma_2\mathbf{P}$*-complete unless* $\mathbf{PH} = \Sigma_3\mathbf{P}$.

3.1.5 Extensions

There are some nice extensions and applications of the techniques seen so far. We present some below.

Circuits

The proof of Theorem 3.1 works as well for circuits. Therefore we can adapt the interactive proof for FNI for the circuit non-isomorphism problem which is therefore in $\mathbf{IP}[2]^{\mathbf{NP}}$.

Corollary 3.26 *The isomorphism problem for boolean circuits is not* $\Sigma_2\mathbf{P}$*-complete unless* $\mathbf{PH} = \Sigma_3\mathbf{P}$.

Affine Equivalence

We can also use the more flexible transformations defined in Section 2.6. There we mentioned congruence and linear and affine equivalence. We directly argue for the most general transformation: affine equivalence. The only difference to the above protocol for FNI is when the verifier randomly generates a permutation. Now, the verifier must randomly generate an affine transformation. To achieve this, the verifier randomly generates an n-bit vector and an $n \times n$ 0-1-matrix. To constitute an affine transformation, the matrix should be nonsingular. If the matrix is singular, the verifier accepts immediately. Otherwise, the protocol proceeds as in the case of a permutation. Our next lemma ensures that there are enough non-singular matrices for the verifier to find one with high probability.

Lemma 3.27 *At least* $1/4$ *of the* $n \times n$ *matrices over* GF(2) *are non-singular.*

Proof We successively choose the column vectors of a $n \times n$ matrix such that the next column vector is linearly independent of the previous ones. The first column can be chosen arbitrary, except that it can't be zero. So there are $2^n - 1$ choices.

Any k linearly independent vectors in $\mathrm{GF}(2)^n$ span a vector space of size 2^k. Therefore, when we choose the $(k+1)$-st column, we have $2^n - 2^k$ choices.

In total, $\prod_{k=0}^{n-1}(2^n - 2^k)$ of the 2^{n^2} $n \times n$ matrices over GF(2) are non-singular. Thus, their proportion is

$$\begin{aligned}
\frac{1}{2^{n^2}} \prod_{k=0}^{n-1} (2^n - 2^k) &= \prod_{k=1}^{n} (1 - \frac{1}{2^k}) \\
&= \frac{1}{2} \prod_{k=2}^{n} (1 - \frac{1}{2^k}) \quad \text{(for } n \geq 2\text{)} \\
&\geq \frac{1}{2} \prod_{k=2}^{n} (1 - \frac{1}{k^2}) \quad \text{(for } n \geq 6\text{)} \\
&= \frac{1}{2} \, (\frac{1}{2} \, \frac{n+1}{n}) \\
&\geq \frac{1}{4},
\end{aligned}$$

where the second line from bottom follows by induction on n. For $n \leq 6$ the claim is true as well. □

Corollary 3.28 *The affine nonequivalence problem for formulas and circuits is in* $\mathbf{IP}[2]^{\mathbf{NP}}$. *Therefore the corresponding affine equivalence problems for formulas and circuits are not* $\Sigma_2\mathbf{P}$*-complete unless* $\mathbf{PH} = \Sigma_3\mathbf{P}$.

Proof Let C_0 and C_1 be the input circuits, both in variables $x_1, \ldots, x_n$, and w.l.o.g. of the same length. For completeness, we give the full protocol in Figure 3.7.

We show that the protocol works correctly. If (C_0, C_1) are *not* affine equivalent the honest prover can *always* convince the verifier.

Now consider the case where $(C_0, C_1) \notin$ CANE. It suffices to show that the probability that a specific circuit E is presented to the prover is independent of whether it was produced from C_0 or from C_1. In this case, the acceptance probability of the verifier is bounded by $1/2$ for the right answer of the prover, plus $(3/4)^5 < 1/16$ for not finding a nonsingular matrix, plus $1/4$ for not getting a normal form. This sums up to $13/16$. By executing the protocol four times in parallel, we can bring the acceptance probability down to $(13/16)^4 < 1/2$.

- [V:] The verifier randomly picks $i \in \{0,1\}$, and an n-bit vector $\boldsymbol{r}$. Furthermore, the verifier makes up to five trials to randomly get a nonsingular $n \times n$ 0-1 matrix.
  If all trials fail, the verifier stops and accepts directly. Otherwise, let $\boldsymbol{R}$ be the (nonsingular) random matrix, and let $D = C_i(\boldsymbol{xR} + \boldsymbol{r})$.
  Next, the verifier produces the randomized normal form from D according to the algorithm of Theorem 3.1 and obtains the equivalent circuit E, and sends E to the prover. If the algorithm fails to produce a normal form, the verifier accepts directly.
- [P:] The prover answers by sending $j \in \{0,1\}$ to the verifier.
- [V:] Finally, the verifier accepts if $i = j$, and rejects otherwise.

Fig. 3.7: Interactive protocol for the affine nonequivalence problem for circuits

It remains to argue that independent of whether i was chosen to be 0 or 1, we have the same chance of getting circuit E. Let $\boldsymbol{xA} + \boldsymbol{c}$ be the affine transformation so that $C_1(\boldsymbol{xA}+\boldsymbol{c})$ is equivalent to C_0. For a random affine transformation, say $\boldsymbol{xR}+\boldsymbol{r}$, applied to C_0, we get $D_0 = C_0(\boldsymbol{xR}+\boldsymbol{r})$. The equivalent circuit via C_1 is $D_1 = C_1(\boldsymbol{xAR}+\boldsymbol{cR}+\boldsymbol{r})$. Now note that $\boldsymbol{x} \mapsto \boldsymbol{xAR} + \boldsymbol{cR} + \boldsymbol{r}$ is still a random affine transformation for fixed $\boldsymbol{A}$ and $\boldsymbol{c}$. Therefore we have the same probability to get D_0 when $i = 0$ and to get D_1 when $i = 1$. Since D_0 and D_1 are equivalent, our randomized normal form algorithm has an identical output distribution on input D_0 and on input D_1. $\square$

Small Circuits for NP

We used the algorithm Randomized-Normal-Form to obtain formulas equivalent to a given input formula. This works as well for circuits. Now we change the setting a little: assume we just know the *existence* of a circuit C_{sat} of a certain size that accepts $\text{Sat}^{\leq m}$, the set of satisfiable formulas up to length m, but the circuit is *not* given as input (only its size). Watanabe (see [BCG+96]) observed that Randomized-Normal-Form can be modified to still output circuits equivalent to C_{sat}, thereby providing us with a circuit that we knew only to exist before! Let us consider this in more detail.

The only places where Randomized-Normal-Form used the input circuit was for equivalence tests or to compute counterexamples for the (majority) circuit, call it C. We have to show that we still can do so in $\mathbf{P}^{\mathbf{NP}}$ in the new setting.

Let $L(C)$ denote the set of inputs accepted by circuit C. The equivalence test now turns into the question whether $L(C) = \textsc{Sat}^{\leq m}$. If not, a counterexample is a formula in $\textsc{Sat}^{\leq m} \triangle L(C)$, the symmetric difference of $\textsc{Sat}^{\leq m}$ and $L(C)$.

Lemma 3.29 *Given a circuit C with m inputs, we can decide whether C accepts* $\textsc{Sat}^{\leq m}$ *or compute a counterexample otherwise in* $\mathbf{P^{NP}}$.

Proof For a given circuit C, we will define a set A_C of formulas F that has the following properties:

- $A_C \subseteq \textsc{Sat}^{\leq m} \triangle L(C)$,
- if $\textsc{Sat}^{\leq m} \neq L(C)$, then $A_C \neq \emptyset$, and
- $A = \{ (C, F) \mid F \in A_C \} \in \mathbf{NP}$.

Note that this suffices to prove the lemma: having such a set A, we can do a prefix-search within $\mathbf{P^{NP}}$ to compute an element of A_C for any given C. If we fail to find one, then $A_C = \emptyset$ and we conclude that $\textsc{Sat}^{\leq m} = L(C)$. Otherwise we get a counterexample.

To define set A_C we consider two cases:

Case 1: $\textsc{Sat}^{\leq m} - L(C) \neq \emptyset$. Then $A_C = \textsc{Sat}^{\leq m} - L(C)$ fulfills all the requirements.

Case 2: $\textsc{Sat}^{\leq m} \subseteq L(C)$. Note that we *cannot* define A_C as $L(C) - \textsc{Sat}^{\leq m}$ because then we only have $\mathbf{coNP}$ as an upper bound on A.

Instead, we exploit the *self-reducibility* of $\textsc{Sat}$. That is, let $F = F(x_1, \ldots, x_n)$, then $F_{x_1=b}$ denotes the sub-formula of F obtained by fixing x_1 to value b, for $b \in \{0,1\}$. We have

$$F \in \textsc{Sat} \quad \Longleftrightarrow \quad F_{x_1=0} \in \textsc{Sat} \text{ or } F_{x_1=1} \in \textsc{Sat}.$$

The idea now is to replace the test $F_{x_1=b} \in \textsc{Sat}$ in this equation by checking whether $C(F_{x_1=b}) = 1$. Clearly this is not the same test any more: just in the case that $C(F_{x_1=b}) = 0$ we can be sure that this is correct, i.e., that $F_{x_1=b} \notin \textsc{Sat}$. Therefore, we define A_C as the set of formulas $F(x_1, \ldots, x_n)$ such that

- $C(F) = 1$ and
- either $n = 0$ (F has no variables) and $F \equiv 0$,
- or $C(F_{x_1=0}) = C(F_{x_1=1}) = 0$.

Then clearly $A_C \subseteq L(C) - \textsc{Sat}^{\leq m}$. Observe also that if $L(C) - \textsc{Sat}^{\leq m} \neq \emptyset$, then $A_C \neq \emptyset$: suppose that C rejects the constant 0 formula (otherwise we are done). Now take a formula $F \in L(C) - \textsc{Sat}^{\leq m}$ and look at the formulas obtained from F via the self-reduction as above. If we do this process iteratively for the new formulas on the remaining variables, we construct the *self-reduction tree* of F. Since F is unsatisfiable, all the formulas constructed are unsatisfiable too, and hence the bottom level of the tree, i.e., when there are no more variables in the resulting formulas, consists only of the constant 0 formula. Hence there must be a formula in the tree which is accepted by C but both of its successors are rejected by C. Therefore $A_C \neq \emptyset$ in case 2 as well.

Finally note that $A \in \mathbf{NP}$: on input (C, F), first check the items of case 2 from the previous paragraph, and accept if they are fulfilled. This is a polynomial-time computation. If not, then accept if $F \in \textsc{Sat}^{\leq m} - L(C)$, and reject otherwise. This is an $\mathbf{NP}$-predicate. □

Now assume that there exists a circuit of size s with m inputs that accepts $\textsc{Sat}^{\leq m}$. We modify our algorithm Randomized-Normal-Form: the input are numbers s and m (instead of a formula). The algorithm does the same as before, but with equivalence tests and the computation of counterexamples replaced by the method of Lemma 3.29. Then the outcome is, with high probability, a randomized normal form of the circuit of size s that we assumed to exist, i.e., a circuit for $\textsc{Sat}^{\leq m}$.

Corollary 3.30 *There is a probabilistic algorithm R that has access to an* $\mathbf{NP}$*-oracle and, on input of $m, s > 0$, has the following property: If there is a circuit of size s for* $\textsc{Sat}^{\leq m}$*, then, with probability at least $3/4$, $R(m, s)$ outputs a circuit that accepts* $\textsc{Sat}^{\leq m}$*, and makes no output otherwise. The running time is polynomial in s and m.*

We don't expect $\mathbf{NP}$-complete problems to be solvable by polynomial-time algorithms. As an extension, we also don't expect that $\mathbf{NP}$-complete problems can be solved by *polynomial-size circuits*. In favor of this conjecture Karp and Lipton [KL82] showed that otherwise the polynomial hierarchy collapses to $\Sigma_2\mathbf{P}$ (see [Sch95] for a simplified proof). Watanabe observed that, with Corollary 3.30 we can improve the Karp-Lipton result to $\mathbf{ZPP}^{\mathbf{NP}}$.

Corollary 3.31 $\mathbf{NP}$ *does not have polynomial-size circuits, unless* $\mathbf{PH} = \mathbf{ZPP}^{\mathbf{NP}}$.

Proof Assume that there is some polynomial p such that for all m, there is a circuit of size $p(m)$ that accepts $\textsc{Sat}^{\leq m}$.

Consider a set $L \in \Sigma_2\mathbf{P} = \mathbf{NP}^{\mathbf{NP}}$. Let $L = L(M^A)$ for some $\mathbf{NP}$-machine M with running time q and $A \in \mathbf{NP}$. Define a set B as follows: for a string x and a circuit C let

$$(x, C) \in B \iff M^C(x) \text{ accepts},$$

where $M^C(x)$ uses circuit C to answer oracle queries. That is, there are no oracle queries any more. Instead, M evaluates C (in polynomial time). Hence $B \in \mathbf{NP}$.

Consider the following algorithm for L on input x: first compute a circuit C_{sat} for $\text{SAT}^{\leq m}$, where $m = q(|x|)$ (by Corollary 3.30). Then accept iff $(x, C_{sat}) \in B$.

We conclude that $L \in \mathbf{ZPP}^{\mathbf{NP}}$ and therefore $\Sigma_2\mathbf{P} = \mathbf{ZPP}^{\mathbf{NP}}$. Since $\mathbf{ZPP}^{\mathbf{NP}}$ is closed under complementation this proves the claim. □

BPP Is in the Polynomial Hierarchy

Sets in **BPP** have polynomial-size circuits because, for small enough error, there must be a choice of the random bits that yields the correct result on *all* inputs of a given length. In a circuit we can hardwire this random string. Then the circuit only needs to evaluate a probabilistic (polynomial-time) Turing machine on a specific random string. This yields a polynomial-size circuit. With respect to Corollary 3.31 we conjecture that **NP** is *not* contained in **BPP**.

Using the approximation technique from Section 3.1.3, we show an upper bound for **BPP**: it is contained in the polynomial hierarchy, namely in $\Sigma_2\mathbf{P}$ [Sip83].

Consider a **BPP**-algorithm M that decides some set L correctly with probability, say 2^{-n} on inputs of length n. Let m be the length of the computation paths of M and let Y be the set of rejecting paths of M on some input x. If x is accepted by M, we have $\|Y\| \leq 2^{m-n}$. On the other hand, if x is rejected by M, we have $\|Y\| \geq 2^m - 2^{m-n}$. Now we can directly refer to Corollary 3.16 and Lemma 3.17: choose $t = m - n + 1$. Then we have

$$\begin{aligned} x \in L &\implies \mathbf{Pr}[\text{SEPARATE}_Y(H_1, \ldots, H_t)] \;\geq\; 1/2 \\ x \notin L &\implies \mathbf{Pr}[\text{SEPARATE}_Y(H_1, \ldots, H_t)] \;=\; 0, \end{aligned}$$

where the second line holds because $t2^t < 2^m - 2^{m-n}$ for large enough n. From this representation we get $L \in \mathbf{RP} \cdot \mathbf{coNP} \subseteq \mathbf{RP}^{\mathbf{NP}} \subseteq \Sigma_2\mathbf{P}$. Since **BPP** is closed under complementation, we get

Theorem 3.32 $\mathbf{BPP} \subseteq \mathbf{ZPP}^{\mathbf{NP}} \subseteq \Sigma_2\mathbf{P} \cap \Pi_2\mathbf{P}$.

Precisely the same argument can be used to complete the proof of Proposition 2.9 on page 20 that $\mathbf{AM} = \mathbf{BP}\cdot\mathbf{NP}$. It remained to show that

$\mathbf{BP}\cdot\mathbf{NP} \subseteq \mathbf{coRP}\cdot\mathbf{NP}$. To see this, observe that $\mathbf{BP}\cdot\mathbf{NP}$ can be amplified just like **BPP**. Hence we have $\mathbf{BP}\cdot\mathbf{NP} \subseteq (\mathbf{coRP}\cdot\mathbf{NP})\cdot\mathbf{NP} = \mathbf{coRP}\cdot\mathbf{NP}$.

Subsequently, Lautemann [Lau83] gave an alternative, maybe more direct proof of Theorem 3.32. In fact, his proof shows that $\mathbf{BPP} \subseteq \mathbf{MA}$. Goldreich and Zuckerman [GZ97] show that $\mathbf{MA} \subseteq \mathbf{ZPP}^{\mathbf{NP}}$.

3.2 Comparing FI with Other Problems

We have seen in the preceding sections that the formula isomorphism problem, FI, is in $\Sigma_2\mathbf{P}$, but is very unlikely to be $\Sigma_2\mathbf{P}$-complete. In this section we consider problems that are, more or less, related to FI and compare their complexity via reductions.

We start by noting that FI is **coNP**-hard [BRS98] . Recall that the tautology problem TAUT is **coNP**-complete. Hence it suffices to many-one reduce TAUT to FI. Observe that a tautology is invariant under permutations of its variables: it always represents the constant 1 function. Let true_n denote a fixed tautology in n variables (e.g., $\bigwedge_{i=1}^{n}(x_i \vee \bar{x}_i)$). Then, for a given formula F with n variables, $F \in \text{TAUT} \iff (\mathsf{true}_n, F) \in \text{FI}$.

Recall that FC denotes the *formula congruence problem*. By the same argument, the above reduction from TAUT to FI is a reduction from TAUT to FC as well.

Proposition 3.33 FI *and* FC *are* **coNP**-*hard.*

We will improve Proposition 3.33 in the next sections (Corollary 3.41 and Theorem 3.50).

FI is *not* known to be **NP**-hard. However, Chang (see [BR93]) observed that the graph isomorphism problem, GI, (which is in **NP**) is reducible to FI.

Proposition 3.34 GI $\leq_m^p$ FI.

Proof Let $G = (V, E)$ be a graph where $V = \{1, \dots, n\}$. We define a formula $F_G = F_G(x_1, \dots, x_n)$ as follows.

$$F_G \quad = \quad \bigvee_{(i,j)\in E} (x_i \wedge x_j).$$

Now consider two graphs G_0 and G_1. It is easy to check that $(G_0, G_1) \in \text{GI} \iff (F_{G_0}, F_{G_1}) \in \text{FI}$. □

The formulas that appear in the above reduction from GI to FI have a special form: they are in disjunctive normal form and monotone, i.e., there is no negation. In fact, the isomorphism problem for monotone DNF-formulas is many-one equivalent to GI [Bor97].

3.2.1 USat and FC

Unique Satisfiability (USat) [BG82] is the set of all boolean formulas that have exactly one satisfying assignment. USat is **coNP**-hard. This is because one can reduce the set of *unsatisfiable* formulas to it, which is clearly **coNP**-complete. The trick is to *add* precisely one satisfying assignment to a formula. This is achieved as follows. Let $F = F(x_1, \ldots, x_n)$ be a formula. Now take a new variable z and consider

$$G = G(x_1, \ldots, x_n, z) \quad = \quad (z \wedge F) \;\vee\; (\overline{z} \wedge \bigwedge_{i=1}^{n} x_i).$$

Then F is unsatisfiable iff $G \in$ USat. Chang, Kadin, and Rohatgi [CKR95] showed that USat is *not* in **coNP** unless the polynomial hierarchy collapses to $\Sigma_3\mathbf{P}$ (in fact, they prove a stronger result).

USat can be reduced to FC [BRS98]. To see this, consider a fixed USat-formula, for example, $A(x_1, \ldots, x_n) = \bigwedge_{i=1}^{n} x_i$. Now let $F = F(x_1, \ldots, x_n) \in$ USat and let $\boldsymbol{a} = a_1 \cdots a_n$ be the unique satisfying assignment of F. Construct a congruence mapping as follows: negate a variable x_i if $a_i = 0$ and keep it unchanged otherwise. Then F becomes equivalent to A. Therefore we have $F \in$ USat $\Longrightarrow (F, A) \in$ FC. It is easy to see that the reverse direction holds as well.

Proposition 3.35 USat $\leq_m^p$ FC.

Intuitively we expect FI to be easier than FC because the congruence transformation seems to give more flexibility compared to a permutation. In order to reduce FI to FC, we need a way to force a congruence mapping to use permutations only, and *no* negations.

We start with a slightly simpler task. Let $F = F(x_1, \ldots, x_n)$ be a boolean formula. We want a *marking* or *labelling mechanism* for the variables of a boolean formula such that a labelled variable is a *fix point of any automorphism* of the formula. It is not clear whether there exist such labellings that are efficiently computable. However, the following weaker labelling often suffices.

Definition 3.36 *Variables x_i and x_j are called* equivalent (with respect to F), *if for any assignment $\boldsymbol{a} = a_1 \cdots a_n$ that satisfies F, we have $a_i = a_j$.*

$E_F(x_i)$ denotes the set of variables of F that are equivalent to x_i.

Consider any automorphism $\alpha \in \mathbf{Aut}(F)$. (Recall that by $\mathbf{Aut}(F)$ we denote the set of *automorphisms of F*.) If α maps x_i to x_k, then α

must map all variables equivalent to x_i to variables that are equivalent to x_k, i.e., $\alpha(E_F(x_i)) \subseteq E_F(x_k)$. Furthermore, any variable x_j that is mapped by α to a variable in $E_F(x_k)$ must belong to $E_F(x_i)$. Therefore, $\alpha(E_F(x_i)) = E_F(x_k)$. Since α is a bijection, we conclude that $E_F(x_i)$ and $E_F(x_k)$ must be of the same size.

We exploit this property to label variable x_i: take n new variables $z_1, \ldots, z_n$, and make them equivalent to x_i as follows. Define

$$\begin{aligned} L(x_i, z_1, \ldots, z_n) &= \bigwedge_{j=1}^{n} (x_i \leftrightarrow z_j), \text{ and} \\ F_{[i]} &= F \wedge L(x_i, z_1, \ldots, z_n). \end{aligned}$$

The new variables z_j of $F_{[i]}$ are referred to as the *labelling variables.*

Any assignment that satisfies $F_{[i]}$ must assign the same value to x_i and $z_1, \ldots, z_n$. Thus, x_i has more equivalent variables (with respect to $F_{[i]}$) than any other variable $x_k \notin E_{F_{[i]}}(x_i)$. Hence any automorphism α of $F_{[i]}$ maps $E_{F_{[i]}}(x_i)$ onto itself. We say that α *stabilizes* $E_{F_{[i]}}(x_i)$. Moreover, define α' to coincide with α except for the variables in $E_{F_{[i]}}(x_i)$, where α' is defined to be the identity. Then the resulting permutation is still an automorphism of $F_{[i]}$, with the additional property that it *pointwise stabilizes* $E_{F_{[i]}}(x_i)$.

Lemma 3.37 *For all* $\alpha \in \mathbf{Aut}(F_{[i]})$,

- $\alpha(E_{F_{[i]}}(x_i)) = E_{F_{[i]}}(x_i)$.
- *Define* α' *to coincide with* α *on all variables not in* $E_{F_{[i]}}(x_i)$ *and to be the identity on* $E_{F_{[i]}}(x_i)$. *Then* $\alpha' \in \mathbf{Aut}(F_{[i]})$.

Let $G = G(x_1, \ldots, x_n)$ be a second formula. We label variable x_j with the same label as x_i, namely $L(x_j, z_1, \ldots, z_n)$ and define $G_{[j]} = G \wedge L(x_j, z_1, \ldots, z_n)$. Then any isomorphism for $(F_{[i]}, G_{[j]})$ must map all the variables equivalent to x_j in $G_{[j]}$ to the variables equivalent to x_i in $F_{[i]}$.

Corollary 3.38 *For all* $\varphi \in \mathbf{Iso}(F_{[i]}, G_{[j]})$,

- $\varphi(E_G(x_j)) = E_F(x_i)$.
- *Define* φ' *to coincide with* φ *on all variables not in* $E_G(x_j)$ *and to map* x_j *to* x_i*, to be the identity on* $z_1, \ldots, z_n$*, and arbitrary on the remaining variables of* $E_G(x_j)$. *Then* $\varphi' \in \mathbf{Iso}(F_{[i]}, G_{[j]})$.

It follows that when two formulas $F_{[i]}$ and $G_{[j]}$ as above are isomorphic, we know that there is an isomorphism that maps x_j to x_i and keeps the labelling variables on themselves. We therefore omit to explicitly mention the labelling variables in a label and simply write $L(x_i, n)$ when we label x_i with n variables that do not yet occur in the considered formula.

We have one more technical lemma that states that we can w.l.o.g. assume that the all-zero- and the all-one-assignment do not satisfy a given instance for FI.

Lemma 3.39 *Let* (F_0, F_1) *be an instance for* FI *with* n *variables. For* $b \in \{0, 1\}$ *there exist formulas* G_0 *and* G_1 *with* $n+1$ *variables that have the following properties.*

- b^{n+1} *is* not *a satisfying assignment of* G_0 *and* G_1*, and*
- $(F_0, F_1) \in \mathrm{FI} \iff (G_0, G_1) \in \mathrm{FI}$.

Proof Assume first that $b = 0$. Define $G_i = F_i \wedge z$, for $i = 0, 1$ and a new variable z. Property (a) is obvious. To see (b), let φ be an isomorphism for (G_0, G_1), i.e. φ permutes the variables of G_1 so that it becomes equivalent to G_0. Note that any satisfying assignment for G_0 or G_1 must set z to one. Therefore φ must map z to itself or to a variable equivalent to z. In the latter case, we can modify φ to be the identity on $E_{G_1}(z)$ and we still have an isomorphism for (G_0, G_1). Now, φ is an isomorphism for (F_0, F_1) as well (just ignore variable z).

The case for $b = 1$ is analogous, just replace z by $\overline{z}$ in the definitions of G_0 and G_1. □

As our first application of the labelling mechanism we show that FI and FC are in fact equivalent problems [BRS98].

Theorem 3.40 FI $\equiv_m^p$ FC*.*

Proof Let F_0 and F_1 be two formulas with variables $x_1, \ldots, x_n$. To show that FI $\leq_m^p$ FC, assume that $F_0(1^n) = F_1(1^n) = 0$ (by Lemma 3.39). We define

$$G_i \;=\; \left((F_i \wedge z_0) \vee (z_1 \wedge \bigwedge_{j=1}^{n} x_j) \right) \wedge L(z_0, n+1) \wedge L(z_1, n+2).$$

Clearly, if $(F_0, F_1) \in \mathrm{FI}$ then $(G_0, G_1) \in \mathrm{FC}$. For the reverse direction, assume that $(G_0, G_1) \in \mathrm{FC}$. Note first that any congruence mapping has to stabilize variable z_0 and z_1 ($E_{G_1}(z_0)$ and $E_{G_1}(z_1)$, to be precise): it cannot permute them because of their label and it is not hard to see that it cannot negate them by our construction.

Then it follows that *no* congruence mapping can negate any variable, because for $z = 0$, the sub-formula $\bigwedge_{j=1}^{n} x_j$ has to stay true. Therefore any congruence for G_0 and G_1 is also an isomorphism for F_0 and F_1.

To see that FC $\leq_m^p$ FI, define

$$G_i \quad = \quad \left((F_i \vee z) \wedge (\overline{z} \vee \bigwedge_{j=1}^{n} (x_j \oplus y_j) \right) \wedge L(z, n+1).$$

Assume that $(F_0, F_1) \in$ FC and let $\nu \circ \varphi$ be a congruence. Then we can get an isomorphism for (G_0, G_1) from it by using the y_i's to simulate negations. More precisely, define

- $\pi(x_j) = x_k$ and $\pi(y_j) = y_k$, if $\nu \circ \varphi(x_j) = x_k$, and
- $\pi(x_j) = y_k$ and $\pi(y_j) = x_k$, if $\nu \circ \varphi(x_j) = \overline{x}_k$,

and let π be the identity on the remaining variables. Then π is an isomorphism for (G_0, G_1).

For the reverse direction, assume that $(G_0, G_1) \in$ FI and let π be an isomorphism. π has to stabilize variable z because of its label. Hence, for $z = 0$, π has to be an automorphism on the sub-formula $\bigwedge_{j=1}^{n} (x_j \oplus y_j)$. But this is possible only when π keeps the pairs (x_j, y_j) together. I.e., if $\pi(x_j) = x_k$ then $\pi(y_j) = y_k$, and if $\pi(x_j) = y_k$ then $\pi(y_j) = x_k$. But then we can define a congruence $\nu \circ \varphi$ for (F_0, F_1) in the same way (backwards) as we got π from $\nu \circ \varphi$ above. □

We summarize the reductions we have shown in the following corollary.

Corollary 3.41 **coNP** $\leq_m^p$ USAT $\leq_m^p$ FI $\equiv_m^p$ FC*.*

Recall that USAT is not in **coNP**, unless the polynomial hierarchy collapses [CKR95]. By Corollary 3.41 the same holds for FI and FC.

Before we can give more examples of problems reducible to FI, we need some more techniques. Namely, we show that one can combine two instances for FI according to boolean conjunction and disjunction.

3.2.2 And- and Or-functions

Definition 3.42 *An* and-function *for a set A is a function* $\mathsf{and} : \Sigma^* \times \Sigma^* \mapsto \Sigma^*$ *such that for any $x, y \in \Sigma^*$, we have $x \in A$* and *$y \in A$ if and only if* $\mathsf{and}(x, y) \in A$*. Similar, an* or-function or *for A fulfills $x \in A$* or *$y \in A$ if and only if* $\mathsf{or}(x, y) \in A$*.*

In this section, we show that FI has and- and or-functions. We start by developing some more labelling techniques.

In the previous section we showed how to label a single variable. A more general task is to force automorphisms to stabilize a *set of variables*. Let $\boldsymbol{x} = (x_1, \dots, x_n)$ and $\boldsymbol{y} = (y_1, \dots, y_m)$, let $F = F(\boldsymbol{x}, \boldsymbol{y})$ and suppose we want to consider only automorphisms of F that map x- to x-variables and y- to y-variables. Let $n \leq m$.

An obvious solution is provided by Lemma 3.37: label all variables y_i with label $L(y_i, m)$. The drawback of this solution is that the resulting formula can increase quadratically in size with respect to F. However, later on we need to apply the construction iteratively such that the resulting formula still has polynomial size. In order to guarantee this, the size of the formula we obtain should increase only linearly in size. Thus we need a new technique here. The following works: let s be a new variable and $M \geq 1$. Define formula $S(\boldsymbol{x}, \boldsymbol{y}, s, M)$ as follows:

$$S(\boldsymbol{x}, \boldsymbol{y}, s, M) \quad = \quad (\bigvee_{i=1}^{n} x_i \to s) \quad \wedge \quad (\bigvee_{i=1}^{m} y_i \to \overline{s}) \quad \wedge \quad L(s, M).$$

Let $S = S(\boldsymbol{x}, \boldsymbol{y}, s, n{+}m)$. S has the following property: let a be a satisfying assignment of S. If $\boldsymbol{a}$ assigns a 1 to any of the x-variables, then $\boldsymbol{a}(s) = 1$, which implies that $\boldsymbol{a}(\overline{s}) = 0$. Therefore $\boldsymbol{a}$ must assign 0 to all the y-variables. Symmetrically, if $\boldsymbol{a}$ assigns a 1 to any of the y-variables then $\boldsymbol{a}(\overline{s}) = 1$, which implies that $\boldsymbol{a}(s) = 0$. Therefore $\boldsymbol{a}$ must assign 0 to all the x-variables.

Now consider $F \wedge S$. We claim that any automorphism of $F \wedge S$ must map x- to x-variables and y- to y-variables, unless they are equivalent.

Lemma 3.43 *Let $F = F(\boldsymbol{x}, \boldsymbol{y})$ be a formula as above such that the all-zero assignment does not satisfy F. Let $\varphi \in \mathbf{Aut}(F \wedge S)$.*

- *φ maps x- to x-variables and y- to y-variables except, maybe, for variables x_i and y_j that are set to zero by* every *satisfying assignment of $F \wedge S$ (and which are therefore equivalent).*
- *Define φ' to coincide with φ on all variables that are set to one by at least one satisfying assignment of $F \wedge S$, and to be the identity on the remaining variables. Then $\varphi' \in \mathbf{Aut}(F \wedge S)$ and it maps x- to x-variables and y- to y-variables.*

Proof Any automorphism φ of $F \wedge S$ must stabilize s (more precisely: $E_{F \wedge S}(s)$) because of its label. Let $\boldsymbol{a}$ be an assignment that satisfies $F \wedge S$. By assumption, $\boldsymbol{a}$ is not the all-zero assignment. So let x_i be a variable such that $\boldsymbol{a}(x_i) = 1$ (the case that $\boldsymbol{a}(y_j) = 1$ for some y_j is analogous). Since $x_i \to s$, we have that $\boldsymbol{a}(s) = 1$. Since $y_j \to \overline{s}$, we have that $\boldsymbol{a}(y_j) = 0$

for $j = 1, \ldots, m$. Therefore, φ cannot map x_i to some y_j in order of $\varphi(a)$ to satisfy $F \wedge S$. We conclude that φ must map x_i to some x_j.

It follows that whenever φ maps, say, x_j to y_k, then any satisfying assignment of $F \wedge S$ assigns 0 to both, x_j and y_k, i.e., all these variables are in $E_F(x_j)$. If we modify φ to be the identity on $E_F(x_j)$ (and let it unchanged otherwise), we still have an automorphism for $F \wedge S$. The latter shows part 2 of the lemma. □

We extend the lemma to isomorphisms. Let $G = G(\boldsymbol{x}, \boldsymbol{y})$. Then any isomorphism of $(F \wedge S, G \wedge S)$ must map x- to x-variables and y- to y-variables, unless they are equivalent.

Corollary 3.44 *Let $F = F(\boldsymbol{x}, \boldsymbol{y})$ and $G = G(\boldsymbol{x}, \boldsymbol{y})$ be formulas such that the all-zero assignment does not satisfy F or G. If $(F, G) \in$ FI then there is a $\varphi \in \mathbf{Iso}(F \wedge S, G \wedge S)$ that maps all the x- to x-variables and all the y- to y-variables.*

We need an even more flexible labelling technique. Consider again $F = F(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x} = (x_1, \ldots, x_n)$ and $\boldsymbol{y} = (y_1, \ldots, y_n)$. This time, we allow automorphisms of F to map x- variables to y-variables, but only in a very restricted way:

- either x-variables are only mapped to x-variables
- or x-variables are only mapped to y-variables.

We can achieve this as follows. For new variables s and t and $M \geq 1$ define formula $ST(\boldsymbol{x}, \boldsymbol{y}, s, t, M)$ as follows:

$$\begin{aligned} ST(\boldsymbol{x}, \boldsymbol{y}, s, t, M) \quad = \quad & (\bigvee_{i=1}^{n} x_i \rightarrow s) \quad \wedge \quad (\bigvee_{i=1}^{n} y_i \rightarrow \overline{s}) \quad \wedge \quad (s \leftrightarrow \overline{t}) \\ & \wedge \quad L(s, M) \quad \wedge \quad L(t, M). \end{aligned}$$

Let $ST = ST(\boldsymbol{x}, \boldsymbol{y}, s, t, 2n)$. As above for S, a satisfying assignment of ST can assign a 1 to either any of the x-variables or any of the y-variables, but not to both. The difference to S is that now an automorphism of ST can interchange s and t because they have the same label.

Now consider $F \wedge ST$. We claim that any automorphism of $F \wedge ST$ either maps x- to x-variables and y- to y-variables, or interchanges x- and y-variables, unless they are equivalent.

Lemma 3.45 *Let F be a satisfiable formula as above such that the all-zero assignment does not satisfy F. For all $\varphi \in \mathbf{Aut}(F \wedge ST)$,*

- *either $\varphi(s) = s$ and then we have that φ maps x- to x-variables and y- to y-variables except, maybe, for variables x_i and y_j that are set to zero by* every *satisfying assignment of $F \wedge ST$.*
- *or $\varphi(s) = t$ and then we have that φ interchanges x- and y-variables except, maybe, for variables x_i, x_j or y_i, y_j that are set to zero by* every *satisfying assignment of $F \wedge ST$.*

Furthermore, we can modify φ to an automorphism of $F \wedge ST$ that keeps x- on x-variables in case (i). The case (ii) is more subtle: if we have the same number of x- and y-variables that have value zero in every *satisfying assignment of F, then we can modify φ to interchange all x- and y-variables in case (ii).*

Proof We have to distinguish two cases according to whether an automorphism φ maps s to s or t. In the case that $\varphi(s) = s$, we can directly use the proof of Lemma 3.43. Now let $\varphi(s) = t$ and let $\boldsymbol{a}$ be a satisfying assignment of $F \wedge ST$. Let x_i be a variable such that $\boldsymbol{a}(x_i) = 1$. Because $(x_i \to s)$ is a part of formula ST, we get that $\boldsymbol{a}(s) = 1$. Since $s \leftrightarrow \bar{t}$ is part of the formula ST, we must have $\boldsymbol{a}(t) = 0$.

Now consider formula $(F \wedge ST) \circ \varphi$ which is satisfied by the assignment $a \circ \varphi$. Since φ maps s to t, we have $\boldsymbol{a} \circ \varphi(s) = 0$, and correspondingly $\boldsymbol{a} \circ \varphi(t) = 1$. Therefore $a \circ \varphi$ must assign zero to all the x-variables. Hence φ must interchange x- and y-variables, except, maybe, variables that have value zero in all satisfying assignments.

Let x_i be a variable that has value zero in all satisfying assignments. Then all such variables are precisely those that are equivalent to x_i, i.e., that are in $E_F(x_i)$. Note that φ maps variables in $E_F(x_i)$ again to $E_F(x_i)$. Moreover, we can change φ arbitrarily on $E_F(x_i)$ and it still remains an automorphism for $F \wedge ST$. Therefore, if there is the same number of x-variables as y-variables in $E_F(x_i)$, we can modify φ to interchange all x-with the y-variables. □

A corresponding statement as in Lemma 3.45 holds for isomorphisms.

Corollary 3.46 *Let $F = F(\boldsymbol{x}, \boldsymbol{y})$ and $G = G(\boldsymbol{x}, \boldsymbol{y})$ be formulas such that the all-zero assignment does not satisfy F or G. If $(F, G) \in$* FI *then there is a $\varphi \in$* **Iso**$(F \wedge ST, G \wedge ST)$ *that*

- *either maps x- to x-variables and y- to y-variables,*
- *or interchanges x- and y-variables,*

Now we have all the tools to construct an and- and an or-function.

Theorem 3.47 FI *has and- and or-functions.*

Proof Let (F_0, F_1) and (G_0, G_1) be two instances for FI. The variables of F_0 and F_1 are $x_1, \ldots, x_n$, and the variables of G_0 and G_1 are $y_1, \ldots, y_m$. By Lemma 3.39 we can assume that the all-zero assignment does not satisfy any of these formulas.

And-function. For constructing the and-function, we simply combine the formulas by or-ing together F_0 and G_0 on one side, and F_1 and G_1 on the other side. However, we have to make sure that we don't get isomorphisms that map x-variables to y-variables. For this we use formula $S = S(\boldsymbol{x}, \boldsymbol{y}, s, M)$ from above, where $M = n + m$. Define

$$\mathsf{and}((F_0, F_1), (G_0, G_1)) = (C_0, C_1),$$

where formulas C_0 and C_1 are defined as follows:

$$\begin{aligned} C_0 &= (F_0 \vee G_0) \wedge S, \\ C_1 &= (F_1 \vee G_1) \wedge S. \end{aligned}$$

It remains to show that function and is indeed an and-function for FI. If $(F_0, F_1) \in \text{FI}$ and $(G_0, G_1) \in \text{FI}$, then clearly $(C_0, C_1) \in \text{FI}$. For the reverse direction assume that $(C_0, C_1) \in \text{FI}$. By Corollary 3.44 there is an isomorphism φ that maps x- to x-variables and y- to y-variables, i.e., φ can be written as $\varphi = \varphi_x \cup \varphi_y \cup \varphi_S$, where φ_x is a permutation on $\{x_1, \ldots, x_n\}$, φ_y on $\{y_1, \ldots, y_m\}$, and φ_S on the extra variables from formula S.

We argue that φ_x is an isomorphism for (F_0, F_1) (an analogous argument shows that φ_y is an isomorphism for (G_0, G_1)): consider the following partial assignments.

- $\boldsymbol{a}_y$ assigns zero to all the y-variables and
- $\boldsymbol{a}_S$ assigns one to variable s and the labelling variables.

Then we have $G_0(\boldsymbol{a}_y) = 0$, and $G_1(\varphi_y(\boldsymbol{a}_y)) = G_1(\boldsymbol{a}_y) = 0$. Furthermore, the second item implies that, for any assignment $\boldsymbol{a}_x$ of the x-variables, formula S evaluates to one on $\boldsymbol{a} = (\boldsymbol{a}_x, \boldsymbol{a}_y, \boldsymbol{a}_S)$ and on $\varphi(\boldsymbol{a})$. Now, since by assumption $C_0(\boldsymbol{a}) = C_1(\varphi(\boldsymbol{a})$, we conclude that $F_0(\boldsymbol{a}_x) = F_1(\varphi_x(\boldsymbol{a}_x)$. Therefore, φ_x is an isomorphism for (F_0, F_1).

Or-function. For constructing the or-function, we need a copy of the variables $\boldsymbol{x} = (x_1, \ldots, x_n)$ and $\boldsymbol{y} = (y_1, \ldots, y_m)$ used in formulas F_0, F_1, and G_0, G_1, respectively. Let $\boldsymbol{u} = (u_1, \ldots, u_n)$ and $\boldsymbol{v} = (v_1, \ldots, v_m)$ be new variables. Define

$$\mathsf{or}((F_0,F_1),(G_0,G_1)) \;=\; (D_0,D_1),$$

where formulas D_0 and D_1 are defined as follows:

$$\begin{aligned} D_0 &= ((F_0(\boldsymbol{x}) \vee G_0(\boldsymbol{y})) \;\vee\; (F_1(\boldsymbol{u}) \vee G_1(\boldsymbol{v}))) \;\wedge\; R, \\ D_1 &= ((F_1(\boldsymbol{x}) \vee G_0(\boldsymbol{y})) \;\vee\; (F_0(\boldsymbol{u}) \vee G_1(\boldsymbol{v}))) \;\wedge\; R, \end{aligned}$$

where

$$R \;=\; S((\boldsymbol{x},\boldsymbol{u}),(\boldsymbol{y},\boldsymbol{v}),s_0,M_0) \;\wedge\; ST((\boldsymbol{x},\boldsymbol{y}),(\boldsymbol{u},\boldsymbol{v}),s,t,M).$$

Here we set $M_0 = n+m$ and $M = 2M_0$. (The additional brackets for the variables in formulas S and ST indicate the two groups of variables that occur in the definition of these formulas.)

The rough idea of this definition is that if there is an isomorphism for either (F_0,F_1) or (G_0,G_1), then we can construct an isomorphism for (D_0,D_1) from it: just map the non-isomorphic formulas to themselves. Formula R will ensure that we don't get more isomorphisms than the ones just described. We give more details below.

Suppose first that $(F_0,F_1) \in$ FI and let φ_x be an isomorphism. Let φ_u be φ_x^{-1} but on the u-variables. That is, define

$$\varphi_u(u_i) \;=\; u_j, \quad \text{if } \varphi_x^{-1}(x_i) = x_j.$$

Define φ as the union of φ_x and φ_u and the identity on all the other variables of D_1. Then it is straightforward to check that φ is an isomorphism of (D_0,D_1) which is therefore in FI.

Now assume that $(G_0,G_1) \in$ FI via isomorphism φ_y. Then we get an isomorphism φ for (D_0,D_1) as follows. Define

$$\begin{aligned} \varphi(x_i) &= u_i, \\ \varphi(u_i) &= x_i, \\ \varphi(v_i) &= y_j, \quad \text{if } \varphi_y(y_i) = y_j, \\ \varphi(y_i) &= v_j, \quad \text{if } \varphi_y^{-1}(y_i) = y_j, \\ \varphi(s) &= t, \\ \varphi(t) &= s, \\ \varphi(s_0) &= s_0. \end{aligned}$$

The remaining variables that come from the labelling process are mapped according to the variables they are equivalent to.

In summary, the isomorphisms we constructed have the following property:

- either they map s and t to itself, respectively, and map x- to x-variables, u- to u-variables, y- to y-variables, and v- to v-variables,
- or they interchange s with t and interchange x- with u-variables and y- with v-variables.

Conversely, if there is an isomorphism for (D_0, D_1) that fulfills property (i) or (ii), then it is easy to see that we get an isomorphism for either (F_0, F_1) or (G_0, G_1) from it, respectively. It remains to show that every isomorphism for (D_0, D_1) must satisfy one of these two properties. The only exception from this might be equivalent variables.

Assume that (D_0, D_1) are isomorphic. We consider formula R. By Corollary 3.44, its first part, $S((\boldsymbol{x}, \boldsymbol{u}), (\boldsymbol{y}, \boldsymbol{v}), s_0, M_0)$, implies that there is an isomorphism φ of (D_0, D_1) that can be written as a union of permutations $\varphi_{x,u}$ on x- and u-variables, $\varphi_{y,v}$ on y- and v-variables, $\varphi_{s,t}$ on s and t, and φ_L for the remaining variables from the labelling. Combined with its second part, $ST((\boldsymbol{x}, \boldsymbol{y}), (\boldsymbol{u}, \boldsymbol{v}), s, t, M)$, we get by Corollary 3.46 that $\varphi_{x,u}$, depending on $\varphi_{s,t}$, either maps all x-variables to x-variables or interchanges x- and u-variables. The same holds analogously for $\varphi_{y,v}$. Thus we get an isomorphism that fulfills property (i) or (ii) above. $\square$

We can extend the functions and and or to more than two arguments: suppose we want to build the conjunction or disjunction of k pairs of formulas, each of length at most l. We combine them in a binary tree like fashion with the above functions for two arguments. Since the size of the output of our functions and and or is linear in the size of the input formulas, the size of the formulas grows by some constant factor c when we go from one level of the tree to the next. The length of the resulting formula is therefore bounded by $c^{\log k} l$ which is a polynomial in k and l.

The extended and- and or-functions can be used to turn conjunctive and disjunctive truth-table reductions to FI into many-one reductions: consider a many-one reduction f from set A to a set B. On input x, it produces exactly one string $y = f(x)$ such that $x \in A \Longleftrightarrow y \in B$. A generalization of this are *disjunctive truth-table reductions*, where $f(x)$ is a list of strings, say $y_1, \ldots, y_k$, such that $x \in A \Longleftrightarrow \exists i : \; y_i \in B$. Analogously one can define *conjunctive truth-table reductions*, just replace the existential quantifier by a universal one. Now we use the and- and or-function to combine the queries of a conjunctive and disjunctive truth-table reduction into just one query, respectively, thereby providing a many-one reduction to FI.

Corollary 3.48 *If a set L is disjunctively or conjunctively truth-table reducible to* FI*, then $L \leq_m^p$* FI.

3.2.3 FA and UOCLIQUE

We consider the formula automorphism problem, FA. It is known that for graphs we have GA $\leq_m^p$ GI. We ask whether this relationship carries over to formulas.

Given a formula $F = F(x_1, \ldots, x_n)$, we want to know whether there exists a permutation $\alpha \neq id$ such that $F \equiv F \circ \alpha$. The condition that $\alpha \neq id$ requires that there have to exist $i \neq j$ such that $\alpha(x_i) = x_j$. Hence we have

$$F \in \mathrm{FA} \iff \exists i \neq j : (F_{[i]}, F_{[j]}) \in \mathrm{FI}.$$

We conclude that FA is disjunctively truth-table reducible to FI and hence many-one reducible by Corollary 3.48.

Theorem 3.49 FA $\leq_m^p$ FI.

Recall that USAT $\leq_m^p$ FI (Corollary 3.41). A problem seemingly harder than USAT is the *Unique Optimal Clique problem* (UOCLIQUE) , that is, whether the largest clique of a given graph is unique. The standard reduction from SAT to CLIQUE also reduces USAT to UOCLIQUE, i.e., we have USAT $\leq_m^p$ UOCLIQUE.

USAT can be written as the difference of two **NP**-sets. Namely, define A to be the set of satisfiable formulas that have at least two satisfying assignments. Then $A \in \mathbf{NP}$ and we have USAT = SAT $- A$. On the other hand, UOCLIQUE is not known to be in the *boolean closure of* **NP**. In other words, we don't know *any* boolean expression over **NP**-sets that expresses UOCLIQUE.

An upper bound on the complexity of UOCLIQUE is $\mathbf{P}^{\mathbf{NP}[\log]}$, the class of sets that can be computed in polynomial time with logarithmically many queries to an **NP**-oracle: with a binary search one can compute the size of the largest clique of a given graph. Then, with one more query, one can find out whether there is more than one clique of that size.

Papadimitriou and Zachos [PZ83a] asked whether UOCLIQUE is complete for $\mathbf{P}^{\mathbf{NP}[\log]}$. This is still an open problem. Buhrman and Thierauf [BT96] provide strong evidence that UOCLIQUE is *not* complete for $\mathbf{P}^{\mathbf{NP}[\log]}$.

UOCLIQUE can be disjunctively reduced to USAT [BT96]. To see this, let UCLIQUE be the unique CLIQUE version. That is, given a graph G and a integer k, decide whether G has a unique clique of size k. Now, observe that

$$G \in \mathrm{UOCLIQUE} \iff \exists k : (G, k) \in \mathrm{UCLIQUE}.$$

Furthermore, we have UClique $\leq_m^p$ USat. This is because the generic reduction from Cook [Coo71] that reduces an arbitrary **NP**-set to Sat is *parsimonious*: it maintains the number of solutions that instances have. Therefore the reduction from Clique to Sat is also a reduction from UClique to USat.

Combined with the many-one reduction from USat to FI, we get a disjunctive truth-table reduction from UOClique to FI. Finally, we invoke Corollary 3.48 and turn this into a many-one reduction from UOClique to FI.

Theorem 3.50 UOClique $\leq_m^p$ FI.

An very interesting open problem is whether USat, UOClique, or FI are **NP**-hard. Note however, that they are known to be **NP**-hard under *randomized reductions* [VV86].

Chapter 4

Branching Programs

In this chapter we study another important computational model for boolean functions: *branching programs*, introduced by Lee [Lee59].

Definition 4.51 *A* (deterministic) branching program *B in n variables $x_1, \dots, x_n$ is a directed acyclic graph with the following type of nodes. There is a single node of in-degree zero, the* initial node *or* root *of B. All nodes have out-degree two or zero. A node u with out-degree two is an* internal node *of B and is labelled with a variable x_i, for some $i \in \{1, \dots, n\}$. One of its outgoing edges is labelled with 0 and leads to node $e(u, 0)$. The other is labelled with 1 and leads to node $e(u, 1)$. A node with out-degree zero is a* final node *of B. The final nodes are labelled either with 1, for an* accepting node *or with 0, for a* rejecting node.*The* size *of a branching program is the number of its nodes.*

A branching program B in n variables defines an n-ary boolean function from $\{0,1\}^n$ to $\{0,1\}$ as follows: for an assignment *$\boldsymbol{a} = (a_1, \dots, a_n) \in \{0,1\}^n$, we walk through B, starting at the initial node, always following the (unique) edge labelled a_i when the node has label x_i, until we reach a final node. The function value is the label of the final node.*

We say that a set L has branching programs of size *$s(n)$, if $L^{=n}$ (considered as a boolean function from $\{0,1\}^n$ to $\{0,1\}$) can be computed by a branching program of size $s(n)$, for all n.*

There is a tight relationship between the branching program size and the space complexity of (nonuniform) Turing machines [Cob66, PZ83b]. , as we will see in Theorem 4.53 below.

Definition 4.52 **L** (**NL**) *denotes the class of sets that can be accepted by (nondeterministic) logarithmically space bounded Turing machines.*

L/poly *is the* nonuniform *version of* **L***: for every input length we have a polynomial* advice *for free. Formally, $L \in$* **L/poly** *if there exist $A \in$* **L** *and $h : \mathbf{N} \to \Sigma^*$ such that h is polynomially bounded and for all $x \in \Sigma^*$*

$$x \in L \iff (x, h(|x|)) \in A.$$

Theorem 4.53 *For any $L \subseteq \Sigma^*$, L has polynomial size branching programs iff $L \in$* **L/poly***.*

Proof Let B_n be a branching program for $L^{=n}$. Given B_n and an input x, a Turing machine M can evaluate B_n on input by traversing down the unique path defined by the input x. M has to store the actual node of B_n that is evaluated. Therefore this can be done within space $O(\log |B_n|)$. It follows that $L \in$ **L/poly**, with B_n as the advice string for inputs of length n.

For the reverse direction let M be a $s(n)$ space bounded machine. By elementary counting, M has at most $2^{O(s(n))}$ configurations on input $x = x_1 \cdots x_n$ of length n. Define a branching program B_n as follows: B_n has one node for each configuration of M. The transition of M from one configuration to another depends only on one input bit x_i: the one that is read by the head on the input tape. Hence we will label this configuration with x_i and connect its 0- and 1-edge with the corresponding successor configuration nodes. Initial, accepting, and rejecting nodes are defined as the corresponding configurations of M.

B_n has size $2^{O(s(n))}$, which is polynomial for $s(n) = O(\log n)$. Furthermore, B_n accepts x iff M accepts x. □

It is known that **L/poly** $\subseteq$ $\mathbf{AC}^1$ (see [Joh90]), where $\mathbf{AC}^1$ is the class of functions computed by polynomial-size, logarithmic-depth circuits with unbounded fan-in $\wedge$- and $\vee$-gates. Hence, for any branching program there is an equivalent circuit of size polynomial and logarithmic depth (with respect to the program size). The following theorem makes a more precise statement with respect to the circuit size, but ignores its depth (see [Weg86]).

Theorem 4.54 *For each branching program B there exists an equivalent circuit C of size $|C| = O(|B|)$.*

Proof One way to evaluate a branching program B on some input, is to start at the initial node and walk along the unique path described by the input, until a final node is reached.

Another way is to take the reverse direction: start at the final nodes with values 1 and 0 for the accepting and rejecting nodes, respectively.

Then shift the values along edges (in reverse direction) that are consistent with the given input. Just one value will reach the initial node of B which is the correct value.

This process can easily be accomplished by a circuit: replace each node u of B by a circuit as follows. The accepting and rejecting nodes are replaced by the constant 1 and 0 circuits, respectively. Now consider an inner node u with label x_i. Inductively assume that circuits C_0 and C_1 replace the successor nodes $e(u,0)$ and $e(u,1)$ of u, respectively. The circuit C at u should select the result of the circuit that follows the consistent edge. This is achieved by defining $C = x_i C_1 \vee \overline{x}_i C_0$.

The circuit constructed that way will be equivalent to B. Each node of B contributes at most three gates to our circuit. Therefore we have $|C| \leq 3|B|$. $\square$

Next we show that branching programs can compute boolean formulas within the same size. This follows from the fact that one can combine several branching programs $B_1, \ldots, B_k$ according to some boolean operation into one branching program by concatenating them appropriately: suppose we want to compute the conjunction of $B_1, \ldots, B_k$. We start with B_1 and replace the accepting node of B_i by the initial node of B_{i+1}, for $i = 1, \ldots, k-1$. The accepting node of B_k becomes the accepting node of the resulting branching program B. All rejecting nodes just stay rejecting. Then B computes $\bigwedge_{i=1}^k B_i$. The size of B is $\sum_{i=1}^k |B_i|$.

The case of disjunction is analogous but with the roles of accepting and rejecting nodes reversed. Also complementation of a branching program is easy: simply exchange accepting with rejecting nodes.

Lemma 4.55 *Branching programs can be combined in polynomial time according to any boolean operation.*

Note that the above construction essentially builds a composition of functions computed by branching programs. Therefore we have more general:

Lemma 4.56 *Let $f(x_1, \ldots, x_n)$ be a function that can be computed by a branching program of size s_f. Let furthermore $g_1, \ldots, g_n$ be functions that can be computed by branching programs of size at most s_g. Then there exists a branching program of size $s_f s_g$ that computes $f(g_1, \ldots, g_n)$.*

Given a boolean formula F we can apply Lemma 4.56 successively for sub-formulas of F and build up a branching program for F. Being a bit more careful with the size bound, we get

Corollary 4.57 *For each formula F that uses boolean operations $\{\wedge, \vee, \neg\}$, there exists an equivalent branching program B of size $|B| \leq |F|$.*

If there are also boolean operations other than $\{\wedge, \vee, \neg\}$ in F, then $|B|$ might become larger than $|F|$, but it still remains polynomially bounded in $|F|$.

Theorem 4.54 and Corollary 4.57 imply that the size of the smallest branching that computes some function f lies (up to a constant factor) between the size of the smallest circuit and the size of the smallest formula that compute f.

Since the satisfiability problem for boolean formulas, SAT, is **NP**-complete, this carries over to branching programs by Corollary 4.57. Note that B can be constructed from F in polynomial time.

Corollary 4.58 *The satisfiability problem for branching programs is* **NP***-complete. Consequently, the equivalence problem for branching programs is* **coNP***-complete.*

We conclude that the standard problems have the same complexity for circuits, formulas and branching programs. In particular, the results from Chapter 3 carry over to branching programs: it follows from Theorem 4.54 that the isomorphism problem for branching programs cannot be $\Sigma_2\mathbf{P}$-complete, unless **PH** collapses to $\Sigma_3\mathbf{P}$.

In the following sections we consider restricted kinds of branching programs. One reason for them to be interesting is their use in applications, as for example in circuit verification (see Bryant [Bry92] for an overview).

4.1 Read-k-Times Branching Programs

The restrictions on branching programs usually considered bound the number of times a variable can be tested.

Definition 4.9 *A branching program is called* (syntactic) read-k-times*, if, on each path from the initial node to a final node, every variable occurs at most k times as a node label.*

Whereas our definition considers a branching program as a graph, in the corresponding *non-syntactic* version, it should not be possible to test a variable more than k times on any input, although there might be paths (in the graph) where a variable occurs more than k times. But these can only be *inconsistent* paths that traverse the 0- *and* the 1-edge of some variable. For *read-once* branching program, first considered by Masek [Mas76], there is no difference between the two versions.

4.1.1 An Example

We start by giving an example of a function that can be computed by a small read-once branching program. For $x = x_1 \cdots x_n \in \{0,1\}^n$, define the *weight of* x, as the number of 1's in x, i.e., $w(x) = \sum_{i=1}^n x_i$. Then the function Hidden-Weighted-Bit (HWB for short) outputs the input bit at the weight position:

$$\mathrm{HWB}(x) \quad = \quad \begin{cases} x_{w(x)} & \text{if } w(x) > 0, \\ 0 & \text{otherwise.} \end{cases}$$

Hidden-Weighted-Bit can easily be computed by *read-twice* branching programs: scan the input once to compute its weight, and then select the corresponding bit in a second pass. (We will see in more detail how this can be done in Section 4.3.) Sieling and Wegener [SW95] (see also [GM94]) show that in fact read-once branching programs suffice. The idea is based on a *dynamic programming* approach to Hidden-Weighted-Bit.

Consider an input $x = x_1 \cdots x_n \in \{0,1\}^n$. We try to reduce the input by one bit, say x_n. There are two cases.

- *Case 1*: $x_n = 0$. Then we have $\mathrm{HWB}(x_1 \cdots x_n) = \mathrm{HWB}(x_1 \cdots x_{n-1})$.
- *Case 2*: $x_n = 1$. Then we have $\mathrm{HWB}(x_1 \cdots x_n) = x_{w(x_1 \cdots x_{n-1})+1}$ which we cannot express in terms of HWB. Therefore, we define an auxiliary function G that, on input $x = x_1 \cdots x_n \in \{0,1\}^n$, outputs the input bit at position $w(x) + 1$:

$$G(x) \quad = \quad \begin{cases} x_{w(x)+1} & \text{if } w(x) < n, \\ 1 & \text{otherwise.} \end{cases}$$

  Then we have $\mathrm{HWB}(x_1 \cdots x_n) = G(x_1 \cdots x_{n-1})$ in this case.

More general, we have for all $1 \le i < j \le n$,

$$\mathrm{HWB}(x_i \cdots x_j) \quad = \quad \begin{cases} \mathrm{HWB}(x_i \cdots x_{j-1}) & \text{if } x_j = 0, \\ G(x_i \cdots x_{j-1}) & \text{if } x_j = 1, \end{cases}$$

The crucial point now is that we can symmetrically express G in terms of HWB and G on smaller inputs. Namely, we have

$$G(x_i \cdots x_j) \quad = \quad \begin{cases} \mathrm{HWB}(x_{i+1} \cdots x_j) & \text{if } x_i = 0, \\ G(x_{i+1} \cdots x_j) & \text{if } x_i = 1. \end{cases}$$

For the initial cases we have for $1 \leq i \leq n$

$$\mathrm{HWB}(x_i) = G(x_i) = x_i.$$

The construction of the branching program for HWB is as follows: we simultaneously compute HWB and G according to the above equations. At the root, we test variable x_n and have to compute $\mathrm{HWB}(x_1, \ldots, x_{n-1})$ and $G(x_1, \ldots, x_{n-1})$ at the second level. To compute $\mathrm{HWB}(x_1, \ldots, x_{n-1})$, we put a node with label x_{n-1} and branch to compute $\mathrm{HWB}(x_1, \ldots, x_{n-2})$ and $G(x_1, \ldots, x_{n-2})$. To compute $G(x_1, \ldots, x_{n-1})$, we put a node with label x_1 and branch to compute $\mathrm{HWB}(x_2, \ldots, x_{n-1})$ and $G(x_2, \ldots, x_{n-1})$ in the third level, and so on.

At each subsequent level the length of the input decreases by one. The number of sub-functions with k variables for the two functions is $2(n-k)$. Therefore the number of nodes of the resulting branching program is less than $2n^2$. Clearly, each variable is read only once on each path from the initial node to a leaf.

4.1.2 The Satisfiability Problem

Let us consider the satisfiability problem for read-k-times branching program. A subtlety here is that this includes testing whether a given branching program B has in fact the read-k-times property. B is *not* read-k-times, iff there exit a variable x_i and a path in B where x_i is tested more than k times. The latter condition is a reachability problem on graphs combined with counting the number of occurances of a certain variable. Hence this can be accomplished in **NL**, and therefore in polynomial time. Hence, in the following we will assume that the programs under consideration have in fact read-k-times property.

For $k = 1$ the satisfiability problem is easy: a read-once branching program is satisfiable iff there is a path from the initial to the accepting node. So this is a reachability problem on graphs which can again be solved in **NL** and therefore in polynomial time.

How about $k = 2$? The argument just used for read-once programs does not work anymore. In fact, we crossed the borderline [BSSW98]:

Theorem 4.10 *The satisfiability problem for read-twice branching programs is* **NP***-complete.*

Proof Let $F(x_1, \ldots, x_n)$ be a CNF-formula that has m clauses, $F = C_1 \wedge \cdots \wedge C_m$. We apply the construction in Lemma 4.56. The branching program B we get considers the literals of each clause C_i until it finds one that satisfies C_i. Then it goes to the next clause C_{i+1}. It accepts iff all clauses have been satisfied.

B can read a variable as often as it occurs in F. Therefore it might not be read-twice yet. Recall that one can reduce the number of occurrences of variables in F down to three such that the new formula is satisfiable iff F is satisfiable. This is achieved by introducing *new variables*.

We consider variable x_1. With the other variables we do the same. Take $m-1$ new variables $y_1, \dots, y_{m-1}$. If x_1 occurs in the i-th clause C_i of F, replace x_1 by y_i. Then we force the y_i-variables to have the same value as x_1 for any satisfying assignment, by appending the formula

$$(x_1 \to y_1) \wedge (y_1 \to y_2) \wedge \cdots \wedge (y_{m-1} \to x_1) \tag{4.1}$$

to F. Clearly F is satisfiable iff the resulting formula, call it G, is satisfiable.

G consists of two parts. The first part is F with new names for the variables. Here, every variable occurs once now. The second part forces the new variables to coincide with the old ones that they replace. There are two more occurrences of each variable, and hence three in total. In a branching program we can manage that the two occurrences in the second part of G are on different paths, so that the program is read-twice.

It is enough to give a branching program for the formula (4.1). The initial node is labelled by x_1. If x_1 has value 0 then all y_i's have to have value 0 too. So following the 0-edge of x_1 there is a chain of $m-1$ nodes labelled $y_1, \dots, y_{m-1}$, respectively. The 0-edge of y_i goes to y_{i+1} except for y_{m-1} where it goes to the accepting node. All 1-edges of the chain go to the rejecting node. Analogously, there is a chain starting from the 1-edge of x_1 that checks that all y_i' have value 1. □

Corollary 4.11 *The equivalence problem for read-twice branching programs is* **coNP**-*complete.*

The equivalence problem for *read-once* branching programs is more tricky. In fact, no polynomial-time algorithm is known to check the equivalence of two read-once branching programs. However, Blum, Chandra, and Wegman [BCW80] give a *randomized* polynomial-time algorithm for the problem which puts it in **coRP** (see also [Sch95]). Their algorithm is based on the fact that one can *arithmetize* branching programs, that is, one can transform a branching program into an equivalent arithmetic expression.

4.1.3 Arithmetization

Given a branching program B, our goal is to derive an arithmetic expression, i.e., a polynomial p_B (over the rationals, say), such that B and p_B agree on all inputs for B.

As a first step, note that branching program B can be seen as a compact way of denoting a DNF-formula: each path of B from the initial to the accepting node is a conjunction of literals. Namely, if a node is labeled with variable x_i, the zero-edge from this node contributes $\overline{x}_i$ to the conjunction, whereas the one-edge contributes x_i. The boolean function represented by B is the disjunction of all these conjunctions.

A DNF-formula can easily be transformed into an arithmetic expression: keep a variable x_i as x_i and replace a negated variable $\overline{x}_i$ by $1 - x_i$. With respect to the operations, replace a conjunction by multiplication and a disjunction by addition over $\mathbf{Q}$, the rational numbers. Let p_B be the resulting polynomial.

We argue that B and p_B agree on $\{0,1\}^n$. That is,

$$B(\boldsymbol{a}) \quad = \quad p_B(\boldsymbol{a}) \quad \text{for all } \boldsymbol{a} \in \{0,1\}^n.$$

Consider an assignment $\boldsymbol{a} \in \{0,1\}^n$. If $B(\boldsymbol{a}) = 1$, then exactly one path of B evaluates to true under $\boldsymbol{a}$. Therefore, precisely one product term in p_B will be one on input $\boldsymbol{a}$, i.e., $p_B(\boldsymbol{a}) = 1$. On the other hand, If $B(\boldsymbol{a}) = 0$, then no path of B evaluates to true under $\boldsymbol{a}$. Therefore, all product terms in p_B will be zero on input $\boldsymbol{a}$, i.e., $p_B(\boldsymbol{a}) = 0$.

The degree of p_B is at most n in general. However, if B is a read-k-times branching program, then the degree is bounded by k. In particular, p_B is multilinear if B is read-once.

Written as a sum of monomials, p_B may consist of exponentially many terms in the size of B. Therefore we cannot explicitly write down this polynomial as a sum of monomials in polynomial time. Nevertheless, we can construct an arithmetic expression for p_B in polynomial time that can be evaluated efficiently at any point in $\mathbf{Q}^n$.

Theorem 4.12 *For each branching program B one can construct in polynomial time a polynomial p_B over $\mathbf{Q}$ such that B and p_B are equivalent on $\{0,1\}^n$, where n is the number of variables. Moreover, if B is read-once, then p_B is multilinear.*

Proof We assign a polynomial to each node and each edge of B.

- To the root, we assign the constant 1 polynomial.

- If we have a polynomial p_u for node u and u is labelled with variable x_i, we assign polynomials $x_i p_u$ and $(1 - x_i)p_u$ to the 1- and 0-edge of u, respectively.
- Suppose $q_1, \dots, q_l$ are the polynomials at the l incoming edges of node v, then we assign $p_v = \sum_{i=1}^{l} q_i$ to node v.

Finally, p_B is the polynomial at the accepting node of B. The size of p_B is bounded by $O(|B|)$. □

We show in Section 4.2 how to solve the equivalence problem for polynomials by a randomized polynomial-time algorithm. This covers read-once branching programs as a special case by Theorem 4.12 (Corollary 4.19).

Arithmetization is also a basic tool when we study the *isomorphism problem* for read-once branching programs. Because of the randomized algorithm for the equivalence problem, the isomorphism problem is in $\mathbf{NP} \cdot \mathbf{coRP}$. We ask whether it is **NP**-hard. In Section 4.2.2 we provide a constant round interactive proof system for the corresponding *non-isomorphism* problem. This shows that the non-isomorphism problem for read-once branching programs is in $\mathbf{BP} \cdot \mathbf{NP}$. As in the case of the graph isomorphism problem (see Corollary 2.11) we get that it is not **NP**-hard unless the polynomial hierarchy collapses to $\Sigma_2\mathbf{P}$ (Corollary 4.24).

4.1.4 $(1, +k)$-Branching Programs

An extension of read-once branching programs are (syntactic) $(1, +k)$-branching programs, where, on every path, there can be up to k variables that are tested more than once. Note that different variables can be tested more than once on different paths.

Savický [Sav98] showed that the satisfiability and equivalence test for read-once branching programs can be extended to the case of $(1, +k)$-branching programs. We give the argument for $k = 1$ which also provides the crucial idea for the general case.

Let us see first how to check the $(1, +1)$-property: the property does *not* hold, iff there are two variables x_i and x_j $(i \neq j)$ and there is a path such that x_i and x_j are tested more than once on that path. This can be decided in **NL**, and hence in polynomial time. In the following we will therefore assume that the programs under consideration have in fact the $(1, +1)$-property.

The Satisfiability Problem

The satisfiability problem for read-once branching programs is simply a reachability problem on graphs. This is no longer true for $(1,+1)$-branching programs, because here we should only consider *consistent paths* with respect to the variable that is tested more than once on a path.

The idea for the satisfiability test is to partition a given $(1,+1)$-branching program B into consistent paths where a specific variable is tested more than once. That is, for $i = 1, \dots, n$ and $b \in \{0,1\}$, define predicates

$$\begin{aligned} P_i^b(B) \quad &\iff \quad \text{there exists an accepting path in } B \text{ where} \\ &\qquad\quad x_i \text{ is tested more than once and } x_i = b. \end{aligned}$$

Predicates $P_i^b(B)$ can be decided in polynomial time (in fact, in **NL**). If at least one predicate $P_i^b(B)$ is true, then B is satisfiable.

However, even if all the predicates fail, B might still accept some input via paths that are read-once. For those we do a counting argument. Namely, define

$$\begin{aligned} N(B) &= \text{the number of accepting paths in } B\text{, and} \\ N_i(B) &= \text{the number of accepting paths in } B \text{ where} \\ &\qquad x_i \text{ is tested more than once.} \end{aligned}$$

Note that $N(B)$ and $N_i(B)$ also count inconsistent path. So these numbers are just counting paths in a graph, a problem that is captured by the counting logspace class #**L**. Recall that #**L**-functions can be computed in polynomial time.

Now observe that the paths not counted by the numbers $N_i(B)$ are the read-once paths. Hence B accepts an input via some read-once path iff $N(B) - \sum_i N_i(B) > 0$.

In summary, we have that the $(1,+1)$-branching program B is satisfiable, iff

- $P_i^b(B)$ holds for some i and b, or
- $N(B) - \sum_i N_i(B) > 0$.

Theorem 4.13 *The satisfiability problem for* $(1,+1)$*-branching programs is in* **P**.

Sieling [Sie98] showed that the satisfiability problem for *non-syntactic* $(1,+1)$-branching programs is **NP**-complete. This follows from a modification of the **NP**-completeness proof for read-twice branching programs (Theorem 4.10).

The Equivalence Problem

To solve the equivalence problem for $(1,+1)$-branching programs, we do again arithmetization. That is, we reduce the problem to an equivalence test for polynomials. However, this only works if we can get multilinear polynomials, as in the case of read-once branching programs. But the arithmetization described in Section 4.1.3 would yield higher degree polynomials in the case of $(1,+1)$-branching programs.

To get around this problem, we modify the arithmetization process such that the resulting polynomial is multilinear and still equivalent over $\{0,1\}$. Like in satisfiability test above, we partition a given $(1,+1)$-branching program B into consistent paths where a specific variable is tested more than once. More precisely, let us consider the case $x_i = 1$. Cancel the 0-edges of x_i in B and call the resulting program B'. Write the arithmetization of B' as

$$p_{B'}(\boldsymbol{x}) \quad = \quad a_d x_i^d + \cdots + a_2 x_i^2 + a_1 x_i + a_0, \tag{4.2}$$

where the coefficients a_t are polynomials in the remaining variables $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$. The term $a_t x_i^t$ corresponds to the paths in B' where x_i occurs t times along a 1-edge. It follows that a_t is multilinear for $t \geq 2$.

We transform $p_{B'}$ into a multilinear polynomial by cancelling the term $a_1 x_i + a_0$ and replacing x_i^t by x_i in the remaining terms. That is, define polynomial

$$p_i^1(\boldsymbol{x}) \quad = \quad x_i(a_d + \cdots + a_2).$$

The expression for p_i^1 can be computed in polynomial time very similar to the arithmetization shown in Section 4.1.3. The only difference is to keep track of the number of occurrences of x_i. The same consideration for $x_i = 0$ leads to polynomial p_i^0. Just replace x_i by $1 - x_i$.

The sum

$$p(\boldsymbol{x}) \quad = \quad \sum_i \left(p_i^0(\boldsymbol{x}) + p_i^1(\boldsymbol{x})\right)$$

arithmetizes the consistent part of B where a variable is tested at least twice.

It remains to arithmetize the read-once part of B. Let B_i be the subprogram of all paths of B where variable x_i is tested more than once (i.e., including inconsistent paths). Define

$$q(\boldsymbol{x}) \quad = \quad p_B(\boldsymbol{x}) - \sum_i p_{B_i}(\boldsymbol{x}).$$

Then q is the multilinear arithmetization of the read-once part of B. Hence, $p + q$ is a multilinear polynomial that agrees with B on $\{0,1\}^n$, and that can be constructed in polynomial time.

Theorem 4.14 *For every* $(1,+1)$*-branching program one can construct in polynomial time a multilinear polynomial that is equivalent on* 0-1-*inputs.*

The theorem can be extended to $(1,+k)$-branching programs [Sav98].

4.2 Polynomials

Let $\mathbf{F}$ be some field and $p = p(x_1, \ldots, x_n)$ be a multivariate polynomial over $\mathbf{F}$. The *degree*of p is the maximum exponent of any variable when p is written as a sum of monomials (i.e., products of the variables). Polynomials of degree 1 are called *multilinear*. Note the difference to the *total degree* of a polynomial, where one first adds the exponents of the variables in each monomial and then takes the maximum over these sums.

4.2.1 The Equivalence Problem

Given some polynomial p written as an arithmetic expression, we want to find out whether p is in fact the zero polynomial. Note that the obvious algorithm, namely to transform the arithmetic expression in a sum of monomials and check whether all coefficients are zero, can have up to exponential running time (in the size of the input). Efficient *probabilistic* zero-tests were developed by Schwartz [Sch80] and Zippel [Zip79]. The version below is a variant shown by Ibarra and Moran [IM83]. They extended the corresponding theorem for multilinear polynomials shown by Blum, Chandra, and Wegman [BCW80] to arbitrary degrees.

Theorem 4.15 *Let* $p(x_1, \ldots, x_n)$ *be a multivariate polynomial of degree* d *over field* $\mathbf{F}$ *that is not the zero polynomial. Let* $T \subseteq \mathbf{F}$ *with* $\|T\| \geq d$. *Then there are at least* $(\|T\| - d)^n$ *points* $(a_1, \ldots, a_n) \in T^n$ *such that* $p(a_1, \ldots, a_n) \neq 0$.

Proof The proof is by induction on n. For $n = 1$ the theorem is true because a degree d polynomial has at most d roots in $\mathbf{F}$.

Let $n > 1$ and let $p(x_1, \ldots, x_n)$ be a nonzero polynomial of degree d. Let furthermore $\boldsymbol{a} = (a_1, \ldots, a_n)$ be a point such that $p(\boldsymbol{a}) \neq 0$. We define two polynomials, both are sub-functions of p.

$$\begin{aligned} p_0(x_1, \ldots, x_{n-1}) &= p(x_1, \ldots, x_{n-1}, a_n), \\ p_1(x_n) &= p(a_1, \ldots, a_{n-1}, x_n). \end{aligned}$$

By construction, both polynomials are nonzero and have degree bounded by d. p_0 has $n-1$ variables and therefore differs from 0 on at least $(\|T\| - d)^{n-1}$ points in T^{n-1} by the induction hypothesis. Similarly, p_1 has one variable and therefore at least $\|T\| - d$ nonzero points.

For each of the $|T| - d$ choices for a_n where p_1 is nonzero, the corresponding polynomial p_0 has $(|T| - d)^{n-1}$ nonzero points. Therefore the number of nonzero points of p in T^n is at least $(\|T\| - d)^{n-1}(\|T\| - d) = (\|T\| - d)^n$. □

We mention two important consequences of this theorem. First of all, let T be any subset of $\mathbf{F}$ that has at least $d+1$ elements. Then any nonzero polynomial of degree d has a nonzero point in T^n.

Corollary 4.16 *Let $p(x_1, \ldots, x_n)$ be a polynomial of degree d over $\mathbf{F}$, and $T \subseteq \mathbf{F}$ with $\|T\| > d$. Then $p \not\equiv 0 \iff \exists \boldsymbol{a} \in T^n \;\; p(\boldsymbol{a}) \neq 0$.*

By enlarging T even further, we can achieve that any nonzero polynomial p does not vanish on *most* of the points of T^n. This provides the tool for the probabilistic zero-test.

Corollary 4.17 *Let $p(x_1, \ldots, x_n) \not\equiv 0$ be a polynomial of degree d over $\mathbf{F}$, and let $T \subseteq \mathbf{F}$ with $\|T\| \geq 2nd$. Let $\boldsymbol{r} = (r_1, \ldots, r_n)$ be a random element from T^n. Then $p(\boldsymbol{r}) \neq 0$ with probability at least $1/2$.*

Proof $\mathbf{Pr}[p(\boldsymbol{r}) \neq 0] \geq \left(\frac{\|T\|-d}{\|T\|}\right)^n \geq \left(1 - \frac{1}{2n}\right)^n \geq \frac{1}{2}$. □

There are several ways to amplify the correctness of of the zero-test. The most obvious one is to repeat the test some number of times. We have described this in general in Section 2.3. More specific to the test here, we can enlarge our test domain T and refer to the error estimation given in Corollary 4.17. In both possibilities we need more random bits to obtain an amplification. A completely different zero-test was discovered by Chen and Kao [CK97]: for any multivariate polynomial p over the rationals they give *explicit* (irrational) points where p is nonzero if it is not the zero-function. The idea is simply to approximate one of these points. The only place where randomization is used is to choose one of these (fixed) points at random. The probability amplification is obtained by computing better approximations to the chosen irrational number. Hence, amplification does *not* require more random bits here, but more time.

Now it is easy to solve the equivalence problem for polynomials, at least for $\mathbf{F} = \mathbf{Q}$: given some description of two polynomials p_0 and p_1 over $\mathbf{Q}$. The only requirement we have for the description is that we can efficiently evaluate the polynomials at any given point. Then define $p = p_0 - p_1$ and apply Corollary 4.17 to p with $T = \{1, \dots, 2n\}$.

Corollary 4.18 *The equivalence problem for polynomials over* $\mathbf{Q}$ *is in* **coRP**.

Let us apply these results to solve the equivalence problem for read-once branching programs. Given B_0 and B_1, both in n variables. Let p_{B_0} and p_{B_1} be the multilinear polynomials over $\mathbf{F} = \mathbf{Q}$ obtained by arithmetizing B_0 and B_1 (Theorem 4.12). Hence, p_{B_i} agrees with B_i on $\{0,1\}^n$, for $i = 0, 1$. Now consider the polynomial $p = p_{B_0} - p_{B_1}$. It is zero on $\{0,1\}^n$ iff $B_0 \equiv B_1$.

We apply Corollary 4.16 with $T = \{0,1\}$. Since p_{B_0} and p_{B_1} are multilinear, so is p. Hence the assumption of Corollary 4.16 is fulfilled and we conclude that if p is zero on $\{0,1\}^n$, then it must be zero on $\mathbf{Q}$. Therefore

$$p \equiv 0 \iff B_0 \equiv B_1.$$

Now the claim follows from Corollary 4.18. By Theorem 4.14, an analog statement holds for $(1,+1)$-branching programs (in fact, for $(1,+k)$-branching programs).

Corollary 4.19 *The equivalence problems for read-once and* $(1,+1)$*-branching programs are in* **coRP**.

If we start with polynomials equivalent over $\{0,1\}$, a set of size two, the above technique works only for multilinear polynomials. Polynomials of degree two can already agree on $\{0,1\}$ without being identical. But clearly we can argue along the same lines as above for degree d polynomials that are equivalent over a set of size $d+1$.

If the polynomials are over a *finite* field, say $\mathrm{GF}(q)$, where q is some prime power, we run into the problem that there might not be enough elements for our set T in order to make the above test work. A solution to this problem is to work in the extension field $\mathrm{GF}(q^t)$ instead of $\mathrm{GF}(q)$, where t is the smallest integer such that $q^t \geq 2nd$, so that $t = \lceil \log_q 2nd \rceil$. Then we can define $T = \mathrm{GF}(q^t)$. By Corollary 4.16, when $q > d$, we have that two polynomials over $\mathrm{GF}(q)$ are equivalent if and only if they are equivalent over any extension field.

Also, we can still evaluate a polynomial efficiently at a given point in the extension field: for this, we need an irreducible polynomial $\phi(x) \in$

$\mathrm{GF}(q)[x]$ of degree t. We can cycle through all the $q^{t+1} \leq 2ndq^2 < 2nq^3$ polynomials in $\mathrm{GF}(q)[x]$ of degree t and check irreducibility in polynomial time using the Berlekamp algorithm (see [Ber70], Chapter 6). So we will find an irreducible polynomial $\phi(x)$ in polynomial time. Then $\mathrm{GF}(q^t)$ is isomorphic to $\mathrm{GF}(q)[x]/\phi(x)$. Therefore, knowing $\phi(x)$, we can do all computation needed in polynomial time.

Corollary 4.20 *The equivalence problem for polynomials of degree d over a finite field of size greater than d is in* **coRP**.

The lower bound on the field size is crucial: for small fields the equivalence problem for polynomials is **coNP**-complete [IM83]. To see this, let $F = C_1 \wedge \cdots \wedge C_m$ be a CNF-formula. Let q be some prime power. We construct a polynomial p_F over $\mathrm{GF}(q)$ by arithmetizing F as follows. If clause C_1 has, for example, the form $C_1 = x_1 \vee \overline{x}_2 \vee x_3$, we define the polynomial c_1 as

$$c_1 \;=\; 1-(1-x_1^{q-1})x_2^{q-1}(1-x_3^{q-1}).$$

Note that $a^{q-1} = 1$ for all $a \neq 0$. Therefore it suffices to consider assignments over $\{0,1\}$, and there, c_1 and C_1 coincide.

Analogously we get polynomials $c_2, \ldots, c_m$ for the remaining clauses. Then we define $p_F = \prod_{i=1}^{m} c_i$. It follows that $F \in \textsc{Sat} \iff p_F \not\equiv 0$. The degree of p_F is at most $m(q-1)$.

We remark that the technique can be modified to an equivalence test for polynomials over *infinite fields with finite characteristic* [IM83].

4.2.2 The Isomorphism Problem

We show that there is a two-round interactive proof for the non-isomorphism problem for polynomials over $\mathbf{Q}^n$. The polynomials must be given in such a way that one can easily evaluate them at any given point. For concreteness, we assume that we consider arithmetic expressions, which is what we get from branching programs.

We want to use the protocol for graph non-isomorphism. So let p_0 and p_1 be two polynomials over $\mathbf{Q}$ of degree d, both in n variables. The verifier produces a random isomorphic copy p of a randomly chosen input polynomial. But if he now simply presents p to the prover, we run into the same problem we had with boolean formulas in Chapter 3: the syntactic structure might tell the prover which of the two was selected by the verifier.

A way out would be a normal form for polynomials that is easy to compute. However, such a normal form seems not to be known. Our algorithm

Randomized-Normal-Form from Chapter 3 would work. However, it uses an **NP**-oracle which is too much in our setting now.

The idea to get around this problem is as follows. Instead of trying to manipulate all of p, the verifier just sends a *fingerprint* of p to the prover that is independent of the syntactic structure of p:

a pair $(\boldsymbol{r}, p(\boldsymbol{r}))$, for a randomly chosen point $\boldsymbol{r} \in T^n$,

where T is some appropriate test domain. The prover is now asked to tell which of p_0, p_1 was used to obtain the point $(\boldsymbol{r}, p(\boldsymbol{r}))$. If p_0 and p_1 are isomorphic, then the prover cannot detect this and has to guess. So she will fail with probability $\frac{1}{2}$. On the other hand, if p_0 and p_1 are *not* isomorphic, then the prover has a good chance of detecting the origin of $(\boldsymbol{r}, p(\boldsymbol{r}))$. This is because, by Corollary 4.17, different polynomials can agree on T on at most $\frac{1}{2}$ of the points for $\|T\| \geq 2dn$. That is, the origin of $(\boldsymbol{r}, p(\boldsymbol{r}))$ is unique with high probability. With an additional round of communication the prover can *always* convince the verifier of the *non-isomorphism* of p_0 and p_1. We give the details below.

Theorem 4.21 *The non-isomorphism problem for polynomials of degree d over* $\mathbf{Q}$ *is in* $\mathbf{IP}[4]$.

Proof We give an interactive proof system for the problem in Figure 4.1. The inputs are two polynomials p_0 and p_1 of degree d, both in n variables. Let $T = \{1, \ldots, 2dn\}$.

- [V1:] The verifier randomly picks $i \in \{0,1\}$, a permutation φ, and points $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_k \in T^n$, where $k = \lceil n \log n \rceil + 2$. Then he permutes the variables of p_i according to φ, computes $y_l = p_i \circ \varphi(\boldsymbol{r}_l)$, for $l = 1, \ldots, k$, and sends the set of pairs $R = \{\, (\boldsymbol{r}_l, y_l) \mid l = 1, \ldots, k \,\}$ to the prover.
- [P1:] The prover sends $j \in \{0,1\}$ and a permutation φ' to the verifier.
- [V2:] If $i = j$, then the verifier accepts. If $i \neq j$, the verifier checks whether $p_j \circ \varphi'$ *matches the set* R, that is, whether $p_j \circ \varphi'(\boldsymbol{r}_l) = y_l$, for $l = 1, \ldots, k$. If the test fails, the verifier rejects. Otherwise, he sends φ to the prover.
- [P2:] The prover sends a point $\boldsymbol{r}' \in T^n$ to the verifier.
- [V3:] Finally, the verifier accepts iff $p_i \circ \varphi(\boldsymbol{r}') \neq p_j \circ \varphi'(\boldsymbol{r}')$.

Fig. 4.1: Interactive proof for 1-BPNI.

We show that the above protocol works correctly.

Case 1: $p_0 \not\cong p_1$. We show that there is a prover such that the verifier always accepts.

The prover can cycle through all permutations and check for both, p_0 and p_1, whether it matches with the set R sent by the verifier in step V1. Say that polynomial $p_0 \circ \varphi'$ does so. Then the prover sends $j = 0$ and φ' to the verifier in step P1.

If *no* permutation of polynomial p_1 matches R as well, then i must have been 0 and therefore the verifier will accept in the first round.

On the other hand, if some permutation of polynomial p_1 matches R, then the prover cannot tell which one was used by the verifier. If the prover is lucky, i has anyway been zero and the verifier accepts. On the other hand, if $i \neq j$, then the verifier will send φ to the prover in step V2, because $p_j \circ \varphi'$ matches R. Since $p_j \circ \varphi' \neq p_i \circ \varphi$, these polynomials can agree on at most $\frac{1}{2}$ of the points of T^n by Corollary 4.17. Therefore, the prover can find a point $\boldsymbol{r}' \in T^n$ such that $p_j \circ \varphi'(\boldsymbol{r}') \neq p_i \circ \varphi(\boldsymbol{r}')$, and send it to the verifier in step P2 who will accept in step V3. In summary, the verifier accepts with probability one.

Case 2: $p_0 \cong p_1$. We show that for any prover, the verifier accepts with probability at most $\frac{3}{4}$. By executing the protocol several times in parallel, the acceptance probability can be made exponentially small.

The prover will always find permutations of p_0 *and* p_1 that match the set R sent by the verifier in step V1. Therefore, with respect to the test $i = j$ made by the verifier, the best the prover can do is to guess j randomly. This will make the verifier accept with probability $\frac{1}{2}$ in step V2. However, the prover can improve her chances by the condition checked in the second round by the verifier: fix i and φ chosen by the verifier, say $i = 0$. Then there might exist a permutation φ' such that $p_1 \circ \varphi'$ matches R, but $p_0 \circ \varphi \neq p_1 \circ \varphi'$. Now the prover can choose a point $\boldsymbol{r}'$ such that $p_0 \circ \varphi(\boldsymbol{r}') \neq p_1 \circ \varphi'(\boldsymbol{r}')$, and make the verifier accept in step V3 by sending $j = 1$, φ', and $\boldsymbol{r}'$. We will give an upper bound on the probability of this event.

By Corollary 4.17, for any φ' such that $p_0 \circ \varphi \neq p_1 \circ \varphi'$ we have

$$\mathbf{Pr}[p_0 \circ \varphi(\boldsymbol{r}) = p_1 \circ \varphi'(\boldsymbol{r})] \quad < \quad 1/2,$$

for a randomly chosen $\boldsymbol{r} \in T^n$. Since points $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_k \in T^n$ are chosen independently and uniformly at random from T^n, we have

$$\mathbf{Pr}[p_1 \circ \varphi' \text{ matches } R] \quad < \quad 2^{-k},$$

Therefore, considering all such φ', we get that

$$\mathbf{Pr}[\exists \varphi' \ (p_0 \circ \varphi \neq p_1 \circ \varphi' \text{ and } p_1 \circ \varphi' \text{ matches } R] \quad < \quad n! \, 2^{-k} \ \leq \ 1/4$$

by our choice of k. We conclude that the probability that any of the conditions tested by the verifier is satisfied is bounded by $\frac{1}{2} + \frac{1}{4} = \frac{3}{4}$. That is, the verifier accepts with probability at most $\frac{3}{4}$, irrespective of the prover. $\square$

We can directly obtain a *one round* interactive proof by choosing T large, for example $T = \{1, \ldots, 2dnn!\}$. Then, in case $p_0 \not\cong p_1$, the prover can always find a point $\boldsymbol{r}'$ as above without knowing φ. Hence, she can already send $\boldsymbol{r}'$ in the first round to the verifier, who can then make all the tests. However, then we have another difficulty: when T has exponential size, the values of the polynomials might be up to double exponential. In this case, the polynomial-time verifier can no longer deal with such numbers. We will show in the next section how the verifier can still manage this task.

Transforming the private coin proof system into public coin as explained in Section 2.5, we get that the non-isomorphism of polynomials can be decided in $\mathbf{AM} = \mathbf{BP} \cdot \mathbf{NP}$.

Corollary 4.22 *The non-isomorphism problem for polynomials of degree d over $\mathbf{Q}$ is in $\mathbf{BP} \cdot \mathbf{NP}$. Therefore the corresponding isomorphism problem cannot be $\mathbf{NP}$-hard unless $\mathbf{PH} = \Sigma_2\mathbf{P}$.*

As for Corollary 4.20, we can apply the technique to large enough finite fields.

Corollary 4.23 *The non-isomorphism problem for polynomials of degree d over a finite field of size greater than d is in $\mathbf{BP} \cdot \mathbf{NP}$. Therefore the corresponding isomorphism problem cannot be $\mathbf{NP}$-hard unless $\mathbf{PH} = \Sigma_2\mathbf{P}$.*

In the same way as we applied the zero-test for polynomials to the equivalence problem for read-once branching programs, we can now apply the isomorphism test to them.

Corollary 4.24 *The non-isomorphism problem for read-once branching programs is in $\mathbf{BP} \cdot \mathbf{NP}$. Therefore the corresponding isomorphism problem cannot be $\mathbf{NP}$-hard unless $\mathbf{PH} = \Sigma_2\mathbf{P}$.*

Our interactive proof system was motivated by the one for GNI. However, it is more general now in the sense that it can be used to solve GNI as a special case. Namely, one can assign a polynomial to a graph such that non-isomorphic graphs are mapped to non-isomorphic polynomials: let $G = (V, E)$ be a graph, where $V = \{1, \ldots, n\}$. We take one variable x_i for each node $i \in V$. Define

$$\begin{aligned} e_i(x_1, \ldots, x_n) &= x_i^2 \prod_{(i,j) \in E} x_j, \text{ and} \\ p_G(x_1, \ldots, x_n) &= \sum_{i=1}^{n} e_i(x_1, \ldots, x_n). \end{aligned}$$

For graphs G_0, G_1 we have that $G_0 \cong G_1 \Longleftrightarrow p_{G_0} \cong p_{G_1}$. Therefore we obtain again the result about GI from Corollary 4.22.

4.2.3 Extension to Arithmetic Circuits

An *arithmetic circuit over a field* $\mathbf{F}$ is a circuit, where the inputs are field elements and the (fan-in two) gates perform the field operations $+$, $-$, and $\times$. (We could also allow division as long as a circuit guarantees to not divide by zero on any input.) *Circuit size* and *depth* are defined as usual.

Clearly, any polynomial p given as an arithmetic expression as in the last section can be written as an arithmetic circuit, which is therefore the more general model.

Ibarra and Moran [IM83] considered the equivalence problem for arithmetic circuits (called *straight-line programs* there) over $\mathbf{Q}$. They give probabilistic polynomial-time algorithm. If a circuit C has n input variables $x_1, \ldots, x_n$, then C computes a multivariate polynomial p_C over $\mathbf{Q}$. If C has depth d then p_C has degree at most 2^d, since the worst case is that the circuit successively squares the input. Therefore, to obtain a zero-test for p_C, we have to choose T in Corollary 4.17 as $T = \{1, \ldots, 2n2^d\}$, in order to detect a nonzero point with probability more than $\frac{1}{2}$ at a random point from T^n.

However, we do not have a polynomial-time procedure yet because the function values of p_C on T^n could be as large as $(2n2^d)^{n2^d} \leq 2^{N2^N}$ for $N = nd$. Represented in binary, such numbers would be exponentially long. Instead, we evaluate p_C *modulo smaller numbers*, namely from $M = \{1, \ldots, 2^{2N}\}$. (For a zero-test, we can assume that all coefficients are integers so that the function values of p_C are integers too.) $p_C \pmod{m}$ might have more zeros than p_C, however, not too many:

Lemma 4.25 *For any $y \leq 2^{N2^N}$ and a randomly chosen $m \in M$, we have $y \not\equiv 0 \pmod{m}$ with probability at least $\frac{1}{3N}$, for large enough N.*

Proof Any $y \leq 2^{N2^N}$ has at most $N2^N$ prime divisors. By the prime number theorem, there are more than $\frac{2^{2N}}{2N}$ primes in M for large enough N. Therefore M contains at least $\frac{2^{2N}}{2N} - N2^N$ primes that do not divide y. Hence, for m randomly chosen from M, we have

$$\mathbf{Pr}[y \not\equiv 0 \pmod{m}] \geq \frac{\frac{2^{2N}}{2N} - N2^N}{2^{2N}} = \frac{1}{2N} - \frac{N}{2^N} \geq \frac{1}{3N}$$

□

The probabilistic zero-test now works as follows.

Corollary 4.26 *Let $p(x_1, \dots, x_n)$ be a nonzero polynomial of degree 2^d over $\mathbf{Q}$, $T = \{1, \dots, 2n2^d\}$ and $M = \{1, \dots, 2^{2N}\}$, where $N = nd$. Choose $\boldsymbol{r}_1, \dots, \boldsymbol{r}_{6N}$ from T^n and $m_1, \dots, m_{6N}$ from M independently at random. Then $p(\boldsymbol{r}_i) \not\equiv 0 \pmod{m_i}$, for some i, with probability at least $\frac{1}{2}$.*

Proof By Corollary 4.17 and Lemma 4.25, $\mathbf{Pr}[p(\boldsymbol{r}_i) \not\equiv 0 \pmod{m_i}] \geq \frac{1}{2}\frac{1}{3N}$, for any pair $\boldsymbol{r}_i$, m_i. Therefore, $\mathbf{Pr}[p(\boldsymbol{r}_i) \equiv 0 \pmod{m_i}$ for all i $] \leq \left(1 - \frac{1}{6N}\right)^{6N} \leq \frac{1}{2}$. □

Corollary 4.27 *The equivalence problem for arithmetic circuits over* $\mathbf{Q}$ *is in* **coRP**.

Next we extend the protocol for polynomials in Section 4.2.2 to an interactive proof to decide the non-isomorphism of two arithmetic circuits over $\mathbf{Q}$.

Let C_0, C_1 be two arithmetic circuits with n inputs that are of depth d. We take the protocol from the previous section and modify it according to Corollary 4.26. The details are shown in Figure 4.2. Let $T = \{1, \dots, 2n2^d\}$ and $M = \{1, \dots, 2^{2N}\}$, where $N = nd$.

- [V1:] the verifier starts by randomly choosing $i \in \{0, 1\}$ and a permutation φ as before, and now $6Nk$ points $\boldsymbol{r}_1, \dots, \boldsymbol{r}_{6Nk} \in T^n$, where $k = \lceil n \log n \rceil + 2$, and for each point $\boldsymbol{r}_l$ a number $m_l \in M$. Then the verifier computes $y_l = p_{C_i} \circ \varphi(\boldsymbol{r}_l) \bmod m_l$, for $l = 1, \dots, 6Nk$, and sends the set of triples $R = \{ (\boldsymbol{r}_l, y_l, m_l) \mid l = 1, \dots, 6Nk \}$ to the prover.
- [P1:] The prover sends $j \in \{0, 1\}$ and a permutation φ' to the verifier.
- [V2:] If $i = j$, then the verifier accepts. If $i \neq j$, the verifier checks whether $p_{C_j} \circ \varphi'$ *matches the set* R, that is, whether $y_l = p_{C_i} \circ \varphi'(\boldsymbol{r}_l) \bmod m_l$, for $l = 1, \dots, 6Nk$. If the test fails, the verifier rejects. Otherwise, he sends φ to the prover.
- [P2:] The prover sends a point $\boldsymbol{r}' \in T^n$ and $m' \in M$ to the verifier.
- [V3:] Finally, the verifier accepts iff $p_{C_i} \circ \varphi(\boldsymbol{r}') \not\equiv p_{C_j} \circ \varphi'(\boldsymbol{r}') \pmod{m'}$.

Fig. 4.2: Interactive proof for the non-isomorphism of two arithmetic circuits.

Combining the argument in the previous section with Corollary 4.26, the verifier will accept two isomorphic arithmetic circuits with probability at most $\frac{3}{4}$. Note that the prover in step P2 has to prove to the verifier that two numbers *differ*. Therefore the computation modulo some number does not work in favor of the prover in that case. Two non-isomorphic arithmetic circuits are still accepted with probability one.

Theorem 4.28 *The non-isomorphism problem for arithmetic circuits over* $\mathbf{Q}$ *is in* $\mathbf{BP} \cdot \mathbf{NP}$. *Therefore the corresponding isomorphism problem cannot be* $\mathbf{NP}$*-hard unless* $\mathbf{PH} = \Sigma_2\mathbf{P}$.

If the arithmetic circuits are over a *finite* field, say $\mathrm{GF}(q)$, where q is some prime power, we can again use the trick with the extension field as in Corollary 4.20 (with $\mathrm{GF}(q^t)$, where t is the smallest integer such that $q^t \geq 2n2^d$, so that $t = \lceil \log_q 2n2^d \rceil$).

Corollary 4.29 *The non-isomorphism problem for arithmetic circuits of depth d over a finite field of size more than 2^d is in* $\mathbf{BP} \cdot \mathbf{NP}$. *Therefore the corresponding isomorphism problem cannot be* $\mathbf{NP}$*-hard unless* $\mathbf{PH} = \Sigma_2\mathbf{P}$.

4.3 Ordered Branching Programs

Of particular interest in applications are branching programs where the variables are read in a certain fixed order [Bry86], and extensions to several layers.

Definition 4.30 *An* ordered branching program *(also called* ordered binary decision diagram, OBDD *for short) is a read-once branching program such that there is a permutation π on $\{1, \dots, n\}$ such that if an edge leads from a node labelled x_i to a node labelled x_j, then $\pi(i) < \pi(j)$.*

We give some examples. Ordered branching programs can compute any symmetric function in n variables within size $O(n^2)$. This is because ordered branching programs can *count*. We show how to construct a **unary counter** C^u that counts the number of ones in the input: imagine $n + 1$ vertical lines numbered $0, 1, \dots, n$. These lines serve as counters for the number of ones we have seen while reading the input. That is, the initial node is put on line 0. Let u be a node on line k, then node $e(u, 0)$ is put on line k too, whereas node $e(u, 1)$ is put on line $k + 1$. On some input $\boldsymbol{a}$, if we reach the leaf on line k, then there are precisely k ones in $\boldsymbol{a}$.

It remains to define the leaves as accepting or rejecting according to the symmetric function f in order to get a branching program that computes f.

Using the same idea, we can also construct a **binary counter** C^b, where the input is interpreted as a binary number. Here we use 2^n counters and, if we see a one at position j in the input, the 1-edge jumps to the counter in distance 2^j. Hence, C^b can be realized by a ordered branching program of size 2^{n+1}.

The binary counter can only be applied for small numbers in order to get a small binary counter. For larger numbers it often suffices to count modulo some small number m. A **binary counter modulo** m, $C^{b,m}$, can be easily obtained by modifying the above binary counter: we use only m counters and do the counting via the 1-edges modulo m. The size of the resulting branching program is bounded by mn. In particular, we get a linear-size ordered branching program for the parity-function, $C^{b,2}$.

4.3.1 Minimization and Equivalence

Every boolean function $f(x_1, \ldots, x_n)$ can be represented by an ordered branching program of exponential size (for any order). To see this consider the *decision tree* for $f(x_1, \ldots, x_n)$. This is a complete binary tree of depth n, i.e., with 2^n leaves. Label nodes with the variables as follows: the root is labelled by x_1 and the nodes in distance k from the root are labelled by x_{k+1} for $k = 1, \ldots, n-1$. The edges are directed downwards and labelled by 0 and 1 respectively. A decision tree has 2^n leaves, whereas for a branching one accepting and one rejecting node at the end suffice. All paths that correspond to assignments where f has value 1 go the accepting node, all others lead to the rejecting node. The size of this program is exponential, $2^n + 1$.

Sometimes it is possible to compress the tree. Here are two rules that, when applied to an ordered branching program, lead to an equivalent smaller one with the same order.

Rule 1 (Eliminate Duplicate Nodes) *Let u and v be two nodes that are labelled by the same variable such that $e(u,0) = e(v,0)$ and $e(u,1) = e(v,1)$. Then eliminate one of them, say v, and redirect all edges going to v to u.*

Rule 2 (Eliminate Redundant Nodes) *If $e(u,0) = e(u,1) = v$ then eliminate u and redirect all edges going to u to v.*

The rules can in fact be applied to arbitrary branching programs (not just ordered ones). A branching program with no duplicate and no redundant nodes is in *reduced form*. Bryant [Bry86] showed that for ordered branching programs, the reduced form is *minimal* with respect to that order. Moreover, it is *unique*.

Theorem 4.31 *Let $f(x_1, \dots, x_n)$ be a boolean function and π be some order on the variables of f. There is a (up to isomorphism of the underlying graph) unique minimal ordered branching program for f with order π.*

Moreover, given any ordered branching program B that computes f, the minimal program for f can be computed in polynomial time in $|B|$.

Proof We assume w.l.o.g. that π is the identity permutation.

Let $\boldsymbol{a} = (a_1, \dots, a_k)$ be a partial assignment to the variables of f. Then $f_{\boldsymbol{a}}$ denotes the following *sub-function of f*:

$$f_{\boldsymbol{a}}(x_{k+1}, \dots, x_n) \quad = \quad f(a_1, \dots, a_k, x_{k+1}, \dots, x_n).$$

Let $\boldsymbol{b} = (b_1, \dots, b_l)$, and assume that $l \leq k$. We say that $f_{\boldsymbol{a}}$ *is different from* $f_{\boldsymbol{b}}$, $f_{\boldsymbol{a}} \neq f_{\boldsymbol{b}}$ for short, if there is an extension $\boldsymbol{b'}$ to $\boldsymbol{b}$ such that $|\boldsymbol{bb'}| = |\boldsymbol{a}|$ and $f_{\boldsymbol{a}} \neq f_{\boldsymbol{bb'}}$.

Let B be an ordered branching program computing f. For each node u of B let B_u be the sub-branching program of B with initial node u. Observe that each such B_u computes a sub-function $f_{\boldsymbol{a}}$ of f, where $\boldsymbol{a}$ is a partial assignment that leads to node u in B.

Suppose f has s pairwise different sub-functions in the above sense. Then B must have at least s nodes in order to distinguish between these sub-functions. On the other hand, Rule 1 and 2 precisely combine such programs that compute the same sub-functions. Hence, when no more rule is applicable, we are left with pairwise different sub-functions only and thus with s nodes. The resulting branching program is unique because the construction does not depend on the specific initial assignments that lead to a node but merely on the sub-function associated with a node. $\square$

Sieling and Wegener[SW93] showed that the reduction procedure can actually be done in *linear* time.

Observe the similarity of the argument in the last proof with the Myhill/Nerode construction for finite automata. This is not accidentally: one can easily transform an ordered branching program B into a finite automaton that accepts precisely the strings of length n that are accepted by B (maybe with adjusting the order if it is not the identity): include additional *redundant nodes* in the sense of Rule 2 such that all variables are tested on every path. Put self-loops on the rejecting node and both edges from the accepting node lead to the rejecting node.

Let A be this finite automaton. The initial state of A is the initial node of B, the only accepting state of A is the accepting node of B. Then A accepts only strings of length n and of those only the ones accepted by B. Hence the minimum DFA constructed from A leaves us with applying only Rule 2 in order to get back the minimum ordered branching program.

An efficiently computable normal form gives in particular an efficient equivalence test.

Corollary 4.32 *The equivalence problem for ordered branching programs that obey the same order is in* **P**.

An obvious question comes to mind when considering Corollary 4.32: what about the equivalence problem when the two ordered branching programs have different orders? The answer is: this can be solved efficiently as well. The idea is to to change the order of one of the branching programs so that they get the same order [MS94]. Then we can apply Corollary 4.32.

To change the order of a ordered branching program, we use the technique from Theorem 4.31. Potentially, this can lead to much larger branching programs and this can actually happen (see Section 4.3.3). Therefore we measure the running-time in the length of the input *and* the output.

Theorem 4.33 *Given an ordered branching program B and some order π, the minimal ordered branching program B' that is equivalent to B and has order π can be constructed in polynomial time in $|B|$ and $|B'|$.*

Proof Assume w.l.o.g. that π is the identity permutation and let $x_1, \ldots, x_n$ be the variables of B.

The initial step to construct B' is to put a root node and label it with x_1. Now assume that we have constructed B' up to nodes labelled $x_1, \ldots, x_{k-1}$ for some $k \leq n$ and let $u_1, \ldots, u_m$ be the nodes that don't have successors yet. Create $2m$ new nodes v_1^0, $v_1^1, \ldots, v_m^0$, v_m^1 and connect the b-edge of u_i with v_i^b, i.e., define $e(u_i, b) = v_i^b$. The new nodes are labelled with variable x_k.

Some of the new nodes might be duplicate nodes. Therefore we want to apply Rule 1 in order to eliminate them. But Rule 1 simply looks at the edges of the v-nodes which are not there yet. Instead we use program B to find out the duplicate nodes: each v-node is reached by some assignment $a_1, \ldots, a_{k-1}$ to the variables $x_1, \ldots, x_{k-1}$.

We modify B to a program in variables $x_k, \ldots, x_n$ by fixing the values of $x_1, \ldots, x_{k-1}$ to $a_1, \ldots, a_{k-1}$ in B, respectively. For example, to fix variable x_1 to a_1 in B, we construct $B_{x_1=a_1}$ as follows: eliminate all nodes w labelled x_1 in B and redirect edges to w to the node $e(w, a_1)$. Then we continue to fix $x_2, \ldots, x_{k-1}$ in the same way.

It follows that for each of the $2m$ v-nodes we can construct an ordered branching program from B that computes the sub-function of B at that v-node (i.e. in variables $x_k, \ldots, x_n$). Since all these branching programs obey the same order, we can use Corollary 4.32 to (pairwise) check their equivalence. If two v-nodes compute the same sub-function then one of them is duplicate and we can combine them according to Rule 1. Having eliminated all duplicate nodes, we apply Rule 2, if possible.

After the last step, for $k = n$, the result B' is the minimal ordered branching program with identity order. □

Savický and Wegener [SW97] give a linear-time algorithm for changing the order (linear in $|B| + |B'|$).

Now it is easy to solve the equivalence problem for ordered branching programs with different orders: suppose B_0 and B_1 are already minimized. Start changing the order of B_0 into the order of B_1. If $B_0 \equiv B_1$, then our above procedure will directly construct B_1 in a top-down manner. Therefore we can stop the procedure as soon as we detect some disagreement with B_1. (Note that in general we cannot first transform B_0 completely and then compare the outcome with B_1 since the intermediate result could be large.)

Corollary 4.34 *The equivalence problem for ordered branching programs is in* **P**.

It follows that the isomorphism problem for ordered branching program is in **NP**. By Corollary 4.24 it is unlikely to be **NP**-complete.

Corollary 4.35 *The isomorphism problem for ordered branching programs is not* **NP***-complete, unless* $\mathbf{PH} = \Sigma_2\mathbf{P}$.

Note that there is a simpler proof for the latter corollary. This is because by Theorem 4.31 ordered branching programs have a normal form. So one can use the normal form in the interactive proof systems for the complementary problem instead of the polynomials as in the case of read-once branching programs.

Fortune, Hopcroft, and Schmidt [FHS78] extended Corollary 4.34. They showed that it suffices for one of the branching programs to be read-once.

Theorem 4.36 *The equivalence of an ordered branching program with a read-once branching program can be checked in polynomial time.*

Proof Let B be an ordered branching program and C be a read-once branching program, both in n variables.

We reduce the problem whether $B \equiv C$ to two smaller ones: let x_1 be the variable at the initial node u of C and $u_b = e(u, b)$ the node reached along the b-edge from u, for $b \in \{0,1\}$. By $C_{x_1=b}$ we denote the subprogram of C that has initial node u_b. $B_{x_1=b}$ is obtained from B by fixing x_1 to b as described in the proof of Theorem 4.33. Then

$$B \equiv C \iff B_{x_1=0} \equiv C_{x_1=0} \text{ and } B_{x_1=1} \equiv C_{x_1=1}.$$

Our algorithm is simply to repeat this process with the new pairs of branching programs we obtain that way. That is, with the initial node of $C_{x_1=b}$ for the pair $(B_{x_1=b}, C_{x_1=b})$, for $b = 0, 1$. Since this process would lead to exponentially many pairs, we have to reduce the number of pairs. This is done as follows. Different initial assignments to the variables can lead to the same node, say v, in C. Suppose there are l paths $p_1, \dots, p_l$ from the initial node to v in C. Each of them creates a new equivalence problem (B_{p_i}, C_{p_i}) at some point in our procedure. Since all paths reach the same node in C, all programs C_{p_i} are equivalent. Therefore, all programs B_{p_i} have to be equivalent too. But this we can check by Corollary 4.32, since all B_{p_i}'s have the same order. If all B_{p_i}'s turn out to be equivalent, then it is enough to keep just one from the l pairs, say (B_{p_1}, C_{p_1}), to be checked further.

Let C have size m. As soon as the number of pairs we have exceeds m there must be some pairs with the same second component. Then we can reduce the number of pairs as described above. Hence we never have more than $2m$ pairs at any moment of this algorithm.

Finally, when we reach the accepting or rejecting node in C, we can again apply Corollary 4.32 to check the equivalence of such a pair, because then both components are ordered.

Hence our algorithm determines the equivalence of B and C in polynomial time. □

As already mentioned at the end of Section 4.1, no polynomial-time algorithm is known for the equivalence problem for read-once branching programs. Nevertheless, we have an efficient randomized algorithm for it (Corollary 4.19).

4.3.2 Boolean Combinations

One reason that ordered branching programs have turned out to be useful in applications is the efficient equivalence test. Another reason is that there are fast algorithms to combine two such programs according to boolean operations [Bry86]. For example, given two ordered branching programs B_0 and B_1 that obey the same order, then one can construct in polynomial time a new one that computes $B_0 \wedge B_1$, i.e., the conjunction of the result of B_0 and B_1 on an input.

The key idea is again borrowed from finite automata: build a *cross-product* $B_0 \times B_1$ of B_0 and B_1 as follows.

First introduce redundant nodes in the sense of Rule 2 into B_0 and B_1 such that in both, on each path from the initial to a final node *all* variables appear exactly once. For simplicity of notation, let us assume that we have the identity order. Then the nodes at distance l from the root form a *level*: they are all labelled with variable x_{l+1} and edges are only going to the next level, $l+1$, for $l = 0, \ldots, n-1$. Let U_l be the set of nodes of B_0 in level l and V_l be the set of nodes of B_1 in level l.

Define branching program B as follows. The nodes at level l are $U_l \times V_l$, all with label x_{l+1}. The edges are defined so that B_0 and B_1 are simulated in parallel. That is, the 0-edge of node $(u, v) \in U_l \times V_l$ leads to node $(e(u, 0), e(v, 0)) \in U_{l+1} \times V_{l+1}$, and similar for the 1-edge.

From the accepting and rejecting nodes of B_0 and B_1 we obtain four nodes in B in the last level. These nodes we have to define as either accepting or rejecting according to the boolean operation we want to combine B_0 and B_1. For example, if we want to do conjunction, we take (accept,accept) as the accepting node and the other three as rejecting nodes. Similar for disjunction, (reject,reject) is the only rejecting node, the other three accept.

The size of B is clearly bounded by $|B_0||B_1|$.

Since branching programs (even general ones) can be complemented (exchange accepting and rejecting nodes), we can combine ordered branching programs according to any boolean operation. For more sophisticated algorithms see [Bry86, Bry92].

Lemma 4.37 *Ordered branching programs (with the same order) can be combined in polynomial time according to any boolean operation.*

Now we can describe an application of ordered branching programs. In various contexts the problem arises to check the equivalence of circuits, a hard problem in general, as we already know. The approach used with some success is to transform a given circuit into an equivalent ordered branching program. The gates of a circuit perform boolean operations on sub-circuits. Starting at the input level, we successively build branching programs for these sub-circuits and then combine them according to a gate by applying Lemma 4.37. Each time we get a new branching program, we will reduce it to minimum size by invoking Theorem 4.31. (In practice, these two steps are combined into one algorithm so that the outcome of a combination of two branching programs is already minimal.) Since the size of the branching programs can square with each application of Lemma 4.37, we might end up with an exponential size program for a given circuit. But if the branching programs stay small, we have an efficient method of testing the equivalence of two given circuit.

But for many interesting and naturally occurring circuits the approach does not work. The main drawback of ordered branching programs is there limited computational power.

4.3.3 The Computational Power

We give an example of a function f that has small OBDDs for one variable order, but exponential size OBDDs for another order. Define

$$f(x_1, \dots, x_n, y_1, \dots, y_n) \quad = \quad x_1 y_1 \vee \cdots \vee x_n y_n.$$

If we use the construction from Lemma 4.56 we get an ordered branching program of size $2n$ with variable order $x_1 < y_1 < \cdots < x_n < y_n$.

Suppose we have chosen variable order $x_1 < \cdots < x_n < y_1 < \cdots < y_n$. So we have to read all the x_i's before we see y_1 and are able to evaluate the first term $x_1 y_1$. An obvious solution is an OBDD that has a complete binary tree at the top where all the x-variables are read. Then, reading the y-variables, one can check whether $x_i = y_i = 1$ for some i. The size of this OBDD is bounded by $n2^n$, and is definitely larger than 2^n, i.e., it has exponential size.

Intuitively, it seems as there is no way around to store all possible values for variables $x_1, \dots, x_n$ as we did in the above construction. The way to formalize this intuition is indicated in the proof of Theorem 4.31. There we argued that the minimal branching program for a function f has precisely as many nodes as f has pairwise different sub-functions. So let $\boldsymbol{a} \neq \boldsymbol{b} \in \{0,1\}^n$ be two assignments for the variables $x_1, \dots, x_n$. W.l.o.g. suppose they differ at x_1, and let $a_1 = 1$ and $b_1 = 0$. Then

$$f_{\boldsymbol{a}}(1, 0, \dots, 0) = 1 \neq 0 = f_{\boldsymbol{b}}(0, 0, \dots, 0),$$

so that sub-functions $f_{\boldsymbol{a}}$ and $f_{\boldsymbol{b}}$ are different. Therefore, f has at least 2^n pairwise different sub-functions, and hence, the size of any ordered branching program that computes f with respect to this order must have size at least 2^n.

The situation is similar for integer addition [Bry92]: on input of two integers $x = x_{n-1} \cdots x_0$ and $y = y_{n-1} \cdots y_0$ compute the k-th bit of the sum $x+y$. There are linear-size ordered branching program for order $x_0 < y_0 < \cdots < x_{n-1} < y_{n-1}$, but the order $x_0 < \cdots < x_{n-1} < y_0 < \cdots < y_{n-1}$ requires exponential size. The argument is almost the same as above.

It would therefore be very useful to have an algorithm that computes the optimal variable order for a given function to get small ordered branching programs for that function. In terms of a decision problem, we are interested in the problem Optimal-OBDD: given an ordered branching program B and a number s, decide whether there exists an ordered branching program equivalent to B with size at most s.

Optimal-OBDD is in **NP** because we can guess an order π, transform B into that order by Theorem 4.33, and see whether we get no more than s nodes. Bollig and Wegener [BW96] showed that Optimal-OBDD is in fact **NP**-complete. Therefore we cannot hope to determine the optimal variable order efficiently. There exist several heuristic algorithms for the problem (see [BW96] for references).

There are many functions that have exponential size, $2^{\Omega(n)}$, ordered branching programs for *any* variable order. The lower bound proofs essentially follow the idea to show that a function has exponentially many sub-functions, not just for a specific order as in our example above, but for all orders of the variables. Below we give some examples of such functions. (This is not an exhaustive list, further examples can be found in the literature.) In the comments to our examples we already refer to branching program models defined in the following sections.

Hidden-Weighted-Bit
Given $x = x_1 \cdots x_n \in \{0,1\}^n$, define the *weight of* x, as the number of 1's in x, i.e., $w = w(x) = \sum_{i=1}^{n} x_i$. Then the output is x_w if $w > 0$, and 0 otherwise.

Comment. Shown by Bryant [Bry91]. Can be computed by read-once branching programs of size $O(n^2)$ (see Section 4.1), by 2-OBDDs of size $O(n^2)$, and also by *nondeterministic* ordered branching programs of size $O(n^3)$ [NJFSV96] (see Section 4.4).

Indirect-Storage-Access
Given $x = x_0 \cdots x_{m-1} \in \{0,1\}^m$ and $y = y_0 \cdots y_{k-1} \in \{0,1\}^k$, where $m = 2^k$ and hence $n = 2^k + k$. Interpret y as an integer that points to the y-th block z of k bits in x. That is $z = x_{yk} \cdots y_{(y+1)k-1}$. Interpret z again as integer. Then the output is the bit x_z. In case that $y \geq m/k$, the output is defined to be 0.

Comment. Shown by Breitbart, Hunt, and Rosenkrantz [BHR95]. Can be computed by read-once branching programs of size $O(n^2)$ [BHR95] and by *nondeterministic* ordered branching programs of size $O(n^4)$ (see Section 4.4).

Permutation-Matrix
Given a $n \times n$ 0-1-matrix X, decide whether it is a *permutation matrix*. That is, whether there is exactly one 1 in each row and column of X.

Comment. Shown by Krause [Kra91]. The lower bound is extended independently by Jukna [Juk89] and Krause, Meinel, and Waack [KMW91] to *nondeterministic* read-once branching programs, and by Krause [Kra91] and Ponzio [Pon95] to k-OBDDs, for all $k \geq 1$. Can be computed by 2-IBDDs of size $O(n^2)$ (see below) and by bounded-error probabilistic OBDDs [Sau98] (see Section 4.6). The complement of the problem can be computed by nondeterministic OBDDs of polynomial size.

Clique
Given a graph G with n nodes and $k \in \{1, \ldots, n\}$. Determine whether G has a k-clique.

Comment. Shown by Wegener [Weg88], even for read-once branching programs. The lower bound is extended to *nondeterministic* read-once branching programs by Borodin, Razborov, and Smolensky [BRS93]. The problem is **NP**-complete.

Clique-Only
Given a graph G with n nodes and $k \in \{1, \ldots, n\}$. Determine whether G has a k-clique *and* no other edges than the ones in the clique.

Comment. Shown by Žák [Ž84]. The lower bound is extended to *nondeterministic* read-once branching programs by Borodin, Razborov, and Smolensky [BRS93]. Can be computed by read-twice branching programs [Weg88] of size $O(n^3)$ and and by polynomial size bounded-error probabilistic OBDDs [AT98] (see Section 4.6). The complement of the problem can be computed by nondeterministic read-once branching programs of polynomial size.

Parity-of-Triangles
Given a graph with n nodes. Determine whether it has a an odd number of triangles.

Comment. Shown by Babai *et al.* [BHST87] for read-once branching programs. In fact, the lower bound is $2^{\Omega(n^2)}$. See also [SS93]. Can be computed by *parity* ordered branching programs of size $O(n^3)$ [GM96] (see Section 4.5).

Multiplication
Given two binary numbers $x, y \in \{0,1\}^n$ and $k \in \{0, \ldots, 2n-1\}$. Compute the k-th bit of their product. That is, if $xy = z = z_{2n-1} \cdots z_0$, the output is z_k.

Comment. Shown by Bryant [Bry91]. The lower bound holds even for nondeterministic k-OBDDs and $\oplus$-OBDDs (see Ponzio [Pon95]). The lower bound is extended to read-once branching programs by Ponzio [Pon95]. Ablayev and Karpinski [AK98] show that bounded-error probabilistic OBDDs require size $2^{n/\log n}$. The only known upper bounds are (unrestricted) polynomial-size branching programs.

In the corresponding *verification problem* there are given x, y, and z and one has to verify that $xy = z$. Jukna [Juk95] showed that this requires exponential-size read-k branching programs. It can be solved by polynomial-size bounded-error probabilistic OBDDs [AT98] (see Section 4.6).

From a practical point of view it is of great interest whether one can find some less restrictive model in order to be able to compute more functions within small size, but, at the same time, to keep all the nice properties ordered branching programs have.

Consider, for example, read-once branching programs. Although Hidden-Weighted-Bit and Indirect-Storage-Access can be computed by polynomial size read-once branching programs, the other functions in the above list require exponential size read-once branching programs.

Moreover, one cannot combine read-once branching programs by boolean operations such that the result stays small. An example can be derived from the Permutation-Matrix problem: for a matrix X, define function $R(X) = 1$, if there is precisely one 1 in every row, and 0 otherwise. $C(X)$ is defined analogously for the columns of X. Then we have

$$\text{Permutation-Matrix}(X) \quad = \quad R(X) \wedge C(X).$$

$R(X)$ and $C(X)$ can be computed by ordered branching programs of size $O(n^2)$ (with different orders) but their conjunction needs exponential size read-once branching programs. As an upper bound, this representation of Permutation-Matrix gives a 2-IBDD (see definition in Section 4.3.4 below) of size $O(n^2)$.

Finally we compare branching programs with small depth circuits. Recall that $\mathbf{AC}^0$ is the class of functions computed by polynomial-size constant-depth circuits with unbounded fan-in $\wedge$- and $\vee$-gates. Predicates $R(X)$ and $C(X)$ are easily seen to be in $\mathbf{AC}^0$, and therefore also Permutation-Matrix is in $\mathbf{AC}^0$. Hence there are functions in $\mathbf{AC}^0$ that *cannot* be computed by (nondeterministic) polynomial-size read-once branching programs.

On the other hand, the parity-function can be computed by linear-size ordered branching programs, but requires exponential-size $\mathbf{AC}^0$-type circuits. We conclude that these branching programs are incomparable with $\mathbf{AC}^0$.

4.3.4 Generalized Models

In this section we consider several generalizations of ordered branching programs that relax the order restriction in various ways.

Indexed Branching Programs

The first extension of ordered branching programs that we consider allows to concatenate several such programs.

Definition 4.38 *A* read-k-times ordered branching program *or* k-OBDD *is a read-k-times branching program that can be partitioned into k layers. Each layer obeys the same variable order.*

A read-k-times indexed branching program *or* k-IBDD *is defined as a k-OBDD but different layers may have different variable orders.*

In Theorem 4.10 we showed that the satisfiability problem for (general) branching programs is **NP**-complete. The branching program we constructed in that proof consists of two ordered layers, with different orders in each layer however. Hence, this is a 2-IBDD.

Corollary 4.39 *The satisfiability problem for* 2*-IBDDs is* **NP***-complete.*

If we insist on the same order in every layer, i.e. when we consider k-OBDDs, the satisfiability problem stays easy [BSSW98] (as for OBDDs).

Theorem 4.40 *The satisfiability problem for k-OBDDs is in* **P***, for any* $k \geq 1$.

Proof We show the claim for $k = 2$. The extension to arbitrary k is straightforward. So let B be a 2-OBDD and assume that in both layers *all* variables are tested on every path through the layer. This property can be achieved easily by introducing *redundant nodes* in the sense of Rule 2 below.

Now let $v_1, \dots, v_m$ be the nodes in the second layer that are reached when coming from the first layer. We define several OBDDs that are subprograms of B. For $i = 1, \dots, m$, define

- $B_2(v_i)$ as the subprogram of B with initial node v_i, and
- $B_1(v_i)$ as the first layer of B, where v_i is replaced by the accepting node and, for all $j \neq i$, v_j is replaced by the rejecting node.

An input $\boldsymbol{a}$ that is accepted by B must cross the layers via some edge (u, v_i) for some node u, and must then be accepted in the second layer. In other words, $\boldsymbol{a}$ is accepted by $B_1(v_i)$ and $B_2(v_i)$ for some i.

Invoking Lemma 4.37, we can construct an ordered branching program $B'(v_i)$ that computes $B_1(v_i) \wedge B_2(v_i)$. Then B is satisfiable iff there exists a $B'(v_i)$ that is satisfiable. So we have reduced our problem to check the satisfiability of several ordered branching programs. □

Graph Driven Branching Programs

An ordered branching programs defines a total order on the variables. Such an order can be represented as a *list* that starts with the smallest and ends with the largest variable (in that order). The list can actually be derived from an ordered branching program as follows: combine the two leaves into one sink node and then apply Rule 1 as long as possible.

Sieling and Wegener [SW95] and independently Gergov and Meinel [GM94] applied this procedure to read-once branching programs. The result is a directed acyclic graph with one initial and one sink node. All nodes except the sink have out-degree two and are labelled with a variable. The two outgoing edges have label 0 and 1, respectively. On each path from the initial node to the sink, every variable occurs at most once.

Sieling and Wegener [SW95] called this the *oracle graph* of a read-once branching program, whereas Gergov and Meinel [GM94] called it its *type*.

We have seen above that the boolean combination of two read-once branching programs can lead to exponentially larger programs. Note however that the two programs we combined are of different type. So how about when we only consider read-once branching programs of the same type? This seems like a proper generalization of ordered branching programs because these also are of the same type, namely lists.

In fact, it turned out that this works [SW95, GM94]: read-once branching programs of a fixed type have a unique minimal normal form, namely the reduced form, and can be combined efficiently. Hence they also allow efficient equivalence tests. The proofs are based on the ones for ordered branching programs, but the extension is not trivial. We refer the interested reader to the original papers or to the excellent survey paper by Wegener [Weg94].

Because every read-once branching program defines a type, this considerably extends the possibilities of ordered branching programs.

4.4 Nondeterministic Branching Programs

Nondeterministic branching programs are defined as deterministic ones but can have additional unlabelled nodes with unbounded fan-out.

On some input, when the evaluation reaches such a *nondeterministic node*, one can choose an arbitrary edge to proceed. A nondeterministic branching program accepts an input, iff there exists a path that leads to the accepting node.

Restricted versions like read-k-times or ordered nondeterministic branching programs are defined in the obvious way.

We give examples of functions that can be computed by small nondeterministic OBDDs, but that require exponential size deterministic OBDDs.

HIDDEN-WEIGHTED-BIT. Let $x = x_1 \cdots x_n$. Recall from the examples of the beginning of Section 4.3 (page 85) that ordered branching programs can count in unary within size $O(n^2)$. Let

$$C_i^u(x) \quad = \quad \begin{cases} 1 & \text{if } w(x) = i, \\ 0 & \text{otherwise,} \end{cases}$$

where w is the weight function.

By Lemma 4.37, we can construct an ordered branching program for the function $x_i \wedge C_i^u(x)$. Finally observe that

$$\textsc{Hidden-Weighted-Bit}(x) \quad = \quad \bigvee_{i=0}^{n} x_i \wedge C_i^u(x),$$

with $x_0 = 0$. Hence we define the branching program B to have one nondeterministic node at the root that branches to $n+1$ ordered branching programs, that all obey the same order. The size of B is $O(n^3)$.

INDIRECT-STORAGE-ACCESS. Let $x = x_0 \cdots x_{m-1} \in \{0,1\}^m$ and $y = y_0 \cdots y_{k-1} \in \{0,1\}^k$, where $m = 2^k$. Now we use the binary counter for y defined in Section 4.3. Define $C_i^b(y) = 1$, iff $y = i$. C_i^b can be realized by a ordered branching program of size $O((2^k)^2) = O(m^2)$.

We express the function INDIRECT-STORAGE-ACCESS in terms of C^b. Namely, it has value 1 iff y has value j, the j-th block of length k in x, $x^j = x_{jk} \cdots x_{(j+1)k-1}$, has value i, and x_i has value 1. That is

$$\textsc{Indirect-Storage-Access}(x, y) \quad = \quad \bigvee_{j=0}^{\lfloor \frac{m}{k} \rfloor - 1} \bigvee_{i=0}^{m} \left(C_j^b(y) \wedge C_i^b(x^j) \wedge x_i \right).$$

The satisfiability problem for nondeterministic read-once branching programs is a reachability problem on graphs as in the case of deterministic ones, and therefore solvable in polynomial time.

On the other hand, the equivalence problem is hard.

Theorem 4.41 *The equivalence problem for nondeterministic ordered branching programs is* **coNP***-complete.*

Proof We reduce TAUT. Let $F(x_1, \dots, x_n)$ be a DNF-formula, $F = C_1 \vee \dots \vee C_m$. Let B_k be an ordered branching program that accepts an input $\boldsymbol{a} \in \{0,1\}^n$ iff $\boldsymbol{a}$ does satisfy clause C_k. Fix the variable order to $x_1 < \dots < x_n$.

Define branching program B with a nondeterministic node at the root that branches to all programs B_k. Then we have for all assignments $\boldsymbol{a} \in \{0,1\}^n$

$$\begin{aligned} F(\boldsymbol{a}) = 1 &\iff \exists k \; C_k(\boldsymbol{a}) = 1 \\ &\iff \exists k \; B_k(\boldsymbol{a}) = 1 \\ &\iff B(\boldsymbol{a}) = 1. \end{aligned}$$

It follows that $F \in$ TAUT $\iff B \equiv 1$. This proves the claim. □

Partitioned OBDDs [JBFA92] are nondeterministic OBDDs that make a very limited use of the nondeterminism (only the root is a nondeterministic node) and that have in addition pointers (*window functions*) to the part below the root that is satisfied by a given input. Partitioned OBDDs are further investigated in [NJFSV96] and in [BW97]. In particular they allow efficient equivalence tests.

Another way of viewing nondeterministic branching programs is to label the extra nodes with boolean operations. On some input, the evaluation of such a node is then done according to this boolean operation. For example, the nondeterministic nodes can be seen as being labelled by the or-function. Meinel [Mei89] studied such kind of extended branching programs. The complementary class of nondeterministic branching programs is obtained by allowing only the and-function as label. Complementing Theorem 4.41, we have:

Corollary 4.42 *The satisfiability problem for ordered branching programs with and-nodes is* **NP***-complete.*

In the next section we consider ordered branching programs that have nodes labelled by the parity-function.

4.5 Parity Branching Programs

Gergov and Meinel [GM96] introduced a counting version based on nondeterministic OBDDs: *parity ordered branching programs,* $\oplus$*-OBDDs* for short. $\oplus$-OBDDs are nondeterministic OBDDs that accept a given input iff there is an *odd* number of accepting paths.

On first glance, $\oplus$-OBDDs do not seem to be easier to handle than nondeterministic OBDDs. But this turns out to be *not* true. The reason is that $\oplus$-OBDDs can be interpreted as an expression over a field, GF(2).

Note first that the nondeterministic OBDDs for Hidden-Weighted-Bit and Indirect-Storage-Access given in the preceding section both have precisely one accepting path when they accept an input. Therefore these programs already constitute $\oplus$-OBDDs. It is also straightforward to obtain a $\oplus$-OBDD for the function Parity-of-Triangles.

For the rest of this section it is more convinient to consider $\oplus$-OBDDs without having extra nondeterministic nodes but instead have the nondeterminism at the nodes labelled with variables. This is achieved as follows: if a b-edge, $b \in \{0, 1\}$, of node u leads to a nondetermistic node, then we skip this edge and instead introduce b-edges from u to all nodes where the nondetermistic node has edges to. In a similar way one can handle the case of a nondetermistic node at the root.

Gergov and Meinel [GM96] showed that $\oplus$-OBDDs are closed under Boolean operations.

Lemma 4.43 *$\oplus$-OBDDs (with the same order) can be combined in polynomial time according to any Boolean operation.*

Proof Let B_0 and B_1 be $\oplus$-OBDDs that obey the same variable order, and assume that the root nodes of both are labelled with the same variable (otherwise introduce a redundant node). We obtain a $\oplus$-OBDD computing $B_0 \oplus B_1$ by simply joining the two root nodes into one node. In particular, by choosing $B_1 = 1$, it follows that one can complement $\oplus$-OBDDs.

To obtain a $\oplus$-OBDD for $B_0 \wedge B_1$, observe that we can build a cross-product $B_0 \times B_1$ of B_0 and B_1 in the same way as we did it for deterministic OBDDs in Lemma 4.37. The effect is that the number of paths multiply. That is, the number of paths that reach the (accept,accept) node in $B_0 \times B_1$ on some input is precisely the product of the number of accepting paths of B_0 and B_1 on that input. Therefore we take the (accept,accept) node as the accepting node of $B_0 \times B_1$ and the other leaves as rejecting nodes. Then the resulting $\oplus$-OBDD computes $B_0 \wedge B_1$. □

How about the equivalence problem for $\oplus$-OBDDs? Since $B_0 \equiv B_1 \iff B_0 \oplus B_1 \equiv 0$, it suffices to consider the satisfiability problem (by Lemma 4.43). First Gergov and Meinel [GM96] gave an efficient *proba-*

bilistic satisfiability test. Then Waack [Waa97] exhibited even a (deterministic) polynomial-time satisfiability test for $\oplus$-OBDDs.

The probabilistic satisfiability test follows easily from the results we already have established: first arithmetize $\oplus$-OBDDs over GF(2). This is completely analogous as we did in Theorem 4.12 for read-once branching programs, but with arithmetic done in GF(2). (In the second item in the proof of Theorem 4.12, we have to assign the corresponding polynomial to *all* 0- respectively 1-edges that leave a node.) This yields a multilinear polynomial over GF(2) and hence, we can apply Corollary 4.20 to obtain a probabilistic zero-test. Clearly this works more general for read-once branching programs.

Proposition 4.44 *The equivalence problem for $\oplus$-read-once branching programs is in* **coRP**.

It follows from Corollary 4.23 that the corresponding isomorphism problem cannot be **NP**-hard unless the polynomial hierarchy collapses.

Corollary 4.45 *The isomorphism problem for $\oplus$-read-once branching programs is not* **NP***-hard unless* $\mathbf{PH} = \Sigma_2\mathbf{P}$.

Next we show how to test the satisfiability of $\oplus$-OBDDs *deterministically* [Waa97].

Theorem 4.46 *The satisfiability problem for $\oplus$-OBDDs is in* **P**.

Proof Let B be a $\oplus$-OBDD that computes some function f. We want to know whether $f = 0$.

Consider the root of B and let w.l.o.g. x_1 be its label. B computes some function at each node (in the variables that occur below that node). f is computed at the root. Let $g_1, \dots, g_k$ be the functions computed at the nodes that are reached by the 0-edges of the root. Functions $g_{k+1}, \dots, g_{k+l}$ are defined similar for the 1-edges. Then f can be expressed by an equation over GF(2) as follows:

$$\begin{aligned} f &= (x_1+1)g_1 + \cdots + (x_1+1)g_k + x_1 g_{k+1} + \cdots + x_1 g_{k+l} && (4.3)\\ &= g_1 + \cdots + g_k + x_1(g_1 + \cdots + g_{k+l}). && (4.4)\end{aligned}$$

Since x_1 does not occur in the g_i's, we have that $f = 0$ iff $g_1 + \cdots + g_k = 0$ and $g_1 + \cdots + g_{k+l} = 0$. But this is just saying that the g_i's are *linearly dependent*: some of them can in fact be expressed by others and are therefore superfluous. Thus our goal now is to find a way to throw out superfluous nodes such that the functions computed at the remaining nodes are linearly independent. Then equation (4.4) for the root node must directly reduce to $f = 0$ iff f is in fact the zero-function. Therefore this will solve the problem.

Our algorithm works bottom up, starting at the leaf nodes. The two leaves represent the constant functions 0 and 1, respectively. Suppose that we have reduced B up to some level and let $b_1, \dots, b_k$ be the linearly independent functions computed by the nodes up to that level. We proceed to the next level where variable x_l is tested. Let there be m nodes that compute the functions $f_1, \dots, f_m$. Each f_i can be expressed analogously to f in equation (4.4) in terms of the $2k$ *basis functions* $b_1, \dots, b_k$, $x_l b_1, \dots, x_l b_k$. Hence we get a set of m linear equations

$$\begin{pmatrix} a_{1,1} & \cdots & a_{1,2k} \\ \vdots & \vdots & \vdots \\ a_{m,1} & \cdots & a_{m,2k} \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ b_k \\ x_l b_1 \\ \vdots \\ x_l b_k \end{pmatrix} = \begin{pmatrix} f_1 \\ \vdots \\ f_m \end{pmatrix},$$

where $a_{i,j} \in \{0,1\}$ are the corresponding coefficients of each f_i.

Now we do Gaussian elimination to bring the matrix into lower triangular form and simultaneously do the same transformations on the vector $(f_1, \dots, f_m)^T$. Suppose the resulting matrix has d zero-rows at the bottom. Then we get d equations such that we can express $f_{m-d+1}, \dots, f_m$ in terms of $f_1, \dots, f_{m-d}$. So the nodes computing $f_{m-d+1}, \dots, f_m$ are superfluous. We throw them out and change the incoming edges to them according to the equations we obtained, in order to get an equivalent branching program. Now all nodes compute linearly independent functions up that level.

For the running time of the algorithm note that we do Gaussian elimination at every level of B. Hence this sums up to time $O(n|B|^3)$. □

Corollary 4.47 *The equivalence problem for* $\oplus$*-OBDDs that obey the same order is in* **P**.

We remark that the probabilistic algorithm of Gergov and Meinel [GM96] is more efficient than the above deterministic algorithm: it makes linearly many arithmetic operations so that the overall running time is linear up to a polylog factor (see [MS97] for more details).

Behrens and Waack [BW98] have extended the corollary to $\oplus$-OBDDs with *different* orders. Note that this puts the isomorphism problem for $\oplus$-OBDDs into **NP**.

The transformation of a $\oplus$-OBDD B done in the proof of Theorem 4.46 yields a new $\oplus$-OBDD B' such that the functions computed at the nodes of B' are linearly independent. Therefore B' consists only of the rejecting node if $B \equiv 0$. However, in general, B' must not be a minimal $\oplus$-OBDD equivalent to B (with respect to the same order). Nevertheless, Waack [Waa97] showed how to get a minimal $\oplus$-OBDD from B' and Agrawal [Agr97] observed that the construction can be done such that the outcome is *uniquely determined*. Thus we get a *normal form* for $\oplus$-OBDDs.

Theorem 4.48 *Let $f(x_1, \ldots, x_n)$ be a Boolean function and B be some $\oplus$-OBDD that computes f with variable order π. A minimal $\oplus$-OBDD for f (with respect to π) can be computed in polynomial time in $|B|$. Moreover, the construction can be done such that the outcome is* the same *minimal $\oplus$-OBDD for all $\oplus$-OBDDs that compute f (with respect to π).*

Proof Let B be a $\oplus$-OBDD that computes f and let B' be the $\oplus$-OBDD constructed in Theorem 4.46 from B. Our goal is to transform B' into a $\oplus$-OBDD B'' such that

- the nodes of B'' still compute linearly independent functions and
- each node computes some subfunction of f.
  Somewhat stronger, we will have that for each node u there is an input $\boldsymbol{a}$ such that B'' is deterministic on input $\boldsymbol{a}$ from the initial node until node u is reached.

Then B'' must be minimal because every $\oplus$-OBDD $\tilde{B}$ that computes f must also be able to compute all subfunctions of f, maybe as a linear combination of several subprograms. But since the subfunctions that occur in B'' are linearly independent, $\tilde{B}$ cannot have less nodes than B''.

The algorithm works top-down, starting at the initial node. Clearly condition (i) and (ii) hold for the initial node. So suppose we have reached some level such that the conditions hold and let $u_1, \ldots, u_k$ be the nodes of this level. Let x_u be the variable tested by these nodes. Let $v_1, \ldots, v_l$ and $w_1, \ldots, w_m$ be the nodes of the next two levels where variable x_v and x_w is tested, respectively.

We throw out the level of v-nodes and replace it by $2k$ new nodes v_1^0, v_1^1, v_2^0, v_2^1, …, v_k^0, v_k^1 that still test variable x_v and add edges to v_i^0 and v_i^1 from node u_i labelled 0 and 1, respectively. Then we have to connect the new v-nodes with the w-nodes such that the function computed by B' is not altered. That is, we put an edge labelled b' from v_i^b to w_j, if there was a b-edge from u_i to v_r and a b'-edge from v_r to w_j in B', for an odd number of r's.

Now the functions computed at the new v-nodes can be linearly dependent. However, from the $\oplus$-OBDD constructed so far we can express the $2k$ v-nodes in terms of the linearly independent w-nodes. So we can again do Gaussian elimination as in the proof of Theorem 4.46 and reduce the v-nodes so that they compute linearly independent functions. Then we proceed to the next level.

The construction can be made unique by imposing some order on the v-nodes (for example using the lexicographic order with respect to the deterministic path leading to each v-node). Then we can always eliminate, say, the largest linearly dependent v-nodes. □

Recall that a *minimal* branching program minimizes the number of nodes. In the case of nondeterministic branching programs, the number of edges can be up to quadratic in the number of nodes. So the number of edges would be a more appropriate measure for the size of nondeterministic branching programs. However, it is not at all clear how to construct edge-minimal $\oplus$-OBDDs.

To solve the equivalence problem for $\oplus$-OBDDs that can have different orders [BW98] we adapt the proof of Theorem 4.36 (on page 89). Recall that we reduced the equivalence problem for two programs B and C to smaller ones by looking at the subprograms of B and C. If, say, variable x_1 is at the root of C, we consider programs $B_{x_1=b}$ and $C_{x_1=b}$ for $b = 0, 1$. A $\oplus$-OBDD for $B_{x_1=b}$ can be easily constructed from B by skipping the corresponding layer in B and adjusting the edges appropriately.

The main difference to the algorithm in Theorem 4.36 occurs for C: because there can be several b-edges at a node, $C_{x_1=b}$ might not be a subprogram of C but a *sum* of subprograms, i.e., $C_{x_1=b} = C_{u_1} \oplus \cdots \oplus C_{u_l}$. Suppose now that C is already minimized by the algorithm from Theorem 4.48. Recall that each node of C is also reached deterministically for some input. Therefore each subprogram C_{u_i} also occurs directly as a second component of some equivalence problem. That is, there are pairs $(B_1, C_{u_1}), \ldots, (B_l, C_{u_l})$ that have to be checked for equivalence. Hence, the equivalence of $B_{x_1=b}$ and $C_{x_1=b}$ can be verified by checking instead that $B_{x_1=b} \equiv B_{u_1} \oplus \cdots \oplus B_{u_l}$. But this can be done by Corollary 4.47. By the same argument as in Theorem 4.36 the algorithm runs in polynomial time.

Theorem 4.49 *The equivalence problem for $\oplus$-OBDDs (with different orders) is in* **P**.

Corollary 4.50 *The isomorphism problem for $\oplus$-OBDDs is in* **NP** *but not* **NP***-complete unless* $\mathbf{PH} = \Sigma_2\mathbf{P}$.

The equivalence test in Theorem 4.49 can easily be modified to change the order of a given $\oplus$-OBDD into another one. This gives the analog of Theorem 4.33 (on page 88) for $\oplus$-OBDDs.

In the deterministic case, the equivalence of an OBDD with a read-once branching program can be tested in polynomial time by Theorem 4.36. It is an open problem whether this carries over to $\oplus$-OBDDs versus $\oplus$-read-once branching programs. This would improve Theorem 4.49.

Although our algorithms for the satisfiability and equivalence problem for $\oplus$-OBDDs run in polynomial-time, they are not efficient enough for practical purposes. There we should have linear-time algorithms. However, Löbbing, Sieling, and Wegener [LSW98] have shown that such algorithms should be hard to find because they would imply a linear-time algorithm for matrix rank computation. The latter would be a major improvement on the time bounds we have for current algorithms in linear algebra.

4.6 Probabilistic Branching Programs

Ablayev and Karpinski [AK96] introduced *probabilistic branching programs*.

Definition 4.51 Probabilistic branching programs *are branching programs that additionally have unlabelled nodes of unbounded fan-out. These nodes are called* probabilistic nodes.

On some input, when we reach a probabilistic node, the edge to proceed is chosen under uniform distribution out of all outgoing edges. A probabilistic branching program accepts its input if the probability to reach the accepting node is at least $1/2$*. Otherwise the input is rejected.*

A probabilistic branching program has bounded error*, if there is an* $\epsilon > 0$ *such that the acceptance probability is either at most* $1/2 - \epsilon$ *or at least* $1/2 + \epsilon$ *on all inputs.*

We mainly consider bounded-error probabilistic ordered branching programs, BP-OBDDs for short. To begin with, we show some basic properties of BP-OBDDs in the next section: BP-OBDDs can be amplified and are closed under boolean operations.

In Section 4.6.2 we give some examples of problems that can be solved by polynomial-size BP-OBDDs but that require exponential-size read-once branching programs.

In Section 4.6.3 we show that the satisfiability problem for BP-OBDDs is **NP**-complete. Therefore the equivalence problem for BP-OBDDs is **coNP**-complete.

If, on the other hand, the error probability of a given BP-OBDD is very small compared to its size, then the satisfiability problem can be solved in polynomial time. We present an algorithm in Section 4.6.4.

4.6.1 Basic Properties

Let B be a probabilistic branching program in n variables that computes some function f with error $1/2 - \epsilon$. That is, for all $\boldsymbol{a} \in \{0,1\}^n$ we have

$$\mathbf{Pr}[B(\boldsymbol{a}) = f(\boldsymbol{a})] \geq 1/2 + \epsilon.$$

General bounded-error probabilistic branching programs can be derandomized: the majority-function is a symmetric function and can therefore be computed by a polynomial-size ordered branching program. Invoking Lemma 4.56, we can compose the majority-function and B. The result is a branching program that computes a majority vote on the outcomes of B. If we take enough copies of B (depending on ϵ), the error probability of the resulting branching program B' can be made arbitrarily small, say $2^{-(n+1)}$, by Lemma 2.2. There are 2^n strings of length n. Therefore the probability that a setting of the random bits of B' leads to a correct answer on *all* inputs of length n is at least $1/2$. Hence there must *exist* such a setting. Fixing the random choices of B' that way leads to a *deterministic* branching program that computes f. We conclude that bounded-error probabilistic branching program can be derandomized and are therefore not more powerful than deterministic ones.

Next we consider BP-OBDDs. The above construction does not work here because it would destroy the read-once property. Instead we use the *cross product* from Lemma 4.37. Let B_0 and B_1 be BP-OBDDs and assume that they are *layered* such that there are alternating probabilistic and deterministic nodes,and that furthermore *all* variables appear on every path. This can easily be achieved by introducing *redundant nodes*. In the case of a missing probabilistic node introduce a new node with only one edge that has probability 1. Then program $B_0 \times B_1$ has the same layers as B_0 and B_1, and the nodes of each layer are the cross product of nodes of B_0 and B_1 at the corresponding layer.

The size of $B_0 \times B_1$ is bounded by $|B_0||B_1|$. Also the number of paths that reach a node multiply. Therefore, if some input x is accepted by B_0 with probability p_0 and by B_1 with probability p_1, then $B_0 \times B_1$ on input x reaches

- the (accept,accept)-node with probability $p_0 p_1$,
- the (accept,reject)-node with probability $p_0(1 - p_1)$,
- the (reject,accept)-node with probability $(1 - p_0)p_1$, and
- the(reject,reject)-node with probability $(1 - p_0)(1 - p_1)$.

Given BP-OBDD B, we apply the construction t times with $B = B_0 = B_1$. This yields program B^t that consists of t factors B. The acceptance of B^t is defined according to a *majority vote* on its t factors. Clearly this will amplify the correctness again according to Lemma 2.2.

However, we cannot derandomize BP-OBDDs. The reason is that the cross product construction squares the size of the branching program. Therefore $B^t = B \times \cdots \times B$ (t times) has size $|B|^t$. In order to keep the size polynomial in $|B|$, t has to be a constant. By Lemma 2.2 it follows that we can amplify the correctness of B from $\frac{1}{2} + \epsilon$ to $1 - \delta$, for any constant $\delta > 0$.

Lemma 4.52 *Let B be a BP-OBDD that computes some function f with error $1/2 - \epsilon$ and let $0 < \delta < 1/2$. Then there is BP-OBDD of size polynomial in $|B|$ that computes f with error δ.*

Lemma 4.53 *BP-OBDDs (with the same order) can be combined in polynomial time according to any boolean operation.*

Proof BP-OBDDs can be complemented by exchanging accepting and rejecting states. Therefore it remains to show how to construct the conjunction of two BP-OBDDs B_0 and B_1 with n variables. Let the error of both be at most $1/4$ (applying Lemma 4.52).

The idea to get a BP-OBDD that computes $B_0 \wedge B_1$ is the same as for deterministic OBDDs: assume that B_0 and B_1 are layered such that deterministic and probabilistic nodes alternate and that all variables occur on every path. Then we can build the cross product $B = B_0 \times B_1$ and define the (accept,accept) node as the accepting node of B and the other leaves as rejecting nodes.

Let $\boldsymbol{a} \in \Sigma^n$. If $\boldsymbol{a}$ is accepted by both, B_0 and B_1, then B accepts $\boldsymbol{a}$ with probability at least $(3/4)(3/4) = 9/16$. On the other hand, if $\boldsymbol{a}$ is rejected by either B_0 or B_1, then B accepts $\boldsymbol{a}$ with probability at most $1/4$. □

4.6.2 The Computational Power

Ablayev [Abl97] exhibits a function f where nondeterministic read-once branching programs require exponential size, whereas f can be computed by polynomial-size BP-OBDDs. The definition of f is based on comparing certain substrings of an input string. On the other hand, INDIRECT-

Storage-Access and Hidden-Weighted-Bit require exponential-size BP-OBDDs [Sau98] (see [Abl97, Sau98] for more lower bounds).

To demonstrate how branching programs can use randomization we give some examples. Although polynomial size BP-OBDDs cannot multiply [AK98], they can nevertheless *verify* multiplication [AT98] (and independently in [AK98]). That is, we have given x, y, and z and have to check that $xy = z$.

Theorem 4.54 *BP-OBDDs can verify multiplication with one-sided error and within polynomial-size.*

Proof Given n-bit numbers x and y and $2n$-bit number z, we verify that $xy = z$ by using a binary counter as explained at the beginning of Section 4.3. However, the binary counter C^b would lead to an exponential-size branching program. Therefore we count *modulo some small prime* p via function $C^{b,p}$. That is, we construct an ordered branching program $B_p(x, y, z)$ that checks whether

$$xy \equiv z \pmod{p}.$$

B_p is easy to construct: start by computing $(x \bmod p)$ via $C^{b,p}$. Then we read the bits of $y = y_{n-1} \cdots y_0$. Since

$$(x \bmod p)\ y \equiv \sum_{i=0}^{n-1} (x \bmod p)\ 2^i y_i \pmod{p},$$

we can also compute $(xy \bmod p)$. Now it remains to compute $(z \bmod p)$ and to compare it with $(xy \bmod p)$. The size of B_p is $O(p^2 n)$. Note that B_p is ordered.

If indeed $xy = z$, then B_p will accept independently of p. On the other hand, if $xy \neq z$ then B can accept anyway for some prime p, because we could still have that $xy \equiv z \pmod{p}$ in this case.

The crucial point now is that our test can fail only for a small number of primes: namely only those primes that divide $xy - z$. Since $|xy - z| \leq 2^{2n}$, there are at most $2n$ primes where our test can fail.

The final program B works as follows. Let $p_1, \ldots, p_{4n}$ be the first $4n$ prime numbers. B probabilistically branches to all programs B_{p_i}. Each of those checks whether $xy \equiv z \pmod{p_i}$. If $xy = z$, then $\mathbf{Pr}[B(X) \text{ accepts}] = 1$. Otherwise $\mathbf{Pr}[B(X) \text{ accepts}] \leq 1/2$.

How large is B? An upper bound is $4n|B_{p_{4n}}|$. Therefore it is enough to bound p_{4n} by some polynomial. By the Prime Number Theorem there are approximately $N/\ln N$ primes less than N. Hence, there are $4n$ primes less than $N = 8n \ln n$, □

The technique to verify equations using small prime numbers is called *finger printing technique* and goes back to Freivalds [Fre77]. It was first used in the context of BP-OBDDs by Ablayev and Karpinski [AK96].

Theorem 4.54 can be generalized: the equation $xy = z$ or equivalently $xy - z = 0$ is a specific *Diophantine equation*. That is, it asks for the existence of zeros of a multivariate polynomial with integer coefficients over the integers. The same technique as above will work for more general Diophantine equations as long as we can evaluate them modulo some prime number with a polynomial-size OBDD.

For example we can compute $(x^k \bmod p)$ with an OBDD: modify the binary counter $C^{b,p}(x)$ as follows. When it reads the last bit of x and the edge leads to counter r, then redirect the edge to counter $(r^k \bmod p)$.

It follows that we can check Diophantine equations that are given, for example, as a sum of monomials. However, since we are restricted to read every input bit at most once, all variables should occur only once in the polynomial expressions.

Corollary 4.55 *If a polynomial equation can be checked modulo prime numbers by a polynomial-size OBDD, then the equation can be checked by a polynomial-size BP-OBDD.*

We give two more examples that can be solved by small BP-OBDDs, PERMUTATION-MATRIX [Sau98] and CLIQUE-ONLY [AT98], which was independently observed by M. Sauerhoff. [1] The trick in both cases is to transform the problem into an appropriate equation and then apply Corollary 4.55.

Theorem 4.56 PERMUTATION-MATRIX *has polynomial-size BP-OBDDs with one-sided error.*

Proof Let X be a $n \times n$ 0-1-matrix. Let $x_1, \ldots, x_n$ denote the n rows of X and interpret each x_i as a binary number. X is a permutation matrix iff

- each x_i contains precisely one 1 and therefore has value 2^j if the 1 is at position j, and
- $\sum_{i=1}^{n} x_i = 2^n - 1$.

We take the *row-wise* variable order, i.e., $x_{1,1} < x_{1,2} < \cdots < x_{n,n-1} < x_{n,n}$. Condition (i) can clearly be verified by an ordered branching program B_1. Condition (ii) ensures that there is precisely one 1 in every

[1] Personal communication. In fact, Sauerhoff constructs a BP-OBDD for the slightly more difficult case that only the upper triangle of the (symmetric) adjacency matrix of the graph is given as input.

column. This equation can be checked modulo small primes by ordered branching programs. Therefore, by Corollary 4.55, we get a BP-OBDD B_2 such that

- $\mathbf{Pr}[B_2(X) \text{ accepts}] = 1$, if X is a permutation matrix,
- $\mathbf{Pr}[B_2(X) \text{ accepts}] \leq 1/2$, otherwise.

Finally we apply Lemma 4.53 to get a BP-OBDD for $B_1 \wedge B_2$ that computes PERMUTATION-MATRIX. □

Theorem 4.57 CLIQUE-ONLY *has polynomial-size BP-OBDDs with one-sided error.*

Proof Let A be an adjacency matrix of a graph G with n nodes, and $k \leq n$. G has only a k-clique iff

- there exist k rows such that each row contains precisely $k-1$ ones and the remaining rows are all zero, and
- any two nonzero rows must be identical except for the positions where they intersect the main diagonal.

Again we choose the *row-wise* variable order, i.e., $x_{1,1} < x_{1,2} < \cdots < x_{n,n-1} < x_{n,n}$. Condition (i) is easy to check since this requires a combination of unary counters which can therefore be done by deterministic OBDDs . The variable order is *row-wise*, i.e., $x_{1,1} < x_{1,2} < \cdots < x_{n,n-1} < x_{n,n}$. Therefore it remains to check condition (ii) with an BP-OBDD and then apply Lemma 4.53 to build the conjunction of the two programs.

Suppose we add a one at the diagonal positions of the nonzero rows of A. Then condition (ii) says that the resulting nonzero rows must be *identical*. Let $x_1, \ldots, x_n$ denote the rows of A and interpret them as binary numbers. Introducing a one at the diagonal position of nonzero row x_j corresponds to adding 2^{n-j} to x_j. Therefore it suffices to check that for any two consecutive nonzero rows x_j and x_k, we have

$$x_j + 2^{n-j} \quad = \quad x_k + 2^{n-k}. \tag{4.5}$$

We construct a deterministic OBDD B_p that verifies equation (4.5) modulo some small prime p. B_p looks for the first nonzero row, say j and computes $s = (x_j + 2^{n-j} \bmod p)$ by doing a binary count modulo p via function $C^{b,p}$. Then B_p checks for each forthcoming nonzero row k that $x_k + 2^{n-k} = s$. (again by using $C^{b,p}$ to determine the value $(x_k + 2^{n-k} \bmod p)$). The size of B_p is $O(n^2p^2)$.

Now Corollary 4.55 provides a BP-OBDD that checks condition (ii). □

4.6.3 Satisfiability Problems

Let us start with (unbounded-error) probabilistic OBDDs. That is, for a given such program B we want to know whether

$$\exists \boldsymbol{a}: \ \mathbf{Pr}[B(\boldsymbol{a}) \text{ accepts}] \geq 1/2.$$

For every given input $\boldsymbol{a} \in \{0,1\}^n$, the number of paths in B that lead to the accepting, respectively rejecting node can be computed in $\#\mathbf{L}$, and therefore in polynomial time. That is, we can compute the probability $\mathbf{Pr}[B(\boldsymbol{a}) \text{ accepts}]$ in polynomial time. The same argument works for probabilistic read-once branching programs. We conclude that the satisfiability problem is in **NP**. In fact, it is **NP**-complete.

Proposition 4.58 *The satisfiability problem for probabilistic ordered branching programs (with unbounded error) is* **NP***-complete.*

Proof We reduce CNF-SAT. Let $F = \bigwedge_{i=1}^{m} C_i$ be a CNF-formula with m clauses $C_1, \ldots, C_m$. We construct a probabilistic ordered branching program B_F such that

$$F \in \text{SAT} \quad \Longleftrightarrow \quad \exists \boldsymbol{a}: \ \mathbf{Pr}[B_F(\boldsymbol{a}) \text{ accepts}] \geq 1/2.$$

Let B_i be a (deterministic) ordered branching program that accepts if clause C_i is satisfied on a given input. B_F is constructed as follows. The initial node of B_F is a probabilistic node that branches $2m$ times. m of these edges lead to the initial nodes of $B_1, \ldots, B_m$. The remaining m edges go directly to the rejecting node.

It follows that B_F accepts input $\boldsymbol{a}$ if and only if all the programs B_i accept which is only possible when $\boldsymbol{a}$ satisfies F. □

Let us turn to the satisfiability problem for probabilistic ordered branching programs with *bounded error*, BP-OBDDs. A subtlety here is that this includes to check that a given program has in fact bounded error on *all* inputs. However, already this problem is **coNP**-complete.

Proposition 4.59 *Given a probabilistic ordered branching program B and an $\epsilon > 0$. The problem to decide whether B is of bounded error ϵ is* **coNP***-complete.*

Proof The argument is essentially the same as for Proposition 4.58. Consider the case $\epsilon = 1/4$. Construct B_F as above but with $4m - 4$ edges leaving the initial node and $3m - 4$ of them going directly to the rejecting node. Then we have

$$F \in \textsc{Sat} \iff \exists \boldsymbol{a}:\ 1/4 < \mathbf{Pr}[B_F(\boldsymbol{a}) \text{ accepts}] < 3/4.$$

□

Hence, efficient satisfiability algorithms can only exist for the *promise version* of the problem: given B and $\epsilon > 0$, we take as a promise that B is in fact a probabilistic ordered branching program *with error* ϵ. With this assumption we want to decide whether $\exists \boldsymbol{a}:\ \mathbf{Pr}[B_F(\boldsymbol{a}) \text{ accepts}] \geq 1/2+\epsilon$. If the promise is not true, then we can give an arbitrary answer. Still this remains a hard problem: the promise version of the satisfiability problem for BP-OBDDs is **NP**-complete [AT98].

Theorem 4.60 *The satisfiability problem for BP-OBDDs is* **NP***-complete.*

Proof Manders and Adleman [MA78] have shown that some specific Diophantine equations, so called *binary quadratics*, are **NP**-complete. More precisely, the following set Q defined over the natural numbers is **NP**-complete:

$$Q = \{\, (a,b,c) \mid \exists x,y:\ ax^2 + by = c \,\}.$$

From Corollary 4.55 we know that BP-OBDDs can verify such binary quadratics. That is, the set

$$Q' = \{\, (a,b,c,x,y) \mid ax^2 + by = c \,\}$$

can be accepted by a polynomial-size BP-OBDD, call it B.

For fixed a,b,c, we can construct a BP-OBDD B_{abc} from B that computes the sub-function of B with a, b, and c plugged in as constants. Recall that B is deterministic except for the root node. Therefore we can obtain B_{abc} by the construction that we already presented for ordered branching programs (see for example in the proof of Theorem 4.36). For all natural numbers a,b,c, we have that

$$(a,b,c) \in Q \iff B_{abc} \text{ is satisfiable.}$$

This proves the theorem. □

Corollary 4.61 *The equivalence problem for BP-OBDDs is* **coNP***-complete.*

Corollary 4.62 *The isomorphism problem for BP-OBDDs is in* $\Sigma_2\mathbf{P}$*, but not* $\Sigma_2\mathbf{P}$*-complete unless* $\mathbf{PH} = \Sigma_3\mathbf{P}$.

The technique used in the proof of Theorem 4.60 reflects a trade-off between the different requirements applications have for a branching program model to be useful (see also [Weg94] for a discussion of this topic). On the one hand side, the model should be able to express important functions such as for example MULTIPLICATION. On the other hand, it should allow an easy equivalence test.

All branching program models considered up to now allow parameterization, i.e., one can plug in some variables as constants and obtain a branching program of the same restricted model (as we did above when going from B to B_{abc}). Now, if we can verify multiplication in such a model or even verify binary quadratics, then, using parameterization, we can express factoring or binary quadratics as a satisfiability problem for that model. However, then there can be *no* efficient satisfiability test unless factoring is in **P** or $\mathbf{P} = \mathbf{NP}$, respectively.

4.6.4 An Efficient Satisfiability Test for BP-OBDDs with Small Error

In this section we develop an *efficient* satisfiability test for BP-OBDDs that have small error with respect to their size [AT98]. More precisely, we consider the *width* of a BP-OBDD:

Definition 4.63 *Let B be a layered BP-OBDD such that deterministic and probabilistic layers alternate (see Section 4.6.1 on page 106). The* width *of B is the maximum number of nodes in a layer.*

Note that the size of B is bounded by $2n$ times its width if B has n variables.

Our efficient satisfiability test for BP-OBDDs works if the error is bounded by $1/(\text{width} + 2)$. That is, we consider the following problem:

BOUNDED-WIDTH-BP-OBDD-SAT
Given a BP-OBDD B with error ϵ and width W such that $\epsilon < 1/(W+2)$. Decide whether B is satisfiable.

Theorem 4.64 BOUNDED-WIDTH-BP-OBDD-SAT $\in \mathbf{P}$.

Proof Let B be some BP-OBDD with n variables $x_1, \ldots, x_n$, width W and error $\epsilon < 1/(W + 2)$.

The Model. As described above, we can assume that B is layered, so that probabilistic and deterministic layers alternate. We number the layers according to their distance to the root. The root layer (which is a single node) has number 0.

We will modify B and thereby change the probabilities a probabilistic node branches to its successors. In the beginning, all probabilities have the form $1/p$ if a node has p successors. Since we will also get other rational numbers as probabilities, we generalize the BP-OBDD model and write the probabilities on the edges, for example as pairs of integers in binary representation.

Outline of the Algorithm. We want to find out whether B is satisfiable, i.e., whether there exists an input such that B accepts with probability greater than $1 - \epsilon$. We transform B layer by layer, starting at the initial node, i.e. with layer $\ell = 0$ and $B_0 = B$. Suppose we have reached layer $\ell \geq 0$ and let B_ℓ denote the branching program constructed so far. B_ℓ has the following properties which are invariants of our construction:

- [(I1)] B_ℓ is *deterministic* up to layer $\ell - 1$, and identical to B from layer $\ell + 1$ downwards,
- [(I2)] the error of B_ℓ is bounded by ϵ,
- [(I3)] the width of B_ℓ is at most $W + 1$, and
- [(I4)] B_ℓ is satisfiable iff B is satisfiable.

In general, B_ℓ accepts only a subset of the strings accepted by B. Nevertheless, we ensure property (I4) which is enough to check the satisfiability of B. In particular, the resulting branching program, after we have processed the last level, is a *deterministic* ordered branching program. Since the satisfiability problem for ordered branching programs is simply a reachability problem on a directed graph, this will prove our theorem.

The Transformation of Layer ℓ. We now describe how to process layer $\ell \geq 0$. If layer ℓ is deterministic then define $B_{\ell+1} = B_l$ and proceed to the next layer. So let layer ℓ be a probabilistic layer of B_ℓ. We consider three consecutive layers. Let each layer have the maximum number of nodes. [2]

- [layer ℓ] consists of probabilistic nodes $u_1, \ldots, u_{W+1}$. We can inductively assume that the part of B_l from the initial node to the u-nodes is deterministic.

[2]If there are less nodes in some layer, we can add dummy nodes that lead to rejection with probability 1.

- [layer $\ell + 1$] consists of deterministic nodes $v_1, \ldots, v_W$ which all query the same variable, say x_t, for some t.
- [layer $\ell + 2$] consists of probabilistic nodes $w_1, \ldots, w_W$.

These nodes are connected as follows.

- [u-nodes with v-nodes:] each node u_i is connected to all the nodes v_j with the edge between them having probability p_{ij} (in case u_i is not connected to some node v_j, we take the probability p_{ij} to be zero). We have $\sum_j p_{ij} = 1$ for each $1 \le i \le W + 1$.
- [v-nodes with w-nodes:] node v_j is connected to nodes $w_{e(j,0)}$ and $w_{e(j,1)}$ via edges labelled 0 and 1 respectively.

Changing u-nodes. Our first step is to make the nodes $u_1, \ldots, u_{W+1}$ deterministic. For this we introduce $2(W + 1)$ new nodes at layer $\ell + 1$, call them $v_1^0, v_1^1, v_2^0, v_2^1, \ldots, v_{W+1}^0, v_{W+1}^1$ (these nodes will be probabilistic) that replace the old v-nodes. The u-nodes get label x_t, the variable queried by the old v-nodes, and we put the b-edge of u_i to node v_i^b, for $b \in \{0, 1\}$.

Changing v-nodes. Next, we introduce an edge from node v_i^b to the node w_k in layer $\ell + 2$ and assign probability $q_{b,i,k}$ to it such that $q_{b,i,k}$ is precisely the probability to reach w_k from u_i in B_ℓ if x_v has value b. This is achieved by defining

$$q_{b,i,k} \quad = \sum_{\substack{j \\ e(j,b)=k}} p_{ij}. \tag{4.6}$$

Finally, delete all the old v-nodes and edges adjacent to them.

The branching program constructed so far is equivalent to B_ℓ. In fact, the probability to reach a node w_k has not changed and, in particular, the error probability is the same as in B_ℓ, i.e., at most ϵ.

Merging v-nodes and w-nodes. Now we have two consecutive layers of probabilistic nodes. We can merge the layer of w-nodes into the v-nodes and change the probabilities on the edges going out from v-nodes appropriately such that the previous probabilities to reach a node in the layer below the w-nodes is not altered. Then we still have $2(W+1)$ v-nodes and no more w-nodes. Let B'_ℓ be the branching program constructed so far.

A problem and its solution. If we would just use this process to eliminate all the probabilistic nodes from B_ℓ, we might end up with an exponential number of nodes in the final branching program. So we have to reduce the number of v-nodes in B'_ℓ.

The idea now is to use the fact that we only want to know whether B_ℓ *is satisfiable*: we can throw out some v-nodes from B'_ℓ as long as we can guarantee that the reduced branching program is still satisfiable if B'_ℓ is satisfiable.

We show that we can reduce the number of v-nodes to at most $W+1$. Then the resulting branching program, $B_{\ell+1}$, fulfills the invariants (I1), ..., (I4) given in the outline of the construction above. In particular, the width of $B_{\ell+1}$ is bounded by $W+1$. Hence this completes the description of the algorithm.

Reducing the number of v-nodes. Let us rename the v-nodes back to $v_1, \dots, v_{2(W+1)}$ and let $q_{i,k}$ be the probability to reach the k-th node in the next layer from node v_i. Recall that *there are no probabilistic nodes above the v-nodes*. Therefore, B'_ℓ is satisfiable iff there is a v-node v_i such that the acceptance probability of v_i on some input, is at least $1-\epsilon$. Also, each v-node has acceptance probability of either at least $1-\epsilon$ or at most ϵ on any input.

With node v_i we associate a point $\boldsymbol{q}_i = (q_{i,1}, \dots, q_{i,W})$ in the W-dimensional vector space over the rationals, $\mathbf{Q}^W$. Let $\boldsymbol{a}$ be some input to B'_ℓ that reaches node v_i and let $\boldsymbol{y_a} = (y_1, \dots, y_W) \in \mathbf{Q}^W$ be the acceptance probabilities of $\boldsymbol{a}$ when starting at the nodes in the layer below the v-nodes. Then we have

$$\mathbf{Pr}[B'_\ell \text{ accepts } \boldsymbol{a}] \quad = \quad \boldsymbol{q}_i \cdot \boldsymbol{y_a}.$$

We are seeking for an $\boldsymbol{a}$ that is accepted by B'_ℓ with probability at least $1-\epsilon$. The trick now is to relax the condition a bit: instead of $\boldsymbol{a}$, we try to find $\boldsymbol{y} \in (\mathbf{Q} \cap [0,1])^W$ such that $\boldsymbol{q}_i \cdot \boldsymbol{y} \geq 1-\epsilon$. Note that $\boldsymbol{y}$ might not occur as a probability vector $\boldsymbol{y_a}$ for any input $\boldsymbol{a}$ that reaches node v_i. Therefore we call $\boldsymbol{q}_i \cdot \boldsymbol{y}$ a *pseudo-acceptance probability*. Now we take $\boldsymbol{y}$ as an unknown and try to find out whether

> *there exists a $\boldsymbol{y}$ such that v_i is* the only *v-node that has pseudo-acceptance probability* $1-\epsilon$.

If this is *not* the case, then

- either no v-node has pseudo-acceptance probability $1-\epsilon$ (including v_i) *on any input*,

- or there is an input on which both v_i *and some other* v*-node* have pseudo-acceptance probability $1-\epsilon$ with respect to some $\boldsymbol{y}$.

In both cases we can safely delete the node v_i from B'_ℓ and still maintain the property that the resulting branching program is satisfiable if and only if B'_ℓ is: this is simply because the range of $\boldsymbol{y}$ includes all the actually appearing probabilities $\boldsymbol{y_a}$.

A v-node v_i is the only one with pseudo-acceptance probability $1-\epsilon$ if the following system of linear inequalities has a solution $\boldsymbol{y} \in \mathbf{Q}^W$:

$$\begin{array}{rcll} \boldsymbol{q}_i \cdot \boldsymbol{y} & \geq & 1-\epsilon, & \\ \boldsymbol{q}_j \cdot \boldsymbol{y} & \leq & \epsilon, & \text{for } j \neq i, \text{ and} \\ 0 \leq y_k & \leq & 1, & \text{for } 1 \leq k \leq W. \end{array}$$

After deleting a v-node for which this set of inequalities has no solution, we repeat the above process again for other v-nodes until the above set of inequalities has a solution for every remaining v-node. In the ideal case we are left with just one v-node (if B is satisfiable). But since y can take values other than the actually occurring probabilities, there might be more v-nodes left. We now show that the number of remaining v-nodes can be at most $W+1$.

Bounding the number of the remaining v-nodes. Let v_1, …, v_r be the remaining v-nodes, $r \leq 2(W+1)$, with associated points $\boldsymbol{q}_1$, …, $\boldsymbol{q}_r$, and let furthermore $\boldsymbol{y}_i$ be a vector that satisfies the above set of inequalities for v_i.

Consider the set $Q = \{\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_r\}$. We claim that Q is *affinely independent* in $\mathbf{Q}^W$, that is, the points in Q span a $r-1$ dimensional subspace of $\mathbf{Q}^W$ (see [Grü67] for a reference). Since there can be at most $W+1$ affinely independent points in $\mathbf{Q}^W$, it follows that $r \leq W+1$.

By Lemma 4.65 below, to prove our claim that Q is affinely independent, it suffices to show that for any $S \subseteq Q$ there exists an affine plane that separates S from $Q-S$ (i.e., the points in S lie on one side of the plane whereas the points in $Q-S$ lie on the other). We can assume that $|S| \leq r/2$ (otherwise replace S by $Q-S$).

The affine plane can be defined as the set of points $\boldsymbol{x} \in \mathbf{Q}^W$ that fulfill the equation

$$\begin{array}{rcl} \boldsymbol{h}_S \cdot \boldsymbol{x} & = & 1 - \dfrac{1}{W+2}, \text{ where} \\ \boldsymbol{h}_S & = & \displaystyle\sum_{q_j \in S} \boldsymbol{y}_j. \end{array}$$

For any point $\boldsymbol{q}_i \in S$ we have:

$$\boldsymbol{h}_S \cdot \boldsymbol{q}_i \;\geq\; \boldsymbol{y}_i \cdot \boldsymbol{q}_i \;\geq\; 1-\epsilon \;>\; 1-\frac{1}{W+2}.$$

For any point $\boldsymbol{q}_i \in Q - S$ we have:

$$\begin{aligned}
\boldsymbol{h}_S \cdot \boldsymbol{q}_i &= \sum_{q_j \in S} \boldsymbol{y}_j \cdot \boldsymbol{q}_i \\
&\leq \epsilon\,|S| \\
&\leq \epsilon\,\frac{r}{2} \\
&\leq \epsilon\,(W+1) \\
&< \frac{W+1}{W+2} \\
&= 1-\frac{1}{W+2}.
\end{aligned}$$

This proves our claim.

The running time. To see that our algorithm runs in polynomial time, we note that the above system of linear inequalities can be solved in polynomial-time using Khachian's algorithm [Kha79].

A more subtle point we have to take care of is the size of the probability numbers that we write on the edges of our branching programs: they should be represented with only polynomially many bits (in the size of BP-OBDD B).

Because all numbers are between zero and one, it suffices to bound the denominators. In the beginning, in the given BP-OBDD, all probabilities are of the form $1/m$, where $m \leq W$. Let $p_1, \ldots, p_t$ be all primes up to W. If prime p_i occurs in the prime factorization of an m as above, then its exponent is bounded by $\log_{p_i} W$. We show that in the prime factorization of the denominators at the end of the construction, the exponent of prime p_i is bounded by $2n \log_{p_i} W$. Since we consider only $t \leq W$ primes, it follows that the denominators are bounded by W^{2nW}, which can be represented with polynomially many bits.

In each round, there are two steps where we change the probabilities:

- in the sum in equation (4.6). The new denominator is the least common multiple of all the denominators in that sum. Considering its prime factorization, the upper bound on the exponent of each prime remains the same as before.

- when we merge the v-nodes and the w-nodes, we have to multiply two probabilities and add, maybe several. Just as before, addition doesn't matter. For a multiplication, one factor is a probability from a v-node to a w-node which we have already changed, the other factor comes from the branching probability of a w-node which is still from the input branching program B. Hence, a multiplication may add another $\log_{p_i} W$ to the current exponent of prime p_i.

Since B has depth $2n$, we have up to $2n$ rounds of this construction. Therefore, the exponent of prime p_i is bounded by $2n \log_{p_i} W$ as claimed above. We conclude that our algorithm runs in polynomial time.

Now Lemma 4.65 below completes the proof of the theorem. □

Lemma 4.65 *Let $Q \subseteq \mathbf{Q}^m$ such that for every $S \subseteq Q$ there is an affine plane $\boldsymbol{h}_S$ that separates S from $Q - S$. Then Q is affinely independent.*

Proof Let $Q = \{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_r\}$. By definition, Q is affinely dependent if there exist $\lambda_1, \ldots, \lambda_r$ that are *not* all zero, such that $\sum_{i=1}^r \lambda_i\ \boldsymbol{q}_i = \mathbf{0}$ and $\sum_{i=1}^r \lambda_i = 0$.

We embed Q in $\mathbf{Q}^{m+1}$ by mapping $\boldsymbol{q}_i$ to $\boldsymbol{q}'_i = (\boldsymbol{q}_i, 1)$. Let Q' denote the embedding of Q in $\mathbf{Q}^{m+1}$. Then we have that Q is affinely dependent iff Q' *linearly* dependent.

Corresponding to an affine plane that separates S from $Q - S$, we now have a *hyperplane* that separates S' from $Q' - S'$. Namely, for the affine plane $\boldsymbol{h}_S \cdot \boldsymbol{x} = d$ we take the hyperplane $(\boldsymbol{h}_S, -d) \cdot \boldsymbol{x} = 0$.

Now, suppose that Q' is linearly dependent. This implies that there are $\lambda_1, \ldots, \lambda_r \in \mathbf{Q}$ such that not all of them are zero and

$$\sum_{i=1}^{r} \lambda_i\ \boldsymbol{q}'_i \quad = \quad \mathbf{0}. \tag{4.7}$$

Let S' be the set of those $\boldsymbol{q}'_i$ such that $\lambda_i \geq 0$ and let $\boldsymbol{h}_{S'}$ be the hyperplane that separates S' from $Q' - S'$. By equation (4.7) we have

$$0 \quad = \quad \boldsymbol{h}_{S'} \cdot (\sum_{i=1}^{r} \lambda_i\ \boldsymbol{q}'_i) \tag{4.8}$$

$$\quad = \quad \sum_{i=1}^{r} \lambda_i\ (\boldsymbol{h}_{S'} \cdot \boldsymbol{q}'_i) \tag{4.9}$$

Now observe that each term in equation (4.9) is non-negative and at least one term is non-zero. To see this note that if $\lambda_i \geq 0$ then $\boldsymbol{q}'_i \in S'$ and hence $\boldsymbol{h}_{S'} \cdot \boldsymbol{q}'_i > 0$. On the other hand, if $\lambda_i < 0$ then $\boldsymbol{q}'_i \in Q' - S'$ and hence $\boldsymbol{h}_{S'} \cdot \boldsymbol{q}'_i < 0$. Therefore this latter sum cannot be zero and we have a contradiction. We conclude that Q must be affinely independent. □

Note that we cannot push a BP-OBDD with too large error into the range of Theorem 4.64 by the standard amplification technique. This is because there we use the cross-product construction which increases the width of the resulting BP-OBDD. In the specific BP-OBDDs presented in Section 4.6.2 and Theorem 4.60 there is an alternative way of amplification: we could choose more primes. But again this increases the width of the resulting BP-OBDD. More precisely, consider the BP-OBDDs that verify multiplication or binary quadratics. For each prime p, the sub-OBDD that verifies the equation modulo p has width p^2. So if we verify equations modulo the first k primes $p_1, \ldots, p_k$, then the width of the resulting BP-OBDD is

$$W = \sum_{i=1}^{k} p_i^2 \geq k^3.$$

The error is $\Theta(n/k)$. This is somewhat larger than $1/W^{1/3}$. This shows that neither the above polynomial-time algorithm can be generalized to significantly larger error bounds, nor can the NP-completeness proof be generalized to significantly smaller errors, unless $\mathbf{P} = \mathbf{NP}$. It remains an open question to settle the satisfiability problem for errors in the range $O(\frac{1}{W^{1/3}}) \cap \omega(\frac{1}{W})$.

The equivalence problem for BP-OBDDs can be reduced to the satisfiability problem: let B_0 and B_1 be two BP-OBDDs, then $B_0 \not\equiv B_1$ iff $B = B_0 \oplus B_1$ is satisfiable. B can be constructed by Lemma 4.53. Therefore we also get an efficient algorithm for (the promise version of) the equivalence problem for BP-OBDDs of small error. Note that the width of B is bounded by the product of the widths of B_0 and B_1 and the errors sum up.

Corollary 4.66 *The equivalence problem for BP-OBDDs of width W with error ϵ is in* $\mathbf{P}$*, provided that* $2\epsilon < 1/(W^2 + 2)$.

References

[Abl97] F. Ablayev. Randomization and nondeterminism are incomparable for ordered read-once branching programs. In *24rd International Colloquium on Automata Languages and Programming*, Lecture Notes in Computer Science 1256, pages 195–202. Springer-Verlag, 1997.

[Agr97] M. Agrawal. Personal Communication, 1997.

[AK96] F. Ablayev and M. Karpinski. On the power of randomized branching programs. In *23rd International Colloquium on Automata Languages and Programming*, Lecture Notes in Computer Science 1099, pages 348–356. Springer-Verlag, 1996.

[AK98] F. Ablayev and M. Karpinski. A lower bound for integer multiplication on randomized read-once branching programs. Technical Report TR98-011, Electronic Colloquium on Computational Complexity, http://www.eccc.uni-trier.de/eccc/, 1998.

[AT96] M. Agrawal and T. Thierauf. The boolean isomorphism problem. In *37th Symposium on Foundation of Computer Science*, pages 422–430. IEEE Computer Society Press, 1996. To appear in SIAM Journal on Computing.

[AT98] M. Agrawal and T. Thierauf. The satisfiability problem for probabilistic ordered branching programs. In *13th IEEE Conference on Computational Complexity*, pages 81–90, 1998.

[BCG$^+$96] N. Bshouty, R. Cleve, R. Gavaldà, S. Kannan, and C. Tamon. Oracles and queries that are sufficient for exact learning. *Journal of Computer and System Sciences*, 52:421–433, 1996.

[BCW80] M. Blum, A. Chandra, and M. Wegman. Equivalence of free Boolean graphs can be decided probabilistically in polynomial time. *Information Processing Letters*, 10:80–82, 1980.

[BDG88] J. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity Theory I.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1988.

[BDG91] J. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity Theory II.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1991.

[Ber70] E. Berlekamp. Factoring polynomials over large finite fields. *Mathematics of Computation*, 24:713–735, 1970.

[BG82] A. Blass and Y. Gurevich. On the unique satisfiability problem. *Information and Computation*, 55:80–88, 1982.

[BGP98] M. Bellare, O. Goldreich, and E. Petrank. Uniform generation of NP-witnesses using an NP-oracle. Technical Report TR98-033, Electronic Colloquium on Computational Complexity, http://www.eccc.uni-trier.de/eccc/, 1998.

[BHR95] Y. Breitbart, H. Hunt, and D. Rosenkrantz. On the size of binary decision diagrams representing Boolean functions. *Theoretical Computer Science*, 145:45–69, 1995.

[BHST87] L. Babai, A. Hajnal, E. Szemeredi, and G. Turan. A lower bound for read-once-only branching programs. *Journal of Computer and System Sciences*, 35:153–162, 1987.

[BHZ87] R. Boppana, J. Håstad, and S. Zachos. Does co-NP have short interactive proofs? *Information Processing Letters*, 25:27–32, 1987.

[BM88] L. Babai and S. Moran. Arthur-merlin games: A randomized proof system, and a hierarchy of complexity classes. *Journal of Computer and System Sciences*, 36:254–276, 1988.

[Bor97] B. Borchert. Personal Communication, 1997.

[BR93] B. Borchert and D. Ranjan. The subfunction relations are Σ_2^p-complete. Technical Report MPI-I-93-121, MPI Saarbrücken, 1993.

[BRS93] A. Borodin, A. Razborov, and R. Smolensky. On lower bounds for read-k-times branching programs. *Computational Complexity*, 3:1–18, 1993.

[BRS98] B. Borchert, D. Ranjan, and F. Stephan. On the computational complexity of some classical equivalence relations on boolean functions. *Theory of Computing Systems*, 31:679–693, 1998.

[Bry86] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transaction on Computers*, 35(6):677–691, 1986.

[Bry91] R. Bryant. On the complexity of VLSI implementations and graph representation of Boolean functions with applications to integer multiplication. *IEEE Transaction on Computers*, 40(2):205–213, 1991.

[Bry92] R. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[BSSW98] B. Bollig, M. Sauerhoff, D. Sieling, and I. Wegener. Hierarchy theorems for k-OBDDs and k-IBDDs. *Theoretical Computer Science*, 205:45–60, 1998.

[BT96] H. Buhrman and T. Thierauf. The complexity of generating and checking proofs of membership. In *13th Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science 1046, pages 75–87. Springer-Verlag, 1996.

[Bus97] S. Buss. Alogtime algorithms for tree isomorphism, comparison, and canonization. In *Kurt Gödel Colloquium*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[BW96] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1001, 1996.

[BW97] B. Bollig and I. Wegener. Complexity theoretical results on partioned (nondeterministic) binary decision diagrams. In *Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 1295, pages 159–168. Springer-Verlag, 1997.

[BW98] J. Behrens and S. Waack. Equivalence test and ordering transformation for parity-OBDDs of different variable ordering. In *15th Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science 1373, pages 227–237. Springer-Verlag, 1998.

[CK91] P. Clote and E. Kranakis. Boolean functions, invariance groups, and parallel complexity. *SIAM Journal on Computing*, 20:553–590, 1991.

[CK97] Z.Z. Chen and M.Y. Kao. Reducing randomness via irrational numbers. In *ACM Symposium on Theory of Computing*, pages 200–209, 1997.

[CKR95] R. Chang, J. Kadin, and P. Rohatgi. On unique satisfiability and the threshold behavior of randomized reductions. *Journal of Computer and System Sciences*, 50:359–373, 1995.

[Cli76] W. Clifford. On the types of compound statement involving four classes. *Proceedings of the Literary and Philosophical Society of Manchester*, 16(7):88–101, 1876. Also in W. Clifford. *Mathematical Papers*. London, 1–16, 1882.

[Cob66] A. Cobham. The recognition problem for the set of perfect suares. Technical Report RC-1704, IBM Watson Research Centre, 1966.

[Coo71] S. Cook. The complexity of theorem-proving procedures. In *3rd ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[CW79] J. Carter and M. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[FHS78] S. Fortune, J. Hopcroft, and E. Schmidt. The complexity of equivalence and containment for free single variable program schemes. In *5th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science 62, pages 227–240. Springer-Verlag, 1978.

[Fre77] R. Freivalds. Probabilistic machines can use less running time. In B. Gilchrist, editor, *Information Processing 77*, pages 839–842, Amsterdam, 1977. North-Holland.

[FS88] L. Fortnow and M. Sipser. Are there interactive protocols for co-NP languages? *Information Processing Letters*, 28:249–251, 1988.

[GM94] J. Gergov and C. Meinel. Efficient Boolean mainpulation with OBDD's can be extended to FBDD's. *IEEE Transaction on Computers*, 43(10):1197–1209, 1994.

[GM96] J. Gergov and C. Meinel. Mod-2-OBDDs - a data structure that generalizes EXOR-sum-of-products and ordered binary decision diagrams. *Formal Methods in System Design*, 8:273–282, 1996.

[GMR89] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18:186–208, 1989.

[GMW91] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38:691–729, 1991.

[Grü67] B. Grünbaum. *Convex Polytopes.* Interscience Publishers (John Wiley & Sons), 1967.

[GS89] S. Goldwasser and M. Sipser. Private coins versus public coins in interactive proof systems. *Advances in Computing Research*, 5:73–90, 1989.

[GZ97] O. Goldreich and D. Zuckerman. Another proof that BPP $\subseteq$ PH (and more). Technical Report TR97-045, Electronic Colloquium on Computational Complexity, http://www.eccc.uni-trier.de/eccc/, 1997.

[Har66] M. Harrison. On asymptotic estimates in switching and automata theory. *Journal of the ACM*, 13(1):151–157, 1966.

[Har71] M. Harrison. Counting theorems and their application to classification of switching functions. In A. Mukhopadyay, editor, *Recent Developments in Switching Theory*, chapter 4. Academic Press, 1971.

[Hof82] C. Hoffmann. *Group-Theoretic Algorithms and Graph Isomorphism.* Lecture Notes in Computer Science 136. Springer-Verlag, 1982.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, Mass., USA, 1979.

[IM83] O. Ibarra and S. Moran. Probabilistic algorithms for deciding equivalence of straight-line programs. *Journal of the ACM*, 30:217–228, 1983.

[JBFA92] J. Jain, J. Bitner, D. Fussell, and J. Abraham. Functional partitioning for verification and related problems. In *Brown/MIT VLSI Conference*, pages 220–226, 1992.

[Jev92] W. Jevons. *The Principles of Science*, pages 134–146. 1892. First published in *Proceedings of the Manchester Literary and Philosophical Society*, 9:65–68, 1871.

[JLM97] B. Jenner, K. Lange, and P. McKenzie. Tree isomorphism and some other complete problems for deterministic logspace. Technical Report 1059, Université de Montréal, 1997.

[Joh90] D. Johnson. A catalog of complexity classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.* The MIT Press and Elsevier, 1990.

[Juk89] S. Jukna. The effect of null-chains on the complexity of contact schemes. In *7th Fundamentals of Computation Theory*, Lecture Notes in Computer Science 380, pages 246–256. Springer-Verlag, 1989.

[Juk95] S. Jukna. The graph of multiplication is hard for read-k-times networks. Technical Report 95-10, Universität Trier, Fachbereich Mathematik/Informatik, 1995.

[JVV86] M. Jerrum, L. Valiant, and V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science*, 43:169–188, 1986.

[Kha79] L. Khachian. A polynomial algorithm in linear programming. *Russian Academy of Sciences Doklady. Mathematics (formerly Soviet Mathematics–Doklady)*, 20:191–194, 1979.

[KL82] R. Karp and R. Lipton. Turing machines that take advice. *L'Enseignement Mathématique*, 28:191–209, 1982.

[KMW91] M. Krause, C. Meinel, and S. Waack. Separating the eraser Turing machine classes L_e, NL_e, and P_e. *Theoretical Computer Science*, 86:267–275, 1991.

[Kra91] M. Krause. Lower bounds for depth-restricted branching programs. *Information and Computation*, 91:1–14, 1991.

[KST93] H. Köbler, U. Schöning, and J. Torán. *The Graph Isomorphism Problem: its Structural Complexity.* Birkhäuser, 1993.

[Lau83] C. Lautemann. BPP and the polynomial hierarchy. *Information Processing Letters*, 17:215–217, 1983.

[Lee59] C. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4):985–999, 1959.

[LFKN92] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM*, 39:859–868, 1992.

[Lin92] S. Lindell. A logspace algorithm for tree canonization. In *24th ACM Symposium on Theory of Computing*, pages 400–404, 1992.

[LSW98] M. Löbbing, D. Sieling, and I. Wegener. Parity OBDDs cannot be handled efficiently enough. *Information Processing Letters*, 67:163–168, 1998.

[MA78] K. Manders and L. Adleman. NP-complete decision problems for binary quadratics. *Journal of Computer and System Sciences*, 16:168–184, 1978.

[Mas76] W. Masek. A fast algorithm for the string editing problem and decision graphs complexity. Master's thesis, Massachusetts Institute of Technology, 1976.

[Mei89] C. Meinel. *Modified Branching Programs and Their Computational Power.* Lecture Notes in Computer Science 370. Springer-Verlag, 1989.

[MS72] A. Meyer and L. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *13th IEEE Symposium on Switching and Automata Theory*, pages 125–129. IEEE Computer Society Press, 1972.

[MS94] C. Meinel and A. Slobodová. On the complexity of constructing optimal ordered binary decision diagrams. In *Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 841, pages 515–524. Springer-Verlag, 1994.

[MS97] C. Meinel and H. Sack. Case study: Manipulating $\oplus$-OBDDs by means of signatures. Technical Report 97-15, Universität Trier, Fachbereich Mathematik/Informatik, 1997.

[NJFSV96] A. Narayan, J. Jain, M. Fujita, and A. Sangiovanni-Vincentelli. Partitioned ROBDDs - a compact, canonical and efficient manipulable representation for Boolean functions. In *ICCAD*, pages 547–554, 1996.

[Pap94] Christos Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[Pól37] G. Pólya. Kombinatorische Anzahlbestimmungen für Gruppen, Graphen und chemische Verbindungen. *Acta Mathematica*, 68:145–254, 1937.

[Pól40] G. Pólya. Sur les types des propositions composées. *Journal of Symbolic Logic*, 5:98–103, 1940.

[Pon95] S. Ponzio. *Restricted Branching Programs and Hardware Verification.* PhD thesis, Massachusetts Institute of Technology, 1995.

[PZ83a] C. Papadimitriou and D. Zachos. Two remarks on the power of counting. In *6th GI Conference on Theoretical Computer Science*, Lecture Notes in Computer Science 176, pages 269–276. Springer-Verlag, 1983.

[PZ83b] P. Pudlák and S. Zák. Space complexity of computation. Technical report, University of Prague, 1983.

[Sau98] M. Sauerhoff. Lower bounds for randomized read-k-times branching programs. In *15th Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science 1373, pages 105–115. Springer-Verlag, 1998.

[Sav98] P. Savický. A probabilistic nonequivalence test for syntactic $(1, +k)$-branching programs. Technical Report TR98-051, Electronic Colloquium on Computational Complexity, http://www.eccc.uni-trier.de/eccc/, 1998.

[Sch80] J. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, 27:701–717, 1980.

[Sch85] U. Schöning. *Complexity and Structure.* Lecture Notes in Computer Science 211. Springer-Verlag, 1985.

[Sch88] U. Schöning. Graph isomorphism is in the low hierarchy. *Journal of Computer and System Sciences*, 37:312–323, 1988.

[Sch89] U. Schöning. Probabilistic complexity classes and lowness. *Journal of Computer and System Sciences*, 39:84–100, 1989.

[Sch95] U. Schöning. *Perlen der Theoretischen Informatik.* BI Wissenschaftverlag, 1995.

[Sha49] C. Shannon. The synthesis of two-terminal switching circuits. *Bell System Technical Journal*, 28:59–98, 1949.

[Sha92] A. Shamir. IP = PSPACE. *Journal of the ACM*, 39:869–877, 1992.

[Sie98] D. Sieling. A separation of syntactic and nonyntactic $(1, +k)$-branching programs. Technical Report TR98-045, Electronic Colloquium on Computational Complexity, http://www.eccc.uni-trier.de/eccc/, 1998.

[Sip83] M. Sipser. A complexity theoretic approach to randomness. In *15th ACM Symposium on Theory of Computing*, pages 330–335, 1983.

[Sip92] M. Sipser. The history and status of the P versus NP question. In *24th ACM Symposium on Theory of Computing*, pages 603–618, 1992.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.

[SS93] J. Simon and M. Szegedy. A new lower bound theorem for read-only-once branching programs and its applications. In J.Y. Cai, editor, *Advances in Computational Complexity Theory*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 13, pages 183–193. American Mathematical Society, 1993.

[Sto85] L. Stockmeyer. On approximation algorithms for #P. *SIAM Journal on Computing*, 12(4):849–861, 1985.

[SW93] D. Sieling and I. Wegener. Reduction of OBDDs in linear time. *Information Processing Letters*, 48:139–144, 1993.

[SW95] D. Sieling and I. Wegener. Graph driven BDDs - a new data-structure for Boolean functions. *Theoretical Computer Science*, 141:283–310, 1995.

[SW97] P. Savický and I. Wegener. Efficient algorithms for the transformation between different types of binary decision diagrams. *Acta Informatica*, 34:245–256, 1997.

[Thi98] T. Thierauf. The isomorphism problem for read-once branching programs and arithmetic circuits. *Chicago Journal of Theoretical Computer Science*, 1998(1), 1998.

[Tod91] S. Toda. PP is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, 20:865–877, 1991.

[Uma98] C. Umans. The minimum equivalent DNF problem and shortest implicants. In *39th Symposium on Foundation of Computer Science*, pages 556–563. IEEE Computer Society Press, 1998.

[VV86] L. Valiant and V. Vazirani. NP is as easy as detecting unique solutions. *Theoretical Computer Science*, 47:85–93, 1986.

[Ž84] S. Žák. An exponential lower bound for one-time-only branching programs. In *11th Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 176, pages 562–566. Springer-Verlag, 1984.

[Waa97] S. Waack. On the descriptive and algorithmic power of parity ordered binary decision diagrams. In *14th Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science 1200, pages 201–212. Springer-Verlag, 1997.

[Weg86] I. Wegener. *The Complexity of Boolean Functions.* Series in Computer Science. Wiley-Teubner, 1986.

[Weg88] I. Wegener. On the complexity of branching programs and decision trees for clique functions. *Journal of the ACM*, 35:461–471, 1988.

[Weg94] I. Wegener. Efficient data structures for Boolean functions. *Discrete Mathematics*, 136:347–372, 1994.

[Zip79] R. Zippel. Probabilistic algorithms for sparse polynomials. In *International Symposium on Symbolic and Algebraic Computation*, Lecture Notes in Computer Science 72, pages 216–226. Springer-Verlag, 1979.

[Zuc96] D. Zuckerman. Simulating BPP using a general weak random source. *Algorithmica*, 16:367–391, 1996.

Index