Some Results in Computational Complexity

by

Ali Juma

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © 2005 by Ali Juma

Abstract

Some Results in Computational Complexity

Ali Juma

Master of Science

Graduate Department of Computer Science

University of Toronto

2005

In this thesis, we present some results in computational complexity. We consider two approaches for showing that #P has polynomial-size circuits. These approaches use ideas from the interactive proof for #3-SAT. We show that these approaches fail. We discuss whether there are instance checkers for languages complete for the class of approximate counting problems. We provide evidence that such instance checkers do not exist. We discuss the extent to which proofs of hierarchy theorems are constructive. We examine the problems that arise when trying to make the proof of Fortnow and Santhanam's nonuniform BPP hierarchy theorem more constructive.

Acknowledgements

First, I would like to thank my supervisor, Charles Rackoff. Working with Charlie has been an intellectually stimulating and enjoyable experience. I greatly appreciate the many hours that Charlie spent explaining new concepts to me and suggesting ideas for this thesis.

I would like to thank my second reader, Stephen Cook, for his valuable comments and suggestions.

I would like to thank my parents for their constant encouragement and support.

Finally, I would like to thank the Natural Sciences and Engineering Research Council of Canada for providing financial support.

Contents

1 Introduction **1**

2 Does #P have polynomial-size circuits? **5**

2.1 The class #P 5

2.2 Interactive proofs 6

2.2.1 Arithmetizing propositional formulas 8

2.2.2 An interactive proof for #3-SAT 11

2.3 The linear combination approach 13

2.4 The $2n$-variable polynomial approach 17

2.5 The small low-degree descriptor approach 21

2.6 Computing the coefficients of a polynomial 24

3 Are there instance checkers for approximate counting? **30**

3.1 Approximate counting 30

3.2 Instance checkers 32

3.3 Approximate polynomial evaluation 36

3.3.1 Approximating a value using an oracle for approximate counting . 36

3.3.2 Approximating a new value given a set of approximations 39

4 How constructive are proofs of hierarchy theorems? **43**

4.1 Hierarchy theorems 43

4.2 The Time Hierarchy theorem 44
4.3 Tighter time hierarchy results 45
4.4 A hierarchy theorem for probabilistic polynomial time with small advice . 47
4.4.1 Proof of Theorem 4.4.1 48
4.4.2 Making the proof of Theorem 4.4.1 more constructive 55

5 Open questions **60**

Bibliography **62**

Chapter 1

Introduction

Computational complexity is the study of the relationship between the resources available to a computational model and the problems that can be solved by this model. Typically, the resources considered include time, space, randomness, nondeterminism, and nonuniformity. Problems that can be solved using a specific set of resources are grouped together to form a *complexity class*. In this thesis, we will present some results in computational complexity.

The class P is the set of languages accepted by polynomial-time Turing machines, and the class NP is the set of languages accepted by nondeterministic polynomial-time Turing machines. The most important open question in computer science today is whether $\mathsf{P} = \mathsf{NP}$. This question has been studied for over 30 years, without resolution. In [KL82], Karp and Lipton suggest a possible approach for showing $\mathsf{P} \neq \mathsf{NP}$. They show that if every language in EXP, the class of languages accepted by exponential-time Turing machines, is accepted by a family of polynomial-size circuits, then $\mathsf{P} \neq \mathsf{NP}$.

The class $\#\mathsf{P}$ is a class of function problems. Informally, problems in $\#\mathsf{P}$ involve counting the number of different solutions to a question. The problem $\#\mathsf{SAT}$ (computing the number of satisfying assignments for a formula of the propositional calculus) is complete for $\#\mathsf{P}$. Every language in $\mathsf{P}^{\#\mathsf{P}}$ (the class of languages that can be decided in

polynomial time given access to an oracle for a #P-complete problem) has an *interactive proof* [LFKN92]. This means that languages in $\mathsf{P}^{\#\mathsf{P}}$ are "easy" when interaction is allowed; that is, each language in $\mathsf{P}^{\#\mathsf{P}}$ can be decided by a probabilistic polynomial-time verifier that interacts with a computationally unbounded prover. Furthermore, the interactive proofs for $\mathsf{P}^{\#\mathsf{P}}$ are the simplest known non-trivial interactive proofs for NP-hard languages. It is plausible that such interaction can be converted into nonuniformity. If this is the case, then #P has polynomial-size circuits. Since $\mathsf{P}^{\#\mathsf{P}} \subseteq \mathsf{EXP}$, and since every language in EXP has a two-prover interactive proof [BFL91], showing that #P has polynomial-size circuits would be a significant first step toward showing that EXP has polynomial-size circuits.

In Chapter 2, we consider two related approaches for showing that #P has polynomial-size circuits. Like the interactive proof for #SAT, these approaches both rely on the idea of *arithmetizing* propositional formulas. Given a propositional formula, we can construct a multivariable polynomial of degree at most two in each variable, such that for every $\{0, 1\}$-valued assignment to the polynomial's variables, the polynomial evaluates to 1 if and only if the corresponding truth assignment satisfies the formula, and the polynomial evaluates to 0 otherwise.

The first approach we consider is showing that for every $2n$-variable polynomial of degree at most two in each variable, there exists a polynomial-size (in n) circuit that, given oracle access to the polynomial and given values for the first n variables, sums the polynomial over all $\{0, 1\}$-valued assignments to the remaining n variables. Intuitively, the first n variables are used to specify a propositional formula, and the remaining n variables are used to specify a truth assignment to that formula. We show that, in fact, there are polynomials for which such a circuit does not exist.

Note that in the first approach, we consider arbitrary polynomials of degree at most two in each variable. These polynomials do not necessarily correspond to arithmetizations of propositional formulas. The second approach we consider attempts to deal with this

by considering only a subset of polynomials of degree at most two in each variable. In particular, we consider those polynomials that have *small low-degree descriptors*. A small low-degree descriptor for an n-variable polynomial p is a low-degree polynomial in $\log(n)$ variables that can be used to efficiently compute any coefficient of p. We show that for each n there exists a polynomial-size circuit that, given oracle access to any n-variable polynomial that is of degree at most two in each variable and has a small low-degree descriptor, sums the polynomial over all $\{0,1\}$-valued assignments to its variables. However, we then show that there exist propositional formulas whose arithmetizations do not have small low-degree descriptors.

Polynomials with small low-degree descriptors have coefficients that (given the descriptor) are easy to compute. We show that, in general, computing a coefficient of a polynomial is hard. In particular, we show that given an expression in n variables and involving operations $\{+, -, \times\}$, such that the expression is of degree at most two in each variable, the problem of computing the coefficient of a given term is $\mathsf{P}^{\#\mathsf{P}}$-complete.

The class $\#\mathsf{P}$ involves *exact* counting. In Chapter 3, we investigate *approximate* counting. In an approximate counting problem, we specify an approximation parameter r and require that the answer c' be such that $c/r \leq c' \leq r \cdot c$, where c is the exact count.

Every $\mathsf{P}^{\#\mathsf{P}}$-complete language and every language in P has an *instance checker*. Introduced by Blum and Kannan [BK95], an instance checker for a language L is an efficient procedure that can be used to verify the output of a Turing machine that purportedly decides L. Instance checkers are closely related to interactive proofs. It is not known if instance checkers exist for languages complete for any NP-hard natural class strictly contained in $\mathsf{P}^{\#\mathsf{P}}$ (of course, it is not even known if such a class exists). Assuming the *polynomial-time hierarchy* doesn't collapse, the class of approximate-counting problems is an NP-hard class that is strictly contained in $\mathsf{P}^{\#\mathsf{P}}$. We provide evidence that instance checkers do not exist for languages complete for the class of approximate counting problems.

Recently, the existence of instance checkers has been used to prove results about probabilistic computation. In particular, the existence of instance checkers has been used to prove a hierarchy theorem for probabilistic polynomial time with small advice. By the Time Hierarchy theorem, for any constant $d \geq 0$ there exists a language that can be decided in polynomial time but not in time n^d. An analogous theorem is not known for probabilistic polynomial time with bounded error (BPTIME). However, using an instance checker for an EXP-complete language, Barak [Bar02] shows that for any constant $d \geq 0$, there exists a language that can be decided in BPP (probabilistic polynomial time with bounded error) with $\log\log n$ bits of advice but not in $\mathsf{BPTIME}(n^d)$ with $\log\log n$ bits of advice. Fortnow and Santhanam [FS04] improve this result by reducing the advice from $\log\log n$ bits to one bit.

In Chapter 4, we discuss the extent to which proofs of hierarchy theorems are constructive. We consider such a proof to be constructive if rather than simply establishing that $\mathcal{C}_1 \subsetneq \mathcal{C}_2$ for some complexity classes $\mathcal{C}_1$ and $\mathcal{C}_2$, the proof presents a language L such that $L \in \mathcal{C}_2 - \mathcal{C}_1$. Furthermore, we require a constructive proof to establish how hard it is, given a Turing machine M that satisfies the resource bounds of class $\mathcal{C}_1$, to find an input x on which M differs from L. We present the proof of Fortnow and Santhanam's hierarchy theorem, and discuss the problems that arise when trying to make this proof more constructive.

Chapter 2

Does $\#\mathsf{P}$ have polynomial-size circuits?

In this chapter, we consider two related approaches for showing that the class $\#\mathsf{P}$ has polynomial-size circuits. We show that these approaches - the "$2n$-variable polynomial approach" and the "small low-degree descriptor approach" - do not work. Finally, we show that the problem of computing a coefficient of a multivariable polynomial given an expression for the polynomial is $\mathsf{P}^{\#\mathsf{P}}$-complete.

2.1 The class $\#\mathsf{P}$

The class $\#\mathsf{P}$ is a class of function problems. Each language $L \in \mathsf{NP}$ has a corresponding counting problem $\#L \in \#\mathsf{P}$. For each language $L \in \mathsf{NP}$, there is a polynomial-time computable witness relation R_L such that $x \in L$ iff there exists a string y of length polynomial in $|x|$ where $R_L(x, y)$ holds. Such a string y is known as a *witness* for x's membership in L. The problem $\#L$ is to compute, for an input x, the number of witnesses for x's membership in L. Note that the problem $\#L$ depends on the choice of witness relation R_L, and hence $\#L$ is not uniquely determined by L.

Equivalently, $\#\mathsf{P}$ can be defined in terms of counting the number of accepting com-

putation paths of a polynomial-time nondeterministic Turing machine.

Definition 2.1.1 (#P). Let $f : \Sigma^* \to \mathbb{N}$ be a function. $f \in$ #P iff there exists a polynomial-time nondeterministic Turing machine M such that for all $x \in \Sigma^*$, $f(x)$ is the number of accepting computation paths of M on input x.

The language SAT is the set of satisfiable formulas of the propositional calculus. The corresponding counting problem, #SAT, is to count the number of satisfying assignments for a propositional formula. #SAT is #P-complete, and remains #P-complete when restricted to formulas that are in 3-CNF (conjunctive normal form with at most three literals per clause).[1] This restricted problem is called #3-SAT.

2.2 Interactive proofs

The class NP is the set of languages L such that for each string $x \in L$, there is a polynomial-size proof of x's membership in L. The proof is simply a string (the witness) whose correctness can be efficiently checked by a verifier. Instead of requiring a proof to be a single string, what if we allow a prover and verifier to have a "conversation" by sending strings to each other? This leads to the idea of *interactive proofs*, introduced in [GMR89, BM88]. Every language in $\mathsf{P}^{\#\mathsf{P}}$ has an interactive proof [LFKN92]. In fact, interactive proofs for $\mathsf{P}^{\#\mathsf{P}}$-complete languages are the simplest known non-trivial interactive proofs for NP-hard languages. This makes the class $\mathsf{P}^{\#\mathsf{P}}$ a logical starting point for investigating the idea that interaction can be converted into nonuniformity.

In an interactive proof, a computationally unbounded prover P attempts to convince a probabilistic polynomial-time verifier V of the truth of a statement (for example, that some string x is in some language L). P and V exchange one or more messages, and at the end V either accepts or rejects. If the statement being proved is indeed true then V

[1] In fact, #SAT is #P-complete even when restricted to formulas in 2-CNF.

always accepts, and otherwise V almost always rejects.

Definition 2.2.1 (Interactive proof). Let L be a language. We say that L has an *interactive proof* if there exists a pair of interactive Turing machines (P, V) such that V is probabilistic polynomial-time and the following two properties hold:

1. (Completeness) For all $x \in L$, (P, V) accepts input x with probability 1.

2. (Soundness) For all (computationally unbounded) machines P' and for all $x \notin L$, say $|x| = n$, the probability that (P', V) accepts input x is at most $1/n^c$ for all c and sufficiently large n.

We call P the *prover* and we call V the *verifier*. IP is the class of languages that have interactive proofs. For a polynomially length-bounded function f, we say that f has an interactive proof if the language $L = \{(x, y) \mid f(x) = y\}$ has an interactive proof.

Every language $L \in \mathsf{NP}$ has a trivial interactive proof. On input $x \in L$, the prover P sends the verifier V a witness y for x's membership in L, and V simply checks that the witness relation $R_L(x, y)$ holds. If $x \notin L$ then no prover will be able to find a y that makes V accept.

Similarly, every language $L \in \mathsf{coRP}$ has a trivial interactive proof. A language L is in coRP if there exists a probabilistic polynomial-time Turing machine M such that for all inputs x, M accepts x with probability 1 if $x \in L$, and M rejects x with probability at least $1/2$ if $x \notin L$. On input x, the prover P does nothing and the verifier V runs L's coRP machine $|x|$ times on x, accepting iff every such run accepts. Clearly, if $x \in L$ then V always accepts and if $x \notin L$ then V accepts with probability at most $1/2^{|x|}$.

The known interactive proofs for all NP-hard classes contained in $\mathsf{P}^{\#\mathsf{P}}$ are either trivial (as in the case of NP) or use all the ideas used in interactive proofs for $\mathsf{P}^{\#\mathsf{P}}$-complete languages.[2] As an example of the latter case, the simplest known interactive

[2] The coNP language GRAPH-NONISOMORPHISM has a non-trivial interactive proof that does not

proof for TAUTOLOGIES, a coNP-complete language, involves showing that the number of satisfying assignments for a propositional formula F is exactly 2^n (where n is the number of variables in F) and hence this interactive proof uses all the ideas used in the interactive proof for #SAT. On the other hand, the known interactive proofs for PSPACE $\supseteq$ $\mathsf{P}^{\#\mathsf{P}}$ do require additional ideas that are not needed for $\mathsf{P}^{\#\mathsf{P}}$ [Sha92].

The original proof that $\mathsf{P}^{\#\mathsf{P}} \subseteq IP$ [LFKN92] used the #P-complete problem PERMANENT. Babai and Fortnow [BF91] then gave a proof using #3-SAT. Their interactive proof for #3-SAT is based on the idea of *arithmetizing* 3-CNF formulas. In section 2.2.1, we will show how to arithmetize a propositional formula, and in section 2.2.2, we will present the interactive proof for #3-SAT. Then, motivated by the ideas used in the interactive proof, we will investigate approaches for showing that #3-SAT (and hence #P) has polynomial-size circuits.

2.2.1 Arithmetizing propositional formulas

Babai and Fortnow [BF91] and Shamir [Sha92] introduce the idea of arithmetizing propositional formulas. They show how, given a 3-CNF formula F, we can efficiently construct a multivariable polynomial p such that for every $\{0,1\}$-valued assignment to p's variables, p evaluates to 1 iff the corresponding truth assignment satisfies F and p evaluates to 0 iff the corresponding truth assignment does not satisfy F.

Definition 2.2.2 (Arithmetization). Let F be a propositional formula on variables $x_1, \ldots, x_n$, and let p be a polynomial on variables $x_1, \ldots, x_n$. We say that p is an *arithmetization* of F if the following holds:

- Let $\tau : \{x_1, \ldots, x_n\} \to \{true, false\}$ be a truth assignment, and let $a_1, \ldots, a_n \in \{0,1\}$ be such that for each i, $a_i = 1$ iff $\tau(x_i) = true$. Then $p(a_1, \ldots, a_n) = 1$ if τ satisfies F, and $p(a_1, \ldots, a_n) = 0$ otherwise.

---

use the ideas used in interactive proofs for $\mathsf{P}^{\#\mathsf{P}}$-complete languages. However, this language is not known to be NP-hard.

Note that if p is an arithmetization of a propositional formula F, then the number of satisfying truth assignments for F is equal to the sum of p over all $\{0,1\}$-valued assignments to its variables. We will use the following notation for such a sum.

Notation 2.2.1 ($\mathcal{S}(p)$). Let p be a polynomial. Say p's variables are $x_1, \dots, x_n$. Then, $\mathcal{S}(p)$ denotes the sum $\sum_{(x_1,\dots,x_n)\in\{0,1\}^n} p(x_1,\dots,x_n)$.

We now show how to arithmetize a 3-CNF formula. Let $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ be a 3-CNF formula on variables $x_1, \dots, x_n$, where each C_i is a clause. We begin by arithmetizing each clause. So consider a clause C_i, and first consider the case where C_i has exactly three literals. Say $C_i = (\ell_{i_1} \vee \ell_{i_2} \vee \ell_{i_3})$, where each ℓ_{i_j} is a literal. For any literal ℓ, let $\bar{\ell}$ be its negation. Then, define formula C_i' as follows:

$$\begin{aligned} C_i' \;=\; & (\ell_{i_1} \wedge \ell_{i_2} \wedge \ell_{i_3}) \vee (\ell_{i_1} \wedge \ell_{i_2} \wedge \bar{\ell}_{i_3}) \vee (\ell_{i_1} \wedge \bar{\ell}_{i_2} \wedge \ell_{i_3}) \vee (\bar{\ell}_{i_1} \wedge \ell_{i_2} \wedge \ell_{i_3}) \\ & \vee (\ell_{i_1} \wedge \bar{\ell}_{i_2} \wedge \bar{\ell}_{i_3}) \vee (\bar{\ell}_{i_1} \wedge \ell_{i_2} \wedge \bar{\ell}_{i_3}) \vee (\bar{\ell}_{i_1} \wedge \bar{\ell}_{i_2} \wedge \ell_{i_3}) \end{aligned}$$

Now consider the case where C_i has exactly two literals, say $C_i = (\ell_{i_1} \vee \ell_{i_2})$. Then, define formula C_i' as follows:

$$C_i' = (\ell_{i_1} \wedge \ell_{i_2}) \vee (\ell_{i_1} \wedge \bar{\ell}_{i_2}) \vee (\bar{\ell}_{i_1} \wedge \ell_{i_2})$$

Finally, if C_i has exactly one literal, let $C_i' = C_i$. Note that in each case, C_i' is logically equivalent to C_i. To construct an arithmetization $p_i(x_1, \dots, x_n)$ from C_i', we proceed as follows:

- We replace each $\wedge$ with $\times$ (multiplication).
- We replace each $\vee$ with $+$ (addition).
- We replace each negated variable $\neg x_j$ with $(1 - x_j)$.

It is easy to see that p_i is an arithmetization of C_i' (and hence an arithmetization of C_i). The key idea is that any truth assignment that satisfies C_i' satisfies exactly one of its

$\wedge$-clauses. Then, $p = p_1 \cdot p_2 \cdot \ldots \cdot p_m$ is an arithmetization of F. This means that the number of truth assignments that satisfy F is exactly $\mathcal{S}(p)$.

Note that the degree of each variable in p will be equal to the number of times the variable appears in formula F. In fact, we can efficiently construct a polynomial p' of degree at most two in each variable such that the number of truth assignments that satisfy F is exactly $\mathcal{S}(p')$.

We begin by constructing from F a formula F' such that each variable appears at most three times in F', and such that F' has the same number of satisfying truth assignments as F. Initially, $F' = F$. Then, for each variable x_i, if x_i appears $k > 2$ times in F' we create k variables $x_{i,1}, \ldots, x_{i,k}$, and replace the j-th occurrence of x_i in F' with $x_{i,j}$. Also, if $k > 2$, we add the clauses $D_i = (x_{i,1} \rightarrow x_{i,2}) \wedge (x_{i,2} \rightarrow x_{i,3}) \wedge \ldots (x_{i,k-1} \rightarrow x_{i,k}) \wedge (x_{i,k} \rightarrow x_{i,1})$ to F'. Otherwise, we let D_i be the empty conjunction. Since any truth assignment that satisfies F' must set $x_{i,1}, \ldots, x_{i,k}$ to the same value, the number of truth assignments that satisfy F' is the same as the number of truth assignments that satisfy F.

Now, $F' = C_1 \wedge \cdots \wedge C_m \wedge D_1 \wedge \cdots \wedge D_n$. We arithmetize each C_i using the same method as before, and we call each such arithmetization p_i. Then, $p_1 \cdot p_2 \cdot \ldots \cdot p_m$ is a polynomial that is of degree at most one in each variable that appears in a D_i and of degree at most two in each variable that does not appear in a D_i. It remains to arithmetize each D_i. We need the arithmetization of each D_i to be of degree at most one in each variable. However, since each variable that appears in a D_i appears twice in that D_i, if we follow the method used to arithmetize the C_i then the arithmetization will be of degree two in each variable. Instead, we arithmetize each $D_i = (x_{i,1} \rightarrow x_{i,2}) \wedge (x_{i,2} \rightarrow x_{i,3}) \wedge \cdots \wedge (x_{i,k-1} \rightarrow x_{i,k}) \wedge (x_{i,k} \rightarrow x_{i,1})$ as $q_i = x_{i,1} \cdot x_{i,2} \cdot \ldots \cdot x_{i,k} + (1 - x_{i,1}) \cdot (1 - x_{i,2}) \cdot \ldots \cdot (1 - x_{i,k})$, and we arithmetize each empty conjunction D_i as $q_i = 1$. Then, $p' = p_1 \cdot p_2 \cdot \ldots \cdot p_m \cdot q_1 \cdot q_2 \cdot \ldots \cdot q_n$ is an arithmetization of F', and p' is of degree at most two in each variable. Furthermore, we have that the number of truth assignments that satisfy F is exactly $\mathcal{S}(p')$.

2.2.2 An interactive proof for #3-SAT

We now present the interactive proof for #3-SAT described in [BF91].

On input (F, α), where F is a propositional formula on variables $x_1, \ldots, x_n$, the prover P wishes to convince the verifier V that the number of truth assignments to $x_1, \ldots, x_n$ that satisfy F is α.

P starts by selecting a prime q such that $2^n < q < 2^{n+1}$, and sends q to V who verifies that q is indeed prime (and if not, V rejects). All arithmetic will now be performed over the field $\mathbb{F}_q$.

The prover then computes, as described in section 2.2.1, a polynomial $p(y_1, \ldots, y_m)$ that is of degree at most two in each variable such that $\mathcal{S}(p)$ is the number of truth assignments that satisfy F. Note that m will be polynomial in n. P now wishes to convince V that $\mathcal{S}(p) = \alpha$. The interaction then proceeds in m rounds.

Let $\alpha_0 = \alpha$. In round 1, P computes the degree-2 polynomial

$$p_1(y_1) = \sum_{(y_2,\ldots,y_m)\in\{0,1\}^{m-1}} p(y_1, \ldots, y_m)$$

and sends the coefficients of a polynomial p'_1, supposedly equal to p_1, to V. V checks that $p'_1(0) + p'_1(1) = \alpha_0$, and if not, V rejects. V then randomly selects $r_1 \in \mathbb{F}_q$, computes $\alpha_1 = p'_1(r_1)$, and sends r_1 to P. P's new goal is to convince V that

$$\sum_{(y_2,\ldots,y_m)\in\{0,1\}^{m-1}} p(r_1, y_2, \ldots, y_m) = \alpha_1.$$

Similarly, in round i, P computes the degree-2 polynomial

$$p_i(y_i) = \sum_{(y_{i+1},\ldots,y_m)\in\{0,1\}^{m-i}} p(r_1, \ldots, r_{i-1}, y_i, \ldots, y_m)$$

and sends the coefficients of a polynomial p'_i, supposedly equal to p_i, to V. V checks that $p'_i(0) + p'_i(1) = \alpha_{i-1}$, and if not, V rejects. V then randomly selects $r_i \in \mathbb{F}_q$, computes $\alpha_i = p'_i(r_i)$, and sends r_i to P. P's new goal is to convince V that

$$\sum_{(y_{i+1},\ldots,y_m)\in\{0,1\}^{m-i}} p(r_1, \ldots, r_i, y_{i+1}, \ldots, y_m) = \alpha_i.$$

After round m, V computes $p(r_1, \ldots, r_m)$ and checks if this equals α_m. If so, V accepts, and otherwise, V rejects.

It is easy to see that if α is in fact the number of truth assignments that satisfy F and P is an honest prover (one that correctly sends polynomial p_i at each round i) then (P, V) accepts (F, α).

Now suppose that α is not the number of truth assignments that satisfy F. Consider the probability that any cheating prover P' convinces V to accept (F, α). Recall that after each round, P' has a new statement that it is trying to prove. Since α is not the number of truth assignments that satisfy F, we have $\sum_{(y_1,\ldots,y_m)\in\{0,1\}^m} p(y_1, \ldots, y_m) \neq \alpha_0$. If V accepts (F, α), we have $p(r_1, \ldots, r_m) = \alpha_m$. In this case we have that before round 1, P' is trying to prove a false statement, but after round m, P' is trying to prove a true statement. Then there exists an i such that before round i, P' is trying to prove a false statement, but after round i, P' is trying to prove a true statement. That is, there exists an i such that

$$\sum_{(y_i,\ldots,y_m)\in\{0,1\}^{m-i+1}} p(r_1, \ldots, r_{i-1}, y_i, \ldots, y_m) \neq \alpha_{i-1} \tag{2.1}$$

but

$$\sum_{(y_{i+1},\ldots,y_m)\in\{0,1\}^{m-i}} p(r_1, \ldots, r_i, y_{i+1}, \ldots, y_m) = \alpha_i. \tag{2.2}$$

Fix i, suppose that (2.1) holds, and consider the probability that both (2.2) holds and V doesn't reject in round i. Since (2.1) holds, we have $p_i(0) + p_i(1) \neq \alpha_{i-1}$. For V to not reject in round i, we must have $p'_i(0) + p'_i(1) = \alpha_{i-1}$. Therefore, we must have $p_i \neq p'_i$. By the definition of V, we have $p'_i(r_i) = \alpha_i$. For (2.2) to hold, we must have $p_i(r_i) = \alpha_i$, and hence we must have $p_i(r_i) = p'_i(r_i)$. But note that p_i and p'_i are degree-2 polynomials and since $p_i \neq p'_i$, these polynomials can agree on at most two points. Since r_i is randomly chosen, the probability that $p_i(r_i) = p'_i(r_i)$ is $2/|\mathbb{F}_q| < 2/2^n$. That is, the probability that both (2.2) holds and V doesn't reject in round i is less than $2/2^n$. Then, since there are at most m rounds, the probability that P' convinces V to accept (F, α) is $< 2m/2^n$.

Note that V needs to evaluate p only once, and that it is enough for V to have oracle access to p. Also note that arithmetic is performed over the field $\mathbb{F}_q$ simply to ensure that the α_i and the coefficients of the p_i do not get too big; performing arithmetic over $\mathbb{F}_q$ ensures that each such number can be represented using $n + 1$ bits. Alternatively, arithmetic can be performed over the integers, and each r_i can be randomly selected so that $0 \leq r_i \leq 2^n$. In this case, the α_i and the coefficients of the p_i will still have representation length polynomial in n.

Recall that we would like to consider approaches for showing that #3-SAT has polynomial-size circuits. Since we would like to explore whether interaction can be converted into nonuniformity, we will consider approaches that are based on the ideas we have seen in the interactive proof for #3-SAT. In particular, we will consider approaches for showing that there are polynomial-size[3] circuits that, given oracle access to a polynomial p of degree at most two in each variable, compute $\mathcal{S}(p)$. If such polynomial-size circuits exist, then #3-SAT has polynomial-size circuits.

2.3 The linear combination approach

For an n-variable polynomial p of degree at most two in each variable and for a particular point $\vec{a}$, $p(\vec{a})$ is simply some fixed[4] linear combination of p's coefficients. Similarly, the $\mathcal{S}(p)$ is a fixed linear combination of p's coefficients. Now, suppose that the only information we are given about such a polynomial p is its value at each point in a set T. Each such value is the value of some linear combination of p's coefficients. Let V be the set of linear combinations of p's coefficients associated with the points in T. Note that the linear combination associated with $\mathcal{S}(p)$ is either independent of or dependent on the linear combinations in V. If it is independent, then we have no information about

[3]Here we mean polynomial in the number of variables, n.

[4]By "fixed", we mean that the coefficients of the linear combination will be the same for all n-variable polynomials p of degree at most two in each variable.

the value of $\mathcal{S}(p)$, and if it is dependent, then the value of $\mathcal{S}(p)$ is simply a fixed linear combination of the values we have been given.

Suppose that for each n, there exists a set T of points, with $|T|$ polynomial in n, such that for polynomials p on n variables and of degree at most two in each variable, the linear combination of p's coefficients associated with $\mathcal{S}(p)$ is dependent on the linear combinations of p's coefficients associated with the points in T. Then for each n there exists a circuit C_n of size polynomial in n that, given oracle access to any polynomial p on n variables and of degree at most two in each variable, computes $\mathcal{S}(p)$. However, Kabanets and Shpilka [KS03] show that any such set T has exponential size.

Theorem 2.3.1 (Kabanets, Shpilka). *Let $n \in \mathbb{N}$, and let F be a field such that $|F| > 2^n$. Assume there exist k points $\vec{a_1}, \ldots, \vec{a_k} \in F^n$ and k coefficients $\alpha_1, \ldots, \alpha_k \in F$, such that for every polynomial $p \in F[x_1, \ldots, x_n]$ of degree at most two in each variable, we have that*

$$\mathcal{S}(p) = \sum_{i=1}^{k} \alpha_i \cdot p(\vec{a_i}).$$

Then $k \geq 2^n$.

Proof. We begin by defining some notation that we will use.

For $\vec{v} \in F^m$, we will let $(\vec{v})_i$ denote the i-th coordinate of $\vec{v}$.

For $\vec{v}, \vec{w} \in F^m$, we will let $\vec{v} \cdot \vec{w}$ denote the component-wise multiplication of $\vec{v}$ and $\vec{w}$. That is, $\vec{x} = \vec{v} \cdot \vec{w}$ will be the point such that $(\vec{x})_i = (\vec{v})_i \cdot (\vec{w})_i$ for each i. Then, $\vec{v}^2$ will denote $\vec{v} \cdot \vec{v}$, and for $\vec{v_1}, \ldots, \vec{v_j} \in F^m$, $\prod_{i=1}^{j} \vec{v_i}$ will denote $\vec{v_1} \cdot \vec{v_2} \cdot \ldots \cdot \vec{v_j}$.

Finally, for $\vec{v}, \vec{w} \in F^m$, we will let $\langle \vec{v}, \vec{w} \rangle$ denote the inner product of $\vec{v}$ and $\vec{w}$. That is, $\langle \vec{v}, \vec{w} \rangle = \sum_{i=1}^{m} (\vec{v})_i \cdot (\vec{w})_i$.

Now, let $\vec{a_1}, \ldots, \vec{a_k} \in F^n$ and $\alpha_1, \ldots, \alpha_k \in F$, and suppose that for every polynomial $p \in F[x_1, \ldots, x_n]$ of degree at most two in each variable, we have that

$$\mathcal{S}(p) = \sum_{i=1}^{k} \alpha_i \cdot p(\vec{a_i}). \tag{2.3}$$

For $1 \leq i \leq n$, let $\vec{v_i} = (2(\vec{a_1})_i - 1, 2(\vec{a_2})_i - 1, \ldots, 2(\vec{a_k})_i - 1)$. Then, for $S \subseteq \{1, \ldots, n\}$, let

$$\vec{v_S} = \prod_{i \in S} \vec{v_i}.$$

We will show that the $\vec{v_S}$ are linearly independent, and hence $k \geq 2^n$.

Let $S, T \subseteq \{1, \ldots, n\}$, and consider the inner product $\langle \vec{\alpha}, \vec{v_S} \cdot \vec{v_T} \rangle$, where $\vec{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_k)$. Define polynomial $q(x_1, \ldots, x_n) = \prod_{i=1}^{n} (2x_i - 1)^{e_i}$, where for each i,

$$e_i = \begin{cases} 2 & \text{if } \; i \in S \cap T, \\ 0 & \text{if } \; i \notin S \cup T, \\ 1 & \text{otherwise.} \end{cases}$$

Now, we have

$$\langle \vec{\alpha}, \vec{v_S} \cdot \vec{v_T} \rangle = \sum_{i=1}^{k} \alpha_i \cdot q(\vec{a_i}).$$

Then, since q is of degree at most two in each variable, we have by (2.3) that

$$\langle \vec{\alpha}, \vec{v_S} \cdot \vec{v_T} \rangle = \mathcal{S}(q).$$

However, note that

$$\mathcal{S}(q) = \begin{cases} 0 & \text{if } e_i = 1 \text{ for some } i, \\ 2^n & \text{otherwise.} \end{cases}$$

Then, since $S \neq T$ iff there exists i such that $e_i = 1$, we have

$$\langle \vec{\alpha}, \vec{v_S} \cdot \vec{v_T} \rangle = \begin{cases} 0 & \text{if } \; S \neq T, \\ 2^n & \text{if } \; S = T. \end{cases}$$

Suppose that $\sum_{T \subseteq \{1, \ldots, n\}} \beta_T \cdot \vec{v_T} = \vec{0}$, where each $\beta_T \in F$. To show that the $\vec{v_T}$ are linearly independent, we need to show that each $\beta_T = 0$.

Let $S \subseteq \{1, \ldots, n\}$ and consider β_S. We have

$$\begin{aligned}
2^n \cdot \beta_S &= \sum_{T \subseteq \{1,\ldots,n\}} \beta_T \cdot \langle \vec{\alpha}, \vec{v_S} \cdot \vec{v_T} \rangle \\
&= \sum_{T \subseteq \{1,\ldots,n\}} \beta_T \cdot \sum_{i=1}^{k} \alpha_k \cdot (\vec{v_S})_k \cdot (\vec{v_T})_k \\
&= \sum_{T \subseteq \{1,\ldots,n\}} \sum_{i=1}^{k} \alpha_k \cdot (\vec{v_S})_k \cdot \beta_T \cdot (\vec{v_T})_k \\
&= \sum_{i=1}^{k} \alpha_k \cdot (\vec{v_S})_k \cdot \sum_{T \subseteq \{1,\ldots,n\}} \beta_T \cdot (\vec{v_T})_k \\
&= \sum_{i=1}^{k} \alpha_k \cdot (\vec{v_S})_k \cdot \left(\sum_{T \subseteq \{1,\ldots,n\}} \beta_T \cdot \vec{v_T} \right)_k \\
&= \left\langle \vec{\alpha}, \vec{v_S} \cdot \sum_{T \subseteq \{1,\ldots,n\}} \beta_T \cdot \vec{v_T} \right\rangle \\
&= \langle \vec{\alpha}, \vec{v_S} \cdot \vec{0} \rangle \\
&= 0
\end{aligned}$$

Therefore, $\beta_S = 0$. $\square$

Recall that for any n-variable polynomial p of degree at most two in each variable, $\mathcal{S}(p)$ is some fixed linear combination of p's coefficients. Similarly, for each point $\vec{x}$, $p(\vec{x})$ is some fixed linear combination of p's coefficients. It follows from Theorem 2.3.1 that for such a polynomial p, the linear combination of p's coefficients associated with $\mathcal{S}(p)$ is independent of the linear combinations of p's coefficients associated with evaluating p on each value in any set T of points with $|T| < 2^n$. That is, knowing the value of such a polynomial p at each point in such a set T does not in any way constrain the value of $\mathcal{S}(p)$ – for each $\alpha \in F$, there exists an n-variable polynomial p' of degree at most two in each variable such that p' takes on the same values as p on T and $\mathcal{S}(p') = \alpha$.

Corollary 2.3.2 *Let $n \in \mathbb{N}$, let F be a field such that $|F| > 2^n$, and let $p \in F[x_1, \ldots, x_n]$ be a polynomial of degree at most two in each variable. If the only information we have*

about p is its value at fewer than 2^n points, then we have no information whatsoever about the value of $\mathcal{S}(p)$. That is, if we are only given the value of p at fewer than 2^n points, we cannot rule out any $\alpha \in F$ as a possible value of $\mathcal{S}(p)$.

Note that if a circuit C is given oracle access to a polynomial $p \in F[x_1, \ldots, x_n]$, then the only information that C has about p is p's value at the points corresponding to C's oracle queries. This is the case even if C makes its queries adaptively. Combining this fact with Corollary 2.3.2, we get the following additional corollary.

Corollary 2.3.3 *Let $n \in \mathbb{N}$ and let F be a field such that $|F| > 2^n$. Suppose C_n is a circuit that, given oracle access to a polynomial $p \in F[x_1, \ldots, x_n]$ of degree at most two in each variable, makes at most k (possibly adaptive) oracle queries and outputs $\mathcal{S}(p)$. Then $k \geq 2^n$.*

2.4 The $2n$-variable polynomial approach

A polynomial-size circuit can make at most polynomially many oracle queries. Consequently, by Corollary 2.3.3, if we want polynomial-size circuits that, given oracle access to an n variable polynomial p of degree at most two in each variable, compute $\mathcal{S}(p)$, it is not possible to have only one circuit for each n. We need more nonuniformity.

In this section, we will consider allowing a polynomial amount of precomputation on each $2n$-variable polynomial p of degree at most two in each variable. We will view a $\{0,1\}$-valued assignment to the first n variables of such a polynomial as specifying (the arithmetization of) a propositional formula on n variables. So, given a $\{0,1\}$-valued assignment $\vec{x}$ to the first n variables, we would like to sum p over all $\{0,1\}$-valued assignments to the remaining n variables. Suppose there exists a circuit C_n that, given oracle access to any $2n$-variable polynomial $p(\vec{x}, \vec{y})$ of degree at most two in each variable, given polynomially many bits of precomputation on p, and given an input $\vec{x}$, computes $\mathcal{S}(p(\vec{x}, \vec{y}))$. We will show that such a circuit C_n must make exponentially many oracle

queries in the worst case.

Theorem 2.4.1 *Let $n \in \mathbb{N}$, let F be a field such that $|F| > 2^n$, and let $k \in \mathbb{N}$. Suppose C_n is a circuit that, given oracle access to a polynomial $p \in F[x_1, \ldots x_n, y_1, \ldots, y_n]$ of degree at most two in each variable, given $n \cdot 2^{n/3} - 1$ bits of precomputation on p, and given any input $\vec{x} \in \{0,1\}^n$, makes at most k (possibly adaptive) oracle queries and outputs $\sum_{\vec{y} \in \{0,1\}^n} p(\vec{x}, \vec{y})$. Then $k \geq 2^{n/3}$.*

Before proving the theorem, we prove the following lemma.

Lemma 2.4.2 *Let $n \in \mathbb{N}$, let F be a field such that $|F| > 2^n$, and let $k \in \mathbb{N}$. Suppose C_n is a circuit that, given oracle access to $2^{n/3}$ polynomials $p_0, p_1, \ldots, p_{2^{n/3}-1} \in F[x_1, \ldots, x_n]$ of degree at most two in each variable, and given $n \cdot 2^{n/3} - 1$ bits of precomputation on the tuple $(p_0, p_1, \ldots p_{2^{n/3}-1})$, queries each polynomial (possibly adaptively) at most k times and outputs $(a_0, a_1, \ldots a_{2^{n/3}-1})$, where $a_i = \mathcal{S}(p_i)$ for each i. Then $k \geq 2^n$.*

Proof. Let C_n be a circuit that, given oracle access to $2^{n/3}$ polynomials

$$p_0, p_1, \ldots, p_{2^{n/3}-1} \in F[x_1, \ldots, x_n]$$

of degree at most two in each variable, and given $n \cdot 2^{n/3} - 1$ bits of precomputation on the tuple $(p_0, p_1, \ldots p_{2^{n/3}-1})$, queries each polynomial at most $2^n - 1$ times and outputs a tuple $(a_0, a_1, \ldots, a_{2^{n/3}-1}) \in F^{2^{n/3}}$. We say that the circuit *succeeds* if $a_i = \mathcal{S}(p_i)$ for each i. We will show that the circuit does not always succeed.

Let $\mathcal{P}$ be the set of all n-variable polynomials over F of degree at most two in each variable.

Fix a string $s \in \{0,1\}^{n \cdot 2^{n/3}-1}$, and consider running C_n using s as the precomputed string.

Partition $\mathcal{P}^{2^{n/3}}$ into sets $\mathcal{P}_1^{2^{n/3}}, \ldots, \mathcal{P}_m^{2^{n/3}}$ so that the following conditions hold:

1. For each i, if $\vec{p}, \vec{q} \in \mathcal{P}_i^{2^{n/3}}$ then the oracle queries made and responses seen by C_n when it is run with oracle access to $\vec{p}$ are identical to the oracle queries made and responses seen by C_n when it is run with oracle access to $\vec{q}$.

2. For all $\vec{p}, \vec{q} \in \mathcal{P}^{2^{n/3}}$, if the oracle queries made and responses seen by C_n when it is run with oracle access to $\vec{p}$ are identical to the oracle queries made and responses seen by C_n when it is run with oracle access to $\vec{q}$, then there exists i such that $\vec{p}, \vec{q} \in \mathcal{P}_i^{2^{n/3}}$.

Now, fix $1 \leq i \leq m$ and consider running C_n on tuples in $\mathcal{P}_i^{2^{n/3}}$. Note that C_n will give the same output for each tuple in $\mathcal{P}_i^{2^{n/3}}$, since C_n will make the same queries and get the same responses. However, C_n makes fewer than 2^n queries to each polynomial to which it is given oracle access. Then, by Corollary 2.3.2, the value of each sum that C_n is trying to compute is completely unconstrained by the oracle queries made and responses seen by C_n. That is, for each integer $0 \leq j < 2^{n/3}$ and each $b \in F$, there exist one or more polynomials $p \in \mathcal{P}$ such that p is consistent with the responses that C_n receives from its j-th oracle and $\mathcal{S}(p) = b$. Furthermore, the number of such polynomials p is independent of the value of b. Then, for each tuple $(b_0, b_1, \ldots, b_{2^{n/3}-1}) \in F^{2^{n/3}}$, there exist one or more tuples $(p_0, p_1, \ldots, p_{2^{n/3}-1}) \in \mathcal{P}^{2^{n/3}}$ such that for each j, p_j is consistent with the responses that C_n receives from its j-th oracle and $\mathcal{S}(p_j) = b_j$. Again, the number of such tuples $(p_0, p_1, \ldots, p_{2^{n/3}-1})$ is independent of the value of $(b_0, b_1, \ldots, b_{2^{n/3}-1})$. But note that each such tuple will be in $\mathcal{P}_i$. This means that C_n will succeed on exactly

$$\frac{1}{|F|^{2^{n/3}}} < \frac{1}{2^{n \cdot 2^{n/3}}}$$

of the tuples in $\mathcal{P}_i^{2^{n/3}}$.

So, when C_n is run using any fixed precomputed string, it will succeed on fewer than $\frac{1}{2^{n \cdot 2^{n/3}}}$ of the tuples in $\mathcal{P}^{2^{n/3}}$. But there are only $2^{n \cdot 2^{n/3}-1}$ precomputed strings. This means that for more than

$$\left(1 - \frac{2^{n \cdot 2^{n/3}-1}}{2^{n \cdot 2^{n/3}}}\right) = \frac{1}{2}$$

of the tuples in $\mathcal{P}^{2^{n/3}}$, C_n will not succeed no matter which precomputed string it uses. □

We are now ready to prove the theorem.

Proof. (Theorem 2.4.1)

Let C_n be a circuit that, given oracle access to a polynomial $p \in F[x_1, \ldots x_n, y_1, \ldots, y_n]$ of degree at most two in each variable, given $n \cdot 2^{n/3} - 1$ bits of precomputation on p, and given any input $\vec{x} \in \{0,1\}^n$, makes at most k oracle queries and outputs $\sum_{\vec{y} \in \{0,1\}^n} p(\vec{x}, \vec{y})$. Suppose for the sake of contradiction that $k \leq 2^{n/3} - 1$

We will construct a circuit D_n that contradicts Lemma 2.4.2. D_n will have oracle access to $2^{n/3}$ polynomials $p_0, p_1, \ldots, p_{2^{n/3}-1} \in F[x_1, \ldots x_n]$ of degree at most two in each variable, along with $n \cdot 2^{n/3} - 1$ bits of precomputation on $(p_0, p_1, \ldots p_{2^{n/3}-1})$. We will refer to the precomputed string as s.

For $i \in \mathbb{N}$ such that $i < 2^n$, let $\bar{i} = \bar{i}_1 \ldots \bar{i}_n$ denote the n-bit binary representation of i.

Let $q \in F[x_1, \ldots x_n, y_1, \ldots, y_n]$ be defined as follows:

$$q(x_1, \ldots, x_n, y_1, \ldots, y_n) = \sum_{i=0}^{2^{n/3}-1} r_{i,1}(x_1) \cdot r_{i,2}(x_2) \cdots r_{i,n}(x_n) \cdot p_i(y_1, \ldots, y_n)$$

where for each i and k,

$$r_{i,k}(z) = \begin{cases} z & \text{if } \bar{i}_k = 1, \\ 1 - z & \text{if } \bar{i}_k = 0. \end{cases}$$

Note that for $0 \leq i < 2^{n/3}$ and for each $\vec{y} \in F^n$, we have $q(\bar{i}_1, \bar{i}_2, \ldots, \bar{i}_n, \vec{y}) = p_i(\vec{y})$. Also, note that q is of degree at most two in each variable.

For any $\vec{x}, \vec{y} \in F^n$, D_n can compute $q(\vec{x}, \vec{y})$ by making at most $2^{n/3}$ oracle queries.

Now, D_n will simulate C_n running with oracle access to q and with precomputed string s. For $0 \leq i < 2^{n/3}$, D_n will let a_i be the output of C_n on input $(\bar{i}_1, \bar{i}_2, \ldots, \bar{i}_n)$. So D_n will run C_n $2^{n/3}$ times. On each such run, C_n will make at most $2^{n/3} - 1$ queries to q.

To answer each such query, D_n will make at most $2^{n/3}$ queries to its oracles. Then, the total number of queries made by D_n will be at most

$$(2^{n/3}) \cdot (2^{n/3} - 1) \cdot (2^{n/3}) = 2^n - 2^{n/3} < 2^n.$$

This means that D_n will not query any of its oracles more than $2^n - 1$ times.

D_n will output $(a_0, a_1, \ldots, a_{2^{n/3}-1})$. If the precomputed string s used by D_n is the same as the precomputed string that C_n would use on q, then for each i we will have $a_i = \mathcal{S}(p_i)$. □

Note that Theorem 2.4.1 can easily be generalized to $(m+n)$-variable polynomials of degree at most two in each variable, where $m \geq n$ and where we view a $\{0,1\}$-valued assignment to the first m variables of such a polynomial as specifying (the arithmetization of) a propositional formula on n variables.

2.5 The small low-degree descriptor approach

So far, the only restriction we have placed on the polynomials that we have considered is that they are of degree at most two in each variable. However, not every such polynomial will be the arithmetization of some propositional formula. In fact, for each n, there are double-exponentially many n-variable polynomials of degree at most two in each variable, but only exponentially many 3-CNF formulas in n variables. So, despite the results we have seen so far, it may still be the case that there are polynomial-size circuits for #3-SAT. We need to place additional restrictions on the polynomials we consider.

In this section, we will consider polynomials whose coefficients have a "simple" description. Specifically, we will consider polynomials that have a *small low-degree descriptor*. For each n, there are only exponentially many n-variable polynomials of degree at most two in each variable that have such a descriptor. We will show that there exist polynomial-size circuits for computing $\mathcal{S}(p)$ for polynomials p that have small low-degree

descriptors. However, we will also show that there exist propositional formulas whose arithmetizations do not have small low-degree descriptors.

For an n-variable polynomial p, a small low-degree descriptor q is a low-degree $\log(n)$-variable polynomial that we can use to efficiently compute any coefficient of p.

Definition 2.5.1 (Small degree-k descriptor). Let $n \in \mathbb{N}$, let F be a field, and let $p \in F[x_1, \ldots, x_n]$ be a polynomial of degree at most two in each variable, say

$$p(\vec{x}) = \sum_{\vec{i} \in \{0,1,2\}^n} c_{i_1 i_2 \ldots i_n} x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n}.$$

Suppose there exists a polynomial $q \in F[x_1, \ldots, x_{\log n}]$ of degree at most k in each variable such that for all $\vec{i} \in \{0,1,2\}^n$,

$$c_{i_1 i_2 \ldots i_n} = q((i_1 i_2 \ldots i_{\frac{n}{\log n}}),\ (i_{\frac{n}{\log n}+1} i_{\frac{n}{\log n}+2} \ldots i_{\frac{2n}{\log n}}),\ \ldots,\ (i_{\frac{(\log n - 1)n}{\log n}+1} i_{\frac{(\log n - 1)n}{\log n}+2} \ldots i_n))$$

where q's arguments are viewed as ternary numbers. Then we say that q is p's *small degree-k descriptor.*

If we know that an n-variable polynomial p of degree at most two in each variable has a small degree-k descriptor q, then we can compute $\mathcal{S}(p)$ by evaluating p at each point in a set $S \subseteq \{0,1\}^n$ of size polynomial in n and then taking a linear combination of these values (recall that by Theorem 2.3.1, this can't be done for general p). Note that it is enough to know that q exists - we don't need to actually use q.

Theorem 2.5.1 *Let F be a field and let $n, k \in \mathbb{N}$. There exist $m \leq n^{\log(k+1)}$ points $\vec{w_1}, \ldots, \vec{w_m} \in \{0,1\}^n$ and m coefficients $\alpha_1, \ldots, \alpha_m \in F$, such that for every polynomial $p \in F[x_1, \ldots, x_n]$ that is of degree at most two in each variable and has a small degree-k descriptor q, we have that*

$$\mathcal{S}(p) = \sum_{i=1}^{m} \alpha_i \cdot p(\vec{w_i}).$$

Proof. Say the coefficients of $p \in F[x_1, \ldots, x_n]$ are $a_1, a_2, \ldots, a_{3^n}$, and say the coefficients of p's degree-k descriptor q are $b_1, b_2, \ldots, b_{n^{\log(k+1)}}$.

Now, for each $\vec{x} \in F^n$, $p(\vec{x})$ is a linear combination of the a_i. Since q is p's small degree-k descriptor, we have that for each a_i there exists $\vec{y} \in F^{\log n}$ such that $a_i = q(\vec{y})$. But for each $\vec{y} \in F^{\log n}$, $q(\vec{y})$ is a linear combination of the b_i. This means that for each $\vec{x} \in F^n$, $p(\vec{x})$ is a linear combination of the b_i. That is, for each $\vec{x} \in F^n$, there exist $c_1, \ldots, c_{n^{\log(k+1)}} \in F$ such that $p(\vec{x}) = \sum_{i=1}^{n^{\log(k+1)}} c_i \cdot b_i$.

For convenience, if $p(\vec{x}) = \sum_{i=1}^{n^{\log(k+1)}} c_i \cdot b_i$, we will say that $\vec{x}$ and $(c_1, \ldots, c_{n^{\log(k+1)}})$ *correspond* to each other.

Consider the set

$$\mathcal{V} = \left\{ (c_1, \ldots, c_{n^{\log(k+1)}}) \mid \vec{x} \in \{0,1\}^n \text{ and } p(\vec{x}) = \sum_{i=1}^{n^{\log(k+1)}} c_i \cdot b_i \right\}.$$

Let $\mathcal{V}' \subseteq \mathcal{V}$ be a set of linearly independent vectors that span $\mathcal{V}$. Note that $|\mathcal{V}'| \leq n^{\log(k+1)}$. Let $\mathcal{T} \subseteq \{0,1\}^n$ be a set of minimal size such that for every vector $\vec{v} \in \mathcal{V}'$, there exists $\vec{x} \in \mathcal{T}$ such that $\vec{x}$ corresponds to $\vec{v}$. Note that $|\mathcal{T}| \leq |\mathcal{V}'| \leq n^{\log(k+1)}$.

Let $\vec{z} \in \{0,1\}^n$, and say that $p(\vec{z}) = \sum_{i=1}^{n^{\log(k+1)}} c_i \cdot b_i$. The vector $(c_1, \ldots, c_{n^{\log(k+1)}}) \in \mathcal{V}$ is a linear combination of vectors in $\mathcal{V}'$. But then $p(\vec{z})$ is a linear combination of values in the set $\{p(\vec{x}) \mid \vec{x} \in \mathcal{T}\}$.

So, once we compute $p(\vec{x})$ for all $\vec{x} \in \mathcal{T}$, we can take a linear combinations of these values to find the value of $p(\vec{z})$ at any $\vec{z} \in \{0,1\}^n$. But this means that $\mathcal{S}(p)$ is simply a linear combination of values in the set $\{p(\vec{x}) \mid \vec{x} \in \mathcal{T}\}$. $\square$

Theorem 2.5.1 implies that for each $k \in \mathbb{N}$, there exists a family of polynomial-size circuits $\{C_n\}$ such that for each n, circuit C_n, given oracle access to any n-variable polynomial p that is of degree at most two in each variable and has a small degree-k descriptor, computes $\mathcal{S}(p)$. However, it is not clear if this family is uniform. The theorem simply tells us that for each n, there *exists* a polynomial-size set of points T such that we can compute $\mathcal{S}(p)$ by evaluating p at each point in T and then taking a linear combination of these values. What is not clear is whether given n, we can efficiently compute T and the coefficients of the linear combination.

Unfortunately, Theorem 2.5.1 also implies that by restricting our attention to polynomials that have small low-degree descriptors, we have in fact been *too* restrictive - it follows from Theorem 2.5.1 that there exist propositional formulas whose arithmetizations do not have small low-degree descriptors.

Corollary 2.5.2 *Let F be a field and let $k \in \mathbb{N}$. There exists a propositional formula whose arithmetization does not have a small degree-k descriptor.*

Proof. Let $n \in \mathbb{N}$ be sufficiently large so that $2^n > n^{\log(k+1)}$. As given by Theorem 2.5.1, let $\vec{w_1}, \ldots, \vec{w_m} \in \{0,1\}^n$ and $\alpha_1, \ldots, \alpha_m \in F$, with $m \leq n^{\log(k+1)}$, be such that for every polynomial $p \in F[x_1, \ldots, x_n]$ that is of degree at most two in each variable and has a small degree-k descriptor, we have $\mathcal{S}(p) = \sum_{i=1}^{m} \alpha_i \cdot p(\vec{w_i})$.

Let $\vec{y} = (y_1, \ldots, y_n) \in \{0,1\}^n$ be such that $\vec{y}$ is different from every $\vec{w_i}$.

Define propositional formula $A = \ell_1 \wedge \ell_2 \wedge \cdots \wedge \ell_n$, where for each i, $\ell_i = x_i$ if $y_i = 1$ and $\ell_i = \neg x_i$ otherwise. Note that A has exactly one satisfying truth assignment. Following the procedure described in section 2.2.1, the arithmetization of A is

$$p_A(x_1, \ldots, x_n) = \left(\prod_{y_i=1} x_i\right) \cdot \left(\prod_{y_i=0} (1 - x_i)\right).$$

Note that $p_A(\vec{y}) = 1$, and for all $\vec{z} \in \{0,1\}^n$ such that $\vec{z} \neq \vec{y}$, we have $p_A(\vec{z}) = 0$. But then

$$\sum_{i=1}^{m} \alpha_i \cdot p_A(\vec{w_i}) = \sum_{i=1}^{m} \alpha_i \cdot 0 = 0,$$

even though

$$\mathcal{S}(p_A) = 1.$$

This means that p_A does not have a small degree-k descriptor. □

2.6 Computing the coefficients of a polynomial

The coefficients of a polynomial have played an important role in this chapter. We have seen that for a polynomial p, $\mathcal{S}(p)$ is some linear combination of p's coefficients. We have

also seen that there exist polynomial-size circuits for computing $\mathcal{S}(p)$ for polynomials p whose coefficients have a "simple" description. In this section, we show that, in general, computing even a single coefficient of a polynomial is hard. More specifically, we show that given an arithmetic expression for a low-degree polynomial, the problem of computing the coefficient of a given term is $\mathsf{P}^{\#\mathsf{P}}$-complete. We begin by formally defining the problem.

Definition 2.6.1 (Degree-k arithmetic expression)**.** Let $E(x_1, \ldots, x_n)$ be an expression on integers, variables $x_1, \ldots, x_n$, and operations $\{+, -, \times\}$, such that E is of degree at most k in each variable. Then we say that E is a *degree-k arithmetic expression.*

Definition 2.6.2 (k-COEFFICIENT)**.** Given an expression $E(x_1, \ldots, x_n)$ such that E is a degree-k arithmetic expression, and given a term $x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}$ such that each $i_j \in \{0, \ldots, k\}$, the function problem k-COEFFICIENT is to compute the coefficient of the term $x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n}$ in E.

Note that while k-COEFFICIENT is a function problem, there is a natural decision problem associated with it. An instance of the decision problem is of the form $\langle E, x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}, s, m, b \rangle$, and the acceptance condition is that:

- $E(x_1, \ldots, x_n)$ is a degree-k arithmetic expression;
- $i_1, \ldots, i_n \in \{0, \ldots, k\}$, $s \in \{+, -\}$, $m \in \mathbb{N}$, $b \in \{0, 1\}$; and
- the coefficient c of the term $x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}$ in E has sign s, $|c| \geq 2^m$, and the m-th bit of $|c|$ (counting from the right starting at 0) is b.

It is clear that we can solve the function problem in polynomial time given an oracle for the decision problem, and vice-versa. Therefore, for simplicity, we will only refer to the function problem, even when dealing with classes of decision problems.

Theorem 2.6.1 *The problem* 1-COEFFICIENT *is in* P*, and for* $k \geq 2$*, the problem* k-COEFFICIENT *is* $\mathsf{P}^{\#\mathsf{P}}$*-complete.*

We will first consider the problem 1-COEFFICIENT.

Lemma 2.6.2 *The problem* 1-COEFFICIENT *is in* P*.*

Proof. Let $E(x_1, \ldots, x_n)$ be a degree-1 arithmetic expression, and let $i_1, \ldots, i_n \in \{0, 1\}$.

We will describe a polynomial-time algorithm to compute the coefficient of the term $x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}$ in E.

Without loss of generality, we may assume that each $i_j = 1$. To see this, suppose $i_j = 0$ for some j. Then we simply let E' be E with every occurrence of x_j replaced with 0, and we find the coefficient of the term $x_1 x_2 \cdots x_n$ in E'.

Let $b_0, \ldots, b_n$ be such that b_i is the sum of all coefficients of terms of total degree i in E. That is,

$$b_i = \sum_{\substack{c \text{ is the coefficient} \\ \text{of a term of total} \\ \text{degree i in } E(x_1,\ldots,x_n)}} c$$

Then, the coefficient of $x_1 x_2 \cdots x_n$ is b_n. Note that for any $0 \le k \le n$, we have

$$E(k, k, \ldots, k) = \sum_{i=0}^{n} k^i \cdot b_i. \tag{2.4}$$

Consider the system of $(n+1)$ equations on $(n+1)$ variables $b_0, \ldots, b_n$ defined by substituting $k = 0, 1, \ldots, n$ in equation (2.4). This is a Vandermonde system, and hence it has a unique solution. This solution can be found in time polynomial in $|E|$. That is, we can find b_n in time polynomial in $|E|$. □

We now consider the problem k-COEFFICIENT for $k \ge 2$.

Lemma 2.6.3 *Let $k \in \mathbb{N}$. The problem k-*COEFFICIENT *is in* $\mathsf{P}^{\#\mathsf{P}}$*.*

Proof. Consider the unsimplified expansion (i.e. before like terms are collected) of an arithmetic expression $E(x_1, \ldots, x_n)$. For any term $x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}$, let $pos(E, x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n})$ be the sum of all positive coefficients of $x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}$ in the unsimplified expansion of E, and let $neg(E, x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n})$ be the absolute value of the sum of all negative coefficients

of $x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}$ in the unsimplified expansion of E. Then the coefficient of $x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}$ in E is simply $pos(E, x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}) - neg(E, x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n})$.

To show that k-COEFFICIENT is in $\mathsf{P}^{\#\mathsf{P}}$, we will show that the functions neg and pos are in #P.

Consider the function neg. We will define a polynomial-time nondeterministic Turing machine M that on input $\langle E, x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n} \rangle$ has exactly $neg(E, x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n})$ accepting computation paths.

On input $\langle E, x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n} \rangle$, M works as follows. For each $+$ and $-$ in E, M nondeterministically selects one of $\{left, right\}$; if it selects $left$ then it deletes the operator and its left operand, and if it selects $right$ it deletes the operator and its right operand. What remains is an expression on integers, variables, and multiplication. M carries the out the multiplication; this can be done in polynomial time. The result is a single term. If this is a term of the form $c \cdot x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}$ with $c < 0$, then M branches into $|c|$ computation paths and accepts on each such path. Otherwise, M rejects. Then, the total number of accepting computation paths is exactly $neg(E, x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n})$, and hence $neg \in \#\mathsf{P}$.

It is easy to see that we can similarly define a polynomial-time nondeterministic Turing machine that on input $\langle E, x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n} \rangle$ has exactly $pos(E, x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n})$ accepting computation paths. Therefore, $pos \in \#\mathsf{P}$. □

Lemma 2.6.4 *Let $k \geq 2$. The problem k-COEFFICIENT is #P-hard under polynomial-time many-one reductions.*

Proof. Let $G = ((L, R), E)$ be a directed bipartite graph. We say that G has a *perfect matching* $M \subseteq E$ if the following hold:

1. for every $\ell \in L$, there exists exactly one $v \in R$ such that $(\ell, v) \in M$; and
2. for every $r \in R$, there exists exactly one $v \in L$ such that $(v, r) \in M$.

Then, given such a graph G, the problem #PERFECT_MATCHING is to count the number of perfect matchings that G has. #PERFECT_MATCHING is #P-complete [Val79].

We will reduce the problem #PERFECT_MATCHING to 2-COEFFICIENT (and hence to k-COEFFICIENT for $k \geq 2$).

Let $G = ((L, R), E)$ be a directed bipartite graph, and say the vertices of G are $x_1, \ldots, x_n$. Define arithmetic expression

$$F(x_1, \ldots, x_n) = \prod_{(x_i, x_j) \in E} (1 + x_i x_j).$$

Note that each term in the unsimplified expansion of F is formed by choosing, for each edge (x_i, x_j), either 1 or $x_i x_j$ and then taking the product of everything that has been chosen. In other words, each term in the unsimplified expansion of F is formed by first choosing a subset of E. Now, if the subset of E that is chosen is a perfect matching, then the term $x_1 x_2 \cdots x_n$ will be formed. This means that the number of perfect matchings in G will be the number of times that $x_1 x_2 \cdots x_n$ appears in the unsimplified expansion of F. But then the number of perfect matchings in G will simply be the coefficient of $x_1 x_2 \cdots x_n$ in F.

We are not done yet, since F is not necessarily of degree at most two in each variable. Note that the degree of each variable x_i in F will be equal to the number of times this variable appears in F.

We now show how to construct a degree-2 polynomial expression F' such that the coefficient of the term of degree one in each variable will be the number of perfect matchings in G.

For each variable x_i, say that x_i appears m_i times in F, create m_i new variables $x_{i,1}, \ldots, x_{i,m_i}$, and replace the j-th occurrence of x_i in F with $x_{i,j}$. This corresponds to replacing each vertex x_i in G with m_i copies, each of degree one. Then, define

$$C_i = \left(\sum_{j=1}^{m_i} \prod_{\ell \neq j} x_{i,\ell} \right).$$

Finally, let

$$F' = C_1 \cdot C_2 \cdots C_n \cdot F.$$

Note that F is a degree-2 polynomial expression. We claim that the number of perfect matchings in (the original) G is equal to the coefficient of the term

$$t = \prod_{i=1}^{n} \prod_{j=1}^{m_i} x_{i,j}$$

in F'. To see this, note that the purpose of the C_i is to force us to choose which copy of x_i we will use when selecting a subset of E. For example, if $m_i = 3$, then $C_i = (x_{i,1}x_{i,2} + x_{i,1}x_{i,3} + x_{i,2}x_{i,3})$. Then, choosing say $x_{i,1}x_{i,2}$ from C_i means that we will need to use the third copy of x_i (that is, $x_{i,3}$) when selecting a subset of E in order for the constructed term to be t. In other words, when constructing a term in the unsimplified expansion of F', we first choose which copy of each variable we will use, and then we select a subset of E. If the subset we select is a perfect matching in the original graph G and uses only the chosen copy of each variable, then the term constructed will be t. Therefore, the number of perfect matchings in G will be exactly the number of times that the term t appears in the unsimplified expansion of F'. □

Theorem 2.6.1 follows immediately from Lemmas 2.6.2, 2.6.3, and 2.6.4.

Chapter 3

Are there instance checkers for approximate counting?

In Chapter 2, we considered the problem of exact counting. In this chapter, we consider the problem of approximate counting. We discuss the difficulties that are encountered when trying to construct an instance checker for a problem that is complete for approximate counting.

3.1 Approximate counting

In an approximate counting problem, the goal is to compute a function $f \in \#\mathsf{P}$ to within a specified relative error. Under the plausible assumption that the polynomial-time hierarchy doesn't collapse, approximate counting is easier than exact counting.

Definition 3.1.1 (r-Approximation). Let $b, c \in \mathbb{R}$ and let $r \geq 1$. We say that b *r-approximates* c if $c \geq 0$ and $c/r \leq b \leq r \cdot c$ or if $c < 0$ and $r \cdot c \leq b \leq c/r$.

Note that for any $r \geq 1$, the r-approximation of a negative number must be negative, the r-approximation of 0 must be 0, and the r-approximation of a positive number must be positive.

Definition 3.1.2 (r-Approximate counting). Let $f \in \#\mathsf{P}$ and let $r : \mathbb{N} \to \mathbb{R}$ be such that $r(n) \geq 1$ for all n. The *r-approximate counting* problem associated with f is to compute f with relative error at most r. We say that a Turing machine M *r-approximates* f if on each input x, $M(x)$ $r(|x|)$-approximates $f(x)$.

Approximating #3-SAT and approximating the number of accepting inputs of a Boolean circuit are complete problems for approximate counting.

It is easy to see that for any $r \geq 1$, r-approximate counting is NP-hard. If a Turing machine M r-approximates #SAT, then for all inputs x we have that $M(x) > 0$ iff $x \in \mathsf{SAT}$.

On the other hand, the results of Stockmeyer [Sto83] and Toda [Tod91] suggest that approximate counting is easier than exact counting. Before giving these results, we need to define the *polynomial-time hierarchy*.

Definition 3.1.3 (Polynomial-time hierarchy [Sto76]). Define $\Delta_0^P = \mathsf{P}$ and $\Sigma_0^P = \mathsf{P}$. Then, for $i \geq 1$, define $\Delta_i^P = \mathsf{P}^{\Sigma_{i-1}^P}$ and $\Sigma_i^P = \mathsf{NP}^{\Sigma_{i-1}^P}$. The *polynomial-time hierarchy*, denoted PH, is the union $\bigcup_{i \geq 0} \Sigma_i^P$. It can be shown that for each i, we have $\Sigma_i^P \subseteq \Delta_{i+1}^P \subseteq \Sigma_{i+1}^P$, though it is not know if each such containment is proper. If $\Sigma_i^P = \Sigma_{i+1}^P$ for some i, then $PH = \Sigma_i^P$ and we say that the polynomial-time hierarchy *collapses to level i*.

Stockmeyer [Sto83] shows that $(1+\frac{1}{n^{O(1)}})$-approximate counting can be done in polynomial time given access to an oracle for Σ_2^P. That is, $(1+\frac{1}{n^{O(1)}})$-approximate counting is in Δ_3^P. Toda [Tod91] shows that $\mathsf{PH} \subseteq \mathsf{P}^{\#\mathsf{P}}$. Then, if the polynomial-time hierarchy doesn't collapse, we have $\mathsf{P} \subsetneq \mathsf{NP}$ and $\Delta_3^P \subsetneq \mathsf{P}^{\#\mathsf{P}}$ and hence $(1+\frac{1}{n^{O(1)}})$-approximate counting is easier than exact counting but harder than P.

3.2 Instance checkers

Determining if the output of a program is correct is an important problem. Approaches to solving this problem include formally proving the correctness of the program (but this can be difficult in practice) or testing the program on some finite set T of inputs (but this does not guarantee the program's correctness on inputs not in T). A new approach, *instance checking*, was introduced by Blum and Kannan [BK95]. With instance checking, a program's correctness is verified in real-time rather than beforehand. The interaction between an instance checker and a program can be viewed as a restricted form of the interaction between a verifier and a prover in an interactive proof. Recently, the existence of instance checkers has been used to obtain results about probabilistic computation [Bar02, TV02, GSTS03, FS04]. We will see an example of this in Chapter 4. While instance checkers exist for all $\mathsf{P}^{\#\mathsf{P}}$-complete languages and all languages in P, it is not known if instance checkers exist for languages complete for any NP-hard natural class strictly contained $\mathsf{P}^{\#\mathsf{P}}$; in fact, it is not even known if such a class exists.[1] However, as discussed in section 3.1, if the polynomial-time hierarchy doesn't collapse then the class of $(1 + \frac{1}{n^{O(1)}})$-approximate counting problems is an NP-hard class strictly contained in $\mathsf{P}^{\#\mathsf{P}}$.

An instance checker I for a language L can be used to efficiently verify the output of any Turing machine M that purportedly decides L. On an input x, the instance checker I makes at most polynomially many queries to M before either announcing whether or not $x \in L$ or announcing that M does not decide L. More formally, given an input x and given oracle access to M, I outputs $L(x)$ if M correctly decides L, and otherwise I almost always outputs either "?" or $L(x)$.

Definition 3.2.1 (Instance checker). Let L be a language. We say that a probabilis-

[1] The NP language GRAPH-ISOMORPHISM and its complement GRAPH-NONISOMORPHISM do have instance checkers. However, neither language is known to be NP-hard.

tic polynomial-time oracle Turing machine I is an *instance checker* for L if the following properties hold:

1. I always outputs either 0, 1, or "?".

2. For all inputs x, $I^L(x)$ outputs $L(x)$ with probability 1.

3. For all oracles M and for all inputs x, say $|x| = n$, the probability that $I^M(x)$ outputs $(1 - L(x))$ is at most $1/n^c$ for all c and sufficiently large n.

Every language $L \in \mathsf{P}$ has a trivial instance checker. On input x, the instance checker I does not make any oracle queries; it simply simulates the polynomial-time machine for L on x, and outputs 1 if the simulation accepts and 0 otherwise.

Every $\mathsf{P}^{\#\mathsf{P}}$-complete language, every PSPACE-complete language, and every EXP-complete language has an instance checker. This follows from the fact that the interactive proofs for $\mathsf{P}^{\#\mathsf{P}}$ [LFKN92, BF91] and PSPACE [Sha92] and the two-prover interactive proof for EXP [BFL91] can all be converted to *function-restricted* interactive proofs. Defined by Blum and Kannan [BK95], a function-restricted interactive proof for a language L is an interactive proof in which the prover is restricted to be an oracle and the honest prover is simply an oracle for L. The restriction on the prover means that its answers to all potential queries are fixed ahead of time, and hence the answer to any query is completely independent of any queries that have been made previously.

Definition 3.2.2 (Function-restricted interactive proof). Let L be a language. We say that L has a *function-restricted interactive proof* if there exists a probabilistic polynomial-time oracle Turing machine V such that the following two properties hold:

1. (Completeness) For all $x \in L$, V^L accepts x with probability 1.

2. (Soundness) For all oracles P and for all $x \notin L$, say $|x| = n$, the probability that V^P accepts x is at most $1/n^c$ for all c and sufficiently large n.

Theorem 3.2.1 (Blum, Kannan [BK95]). *Let L be a language. L has an instance checker if and only if both L and $\bar{L}$ have function-restricted interactive proofs.*

Proof.

$\Longrightarrow$:

Suppose L has an instance checker I. We construct a function-restricted interactive proof for L as follows. On input x and given access to oracle P, verifier V simulates I^P on input x, and accepts if the simulation outputs 1 and rejects otherwise. By the definition of I, V satisfies the completeness and soundness properties. Similarly, we construct a function-restricted interactive proof for $\bar{L}$ as follows. On input x and given access to oracle P, verifier V' simulates $I^{\bar{P}}$ on input x, and accepts if the simulation outputs 0 and rejects otherwise. Again, by the definition of I, V' satisfies the completeness and soundness properties.

$\Longleftarrow$:

Suppose both L and $\bar{L}$ have function-restricted interactive proofs. Say V_L is the verifier for L and $V_{\bar{L}}$ is the verifier for $\bar{L}$. We construct an instance checker for L as follows. On input x and given oracle access to M, instance checker I first queries M on x. If M accepts x, then I simulates ${V_L}^M$ on input x, and outputs 1 if the simulation accepts and outputs "?" otherwise. If M rejects x, then I simulates ${V_{\bar{L}}}^{\bar{M}}$ on input x, and outputs 0 if the simulation accepts and outputs "?" otherwise. By the definitions of V_L and $V_{\bar{L}}$, I satisfies the properties of an instance checker for L. $\Box$

Consider $L_{\#3\text{-}\mathsf{SAT}}$, the decision version of #3-SAT that is complete for $\mathsf{P}^{\#\mathsf{P}}$ under polynomial-time Turing reductions. $L_{\#3\text{-}\mathsf{SAT}}$ is simply the set of strings $\langle F, i\rangle$ such that F is a propositional formula in 3-CNF that has at least i satisfying truth assignments. Note that the interactive proof for #3-SAT given in section 2.2.2 can be converted into interactive proofs for $L_{\#3\text{-}\mathsf{SAT}}$ and $\overline{L_{\#3\text{-}\mathsf{SAT}}}$. On input $\langle F, i\rangle$, the prover P tells the verifier V how many satisfying assignments F has. If this is at least i in the case of $L_{\#3\text{-}\mathsf{SAT}}$ or

less than i in the case of $\overline{L_{\#3-\mathsf{SAT}}}$, then P and V proceed as in the interactive proof for #3-SAT. Otherwise, V rejects immediately. Now, note that each message that P sends to V can also be sent as a sequence of responses to $\mathsf{P}^{\#\mathsf{P}}$ queries (and hence as responses to $L_{\#3-\mathsf{SAT}}$ queries). In particular, to receive the coefficients of degree-2 polynomial p_j in round j of the interactive proof for #3-SAT, V can make a sequence of $\mathsf{P}^{\#\mathsf{P}}$ queries to determine the value of p_j at three points and V can then use interpolation to compute the coefficients. In this manner, the interactive proofs for $L_{\#3-\mathsf{SAT}}$ and $\overline{L_{\#3-\mathsf{SAT}}}$ can be converted into function-restricted interactive proofs and then, using Theorem 3.2.1, can be converted into an instance checker for $L_{\#3-\mathsf{SAT}}$.

In the remainder of this chapter, we will discuss the problems that arise when we try to convert the instance checker for $L_{\#3-\mathsf{SAT}}$ into an instance checker for the approximation version of #3-SAT. Before proceeding, we need to define what we mean by an instance checker for an approximate counting problem.

Definition 3.2.3 (Instance checker for an approximate counting problem). Let $f \in \#\mathsf{P}$ and let $r : \mathbb{N} \to \mathbb{R}$ be such that $r(n) \geq 1$ for all n. We say that a probabilistic polynomial-time oracle Turing machine I is an instance checker for the r-approximate counting problem associated with f if the following properties hold:

1. I always outputs either a number or "?".

2. For all oracles M that r-approximate f, I^M r-approximates f.

3. For all oracles M and for all inputs x, say $|x| = n$, the probability that $I^M(x)$ outputs neither "?" nor an r-approximation of $f(x)$ is at most $1/n^c$ for all c and sufficiently large n.

3.3 Approximate polynomial evaluation

Fix a function $k(n) = (1 + \frac{1}{n^{O(1)}})$, and consider trying to convert the instance checker for $L_{\#3-\mathsf{SAT}}$ into an instance checker for the k-approximation version of #3-SAT, which we will refer to as k-approx-#3-SAT. Recall from section 2.2.2 that we can either perform arithmetic over a finite field or over the integers. Since it is not clear what it means to "k-approximate" finite field elements, it makes sense to perform arithmetic over the integers. Recall that the instance checker for $L_{\#3-\mathsf{SAT}}$ works in several rounds, where in each round i it uses its oracle to evaluate polynomial p_i at three points, and then, using interpolation, it evaluates p_i at a randomly chosen point r_i. We will show that it is unlikely that there is an instance checker for k-approx-#3-SAT that works in a similar way. In particular, we show in section 3.3.1 that, given access to an oracle for k-approximate counting (that is, an oracle for k-approx-#3-SAT), it is unlikely that we can efficiently k-approximate p_i at a set of integers. Furthermore, we show in section 3.3.2 that even if we can k-approximate p_i at a set T of points using an oracle for k-approximate counting, it is unlikely that we can use this information (and not the oracle) to efficiently k-approximate p_i at a randomly chosen point r_i not in T.

3.3.1 Approximating a value using an oracle for approximate counting

Recall from section 2.2.2 that for each i, polynomial p_i is defined as

$$p_i(y_i) = \sum_{(y_{i+1},\ldots,y_m)\in\{0,1\}^{m-i}} p(r_1,\ldots,r_{i-1},y_i,\ldots,y_m),$$

where p is a polynomial of degree at most two in each variable and p takes on values in $\{0,1\}$ on $\{0,1\}$-valued assignments to its variables.

Since p takes on non-negative values on $\{0,1\}$-valued assignments to its variables,

computing

$$\mathcal{S}(p) = \sum_{(y_1,\ldots,y_m)\in\{0,1\}^m} p(y_1,\ldots,y_m)$$

is a $\#\mathsf{P}$ problem and hence $\mathcal{S}(p)$ can be k-approximated using an oracle for k-approximate counting.

However, recall that each r_j is randomly chosen so that $0 \leq r_j \leq 2^n$. There is no apparent reason to believe that for any particular y_i, $p(r_1,\ldots,r_{i-1},y_i,\ldots,y_m)$ takes on non-negative values on all $\{0,1\}$-valued assignments to $y_{i+1},\ldots,y_m$. Therefore, computing $p_i(y_i)$ for a particular y_i is not necessarily a $\#\mathsf{P}$ problem, though it is a $\mathsf{P}^{\#\mathsf{P}}$ problem. In fact, as we shall see, if $p(r_1,\ldots,r_{i-1},y_i,\ldots,y_m)$ takes on some positive and some negative values on $\{0,1\}$-valued assignments to $y_{i+1},\ldots,y_m$, then k-approximating $p_i(y_i)$ is as hard as exactly computing $p_i(y_i)$.

Theorem 3.3.1 *Let $k \geq 1$. It is $\#\mathsf{P}$-hard to k-approximate $\mathcal{S}(p)$ for polynomials p of degree at most two in each variable. The problem remains $\#\mathsf{P}$-hard when p is restricted so that it takes on values in $\{-1,0,1\}$ on $\{0,1\}$-valued assignments to its variables.*

Proof. Let $q(x_1,\ldots,x_n)$ be a polynomial of degree at most two in each variable such that q takes on values in $\{0,1\}$ on $\{0,1\}$-valued assignments to its variables. As discussed in Chapter 2, computing $\mathcal{S}(q)$ is a $\#\mathsf{P}$-complete problem.

We will show that using an oracle O for k-approximating $\mathcal{S}(p)$ for polynomials p of degree at most two in each variable, we can compute $\mathcal{S}(q)$ in polynomial time.

Let α be an integer, and let $q_\alpha(x_1,\ldots,x_n) = q(x_1,\ldots,x_n) - \alpha\cdot x_1\cdots x_n$. Note that q_α is of degree at most two in each variable, and $\mathcal{S}(q_\alpha) = \mathcal{S}(q) - \alpha$. So $\mathcal{S}(q_\alpha)$ is positive if $\mathcal{S}(q) > \alpha$, negative if $\mathcal{S}(q) < \alpha$, and 0 if $\mathcal{S}(q) = \alpha$. Using oracle O, we can k-approximate $\mathcal{S}(q_\alpha)$. But this means that we can use oracle O to determine if $\mathcal{S}(q_\alpha)$ is positive, negative, or 0. That is, we can determine if $\mathcal{S}(q) > \alpha$, $\mathcal{S}(q) < \alpha$, or $\mathcal{S}(q) = \alpha$ using O. Then, since we know that $0 \leq \mathcal{S}(q) \leq 2^n$, we can use binary search to find $\mathcal{S}(q)$ in polynomial-time using O.

In fact, for $0 \le \alpha \le 2^n$, we can construct q_α more carefully so that it takes on values in $\{-1, 0, 1\}$ on $\{0, 1\}$-valued assignments to its variables. It is easy to see by induction that for any such α, there exist $\ell \le n + 1$ integers $0 \le e_1 < e_2 < \cdots < e_\ell \le n$ such that

$$\alpha = \sum_{j=1}^{\ell} (-1)^{\ell-j} 2^{e_j}.$$

Consider polynomial $t_\alpha(x_1, \ldots, x_n)$ defined as

$$t_\alpha(x_1, \ldots, x_n) = \sum_{j=1}^{\ell} (-1)^{\ell-j} x_1 x_2 \cdots x_{n-e_j}.$$

Note that t_α is of degree at most one in each variable, and t_α takes on values in $\{0, 1\}$ on $\{0, 1\}$-valued assignments to its variables. Now, for any $z \le n$, we have

$$\sum_{(x_1,\ldots,x_n)\in\{0,1\}^n} x_1 x_2 \cdots x_{n-z} = 2^z.$$

This means that $\mathcal{S}(t_\alpha) = \alpha$. Then, if we let $q_\alpha(x_1, \ldots, x_n) = q(x_1, \ldots, x_n) - t_\alpha(x_1, \ldots, x_n)$, we have that q_α is of degree at most two in each variable, q_α takes on values in $\{-1, 0, 1\}$ on $\{0, 1\}$-valued assignments to its variables, and $\mathcal{S}(q_\alpha) = \mathcal{S}(q) - \alpha$. $\square$

Theorem 3.3.1 tells us that unless the polynomial-time hierarchy collapses, we can't, in general, k-approximate $p_i(y_i)$ in polynomial-time using an oracle for k-approximate counting. However, even if the polynomial-time hierarchy doesn't collapse, it may be the case that there is some efficient way of choosing y_i that ensures that $p(r_1, \ldots, r_{i-1}, y_i, \ldots, y_n)$ takes on non-negative values on all $\{0, 1\}$-valued assignments to $y_{i+1}, \ldots, y_n$. Alternatively, we may work over the rationals, and require that each $r_j \in [0, 1]$; for example, we can randomly choose each r_j from the set $\{\frac{0}{2^n}, \frac{1}{2^n}, \ldots, \frac{2^n}{2^n}\}$. By the construction of p in section 2.2.1, this does ensure that for any $y_i \in [0, 1]$, $p(r_1, \ldots, r_{i-1}, y_i, \ldots, y_n)$ takes on non-negative values on all $\{0, 1\}$-valued assignments to $y_{i+1}, \ldots, y_n$. However, even if we use one of these approaches to efficiently k-approximate p_i at a set T of points using an oracle for k-approximate counting, we are still left with the problem of using these approximations to k-approximate p_i at a randomly chosen point r_i.

3.3.2 Approximating a new value given a set of approximations

Suppose we are given k-approximations to a polynomial p at a set T of points. In this section, we will consider whether we can use these approximations to k-approximate p at a point r that is not in T. We will show that, in general, we will be unable to k-approximate $p(r)$.

Theorem 3.3.2 *Let $k > 1$, let $n \geq 1$, and let $d \geq 0$. Let $x_1, \ldots, x_n \in \mathbb{R}$ be such that $x_1 < x_2 < \cdots < x_n$. Let $p(x)$ be a polynomial of degree at most d, and suppose that for each i, we are given a k-approximation y'_i to p at point x_i. Further, suppose that each y'_i is non-negative. No matter how large n is, the following hold:*

(a) If $d = 0$, we can k-approximate p at any point r.

(b) If $d = 1$, we can k-approximate p at any point r such that $x_1 \leq r \leq x_n$.

(c) If $d \geq 1$, there exist $y'_1, \ldots, y'_n$ such that we will not have enough information to k-approximate p at any point r such that $r < x_1$ or $r > x_n$.

(d) If $d \geq 2$, there exist $y'_1, \ldots, y'_n$ such that we will not have enough information to k-approximate p at any point r that is different from the x_i.

Proof.

(a) If $d = 0$, p is constant and hence any of the y'_i will k-approximate $p(r)$.

(b) We use the line p' passing through (x_1, y'_1) and (x_n, y'_n) as our k-approximation. To see this works, let $y_1 = p(x_1)$ and let $y_n = p(x_n)$. Since p is of degree-1, p is simply the line passing through (x_1, y_1) and (x_n, y_n), and hence

$$p(x) = \frac{y_1(x_n - x) + y_n(x - x_1)}{x_n - x_1}.$$

Similarly,

$$p'(x) = \frac{y'_1(x_n - x) + y'_n(x - x_1)}{x_n - x_1}.$$

Now, we have $y_1/k < y'_1 < k \cdot y_1$ and $y_n/k < y'_n < k \cdot y_n$, and for $x_1 \le r \le x_n$ we have $(x_n - r) \ge 0$ and $(r - x_1) \ge 0$. Then, for $x_1 \le r \le x_n$,

$$p(r)/k = \frac{\frac{y_1}{k}(x_n - r) + \frac{y_n}{k}(r - x_1)}{x_n - x_1} \le \frac{y'_1(x_n - r) + y'_n(r - x_1)}{x_n - x_1} = p'(r)$$

and

$$p'(r) = \frac{y'_1(x_n - r) + y'_n(r - x_1)}{x_n - x_1} \le \frac{k \cdot y_1(x_n - r) + k \cdot y_n(r - x_1)}{x_n - x_1} = k \cdot p(r).$$

That is, $p(r)/k \le p'(r) \le k \cdot p(r)$, and hence $p'(r)$ k-approximates $p(r)$ for $x_1 \le r \le x_n$.

(c) Let $y'_1 = y'_2 = \cdots = y'_n = \alpha$ for some $\alpha > 0$. Consider the line p_1 that passes through the points $(x_1, k \cdot \alpha)$ and $(x_n, \alpha/k)$. This line is consistent with the y'_i; that is, for each i we have $p_1(x_i)/k \le \alpha \le k \cdot p_1(x_i)$. Similarly, the line p_2 that passes through the points $(x_1, \alpha/k)$ and $(x_n, k \cdot \alpha)$ is consistent with the y'_i. With the information we have, we can't determine if p is p_1, p_2, or some other polynomial. Therefore, our k-approximation for $p(r)$ must k-approximate both $p_1(r)$ and $p_2(r)$. Now, note that

$$p_1(x) = \frac{k \cdot \alpha(x_n - x) + \frac{\alpha}{k}(x - x_1)}{x_n - x_1}$$

and

$$p_2(x) = \frac{\frac{\alpha}{k}(x_n - x) + k \cdot \alpha(x - x_1)}{x_n - x_1}.$$

Say $r = x_1 - \epsilon$ for some $\epsilon > 0$. Note that $p_1(r) > 0$ for any such ϵ, but $p_2(r) \le 0$ for sufficiently large ϵ. It is clear that if $p_1(r) > 0$ but $p_2(r) \le 0$, we do not have enough information to k-approximate $p(r)$. So suppose ϵ is small enough that $p_2(r) > 0$. We have

$$p_1(r) = p_1(x_1 - \epsilon) = \frac{k \cdot \alpha(x_n - x_1 + \epsilon) - \frac{\alpha}{k}\epsilon}{x_n - x_1}$$

and

$$p_2(r) = p_2(x_1 - \epsilon) = \frac{\frac{\alpha}{k}(x_n - x_1 + \epsilon) - k \cdot \alpha \cdot \epsilon}{x_n - x_1}.$$

Now, a k-approximation of $p_1(r)$ must be $\geq p_1(r)/k$, while a k-approximation of $p_2(r)$ must be $\leq k \cdot p_2(r)$. But we have

$$p_1(r)/k = \frac{\alpha(x_n - x_1 + \epsilon) - \frac{1}{k^2} \cdot \alpha \cdot \epsilon}{x_n - x_1}$$

and

$$k \cdot p_2(r) = \frac{\alpha(x_n - x_1 + \epsilon) - k^2 \cdot \alpha \cdot \epsilon}{x_n - x_1}.$$

So $p_1(r)/k > k \cdot p_2(r)$, and hence there is no number that k-approximates both $p_1(r)$ and $p_2(r)$. This means that we do not have enough information to k-approximate $p(r)$ for $r < x_1$. By a similar argument, we do not have enough information to k-approximate $p(r)$ for $r > x_n$.

(d) Let $y'_1 = y'_2 = \cdots = y'_n = \alpha$ for some $\alpha > 0$. Let r be a point that is different from the x_i. If $n \leq 2$, it is obvious that we do not have enough information to k-approximate $p(r)$, so suppose $n \geq 3$. If $r < x_1$ or $r > x_n$, the argument given in part (c) shows that we do not have enough information to k-approximate p at r. So suppose $x_1 < r < x_n$. Let j be such that $x_j < r < x_{j+1}$, and assume without loss of generality that $x_n - x_{j+1} \geq x_j - x_1$. This means that $j+1 < n$, since $n \geq 3$. Consider the quadratic p_1 that passes through the points $(x_j, k \cdot \alpha)$, $(x_{j+1}, k \cdot \alpha)$, and $(x_n, \alpha/k)$. We have

$$p_1(x) = k \cdot \alpha - \frac{\alpha(k^2 - 1)(x - x_j)(x - x_{j+1})}{k(x_n - x_j)(x_n - x_{j+1})}.$$

Note that this quadratic opens downward, and hence it is symmetric about its maximum. This means that $p_1(x)$ is increasing on $(-\infty, \frac{x_j + x_{j+1}}{2})$ and decreasing on $(\frac{x_j + x_{j+1}}{2}, \infty)$. Since $x_n - x_{j+1} \geq x_j - x_1$, we have $p_1(x_1) \geq p_1(x_n) = \alpha/k$. Then, for each i, we have $p_1(x_i)/k \leq \alpha \leq k \cdot p_1(x_i)$. That is, p_1 is consistent with the y'_i. Now, consider the quadratic p_2 that passes through the points $(x_j, \alpha/k)$, $(x_{j+1}, \alpha/k)$, and $(x_n, k \cdot \alpha)$. We have

$$p_2(x) = \frac{\alpha}{k} + \frac{\alpha(k^2 - 1)(x - x_j)(x - x_{j+1})}{k(x_n - x_j)(x_n - x_{j+1})}.$$

Note that this quadratic opens upward, and hence it is symmetric about its minimum. This means that $p_2(x)$ is decreasing on $(-\infty, \frac{x_j+x_{j+1}}{2})$ and increasing on $(\frac{x_j+x_{j+1}}{2}, \infty)$. Since $x_n - x_{j+1} \geq x_j - x_1$, we have $p_2(x_1) \leq p_2(x_n) = k \cdot \alpha$. Then, for each i, we have $p_2(x_i)/k \leq \alpha \leq k \cdot p_2(x_i)$ That is, p_2 is consistent with the y_i'. With the information we have, we can't determine if p is p_1, p_2, or some other polynomial. Therefore, our k-approximation for $p(r)$ must k-approximate both $p_1(r)$ and $p_2(r)$. Now, for $x_j < r < x_{j+1}$, we have $p_1(r) > k \cdot \alpha > 0$ and $p_2(r) < \frac{\alpha}{k}$. If $p_2(r) \leq 0$, it is clear that we do not have enough information to k-approximate $p(r)$. So suppose $p_2(r) > 0$. A k-approximation of $p_1(r)$ must be $\geq p_1(r)/k$, while a k-approximation of $p_2(r)$ must be $\leq k \cdot p_2(r)$. But we have $p_1(r)/k > \alpha > k \cdot p_2(r)$. This means that there is no number that k-approximates both $p_1(r)$ and $p_2(r)$, and hence we do not have enough information to k-approximate $p(r)$. $\square$

Chapter 4

How constructive are proofs of hierarchy theorems?

In this chapter, we discuss the techniques used to prove hierarchy theorems, and the extent to which such proofs are constructive. The techniques we discuss include diagonalization, used by Hartmanis and Stearns [HS65] to prove the Time Hierarchy theorem, padding arguments, used to to prove tighter time hierarchy results, and the idea of optimal algorithms, used by Fortnow and Santhanam [FS04] to prove a hierarchy theorem for bounded-error probabilistic polynomial time with one bit of advice.

4.1 Hierarchy theorems

A fundamental question in computational complexity is whether the power of a computational model increases when allowed to use more of a particular resource, such as time, space, randomness, nondeterminism, or nonuniformity. A theorem that asserts such an increase is known as a *hierarchy theorem*, as it implies that by increasing the resource in question, we get a hierarchy of progressively larger complexity classes $\mathcal{C}_1 \subsetneq \mathcal{C}_2 \subsetneq \mathcal{C}_3$, etc. Beyond simply knowing that such a hierarchy exists, it seems natural to want to know, for each i, a language L_i such that $L_i \in \mathcal{C}_{i+1} - \mathcal{C}_i$. Furthermore, given a Turing machine

M_i that satisfies the resource bounds of class $\mathcal{C}_i$, we might want to know how hard it is to find an input x on which M_i differs from L_i. Since $L_i \notin \mathcal{C}_i$, it is obvious that M_i will differ from L_i on some input. However, if it is hard to find such inputs, we may find that for a particular application it is sufficient to use a $\mathcal{C}_i$ machine such as M_i to "decide" L_i.

As we shall see, there are both constructive and non-constructive techniques to prove hierarchy theorems. We consider a proof of a hierarchy $\mathcal{C}_1 \subsetneq \mathcal{C}_2 \subsetneq \mathcal{C}_3 \subsetneq \ldots$ to be constructive if for each i, it shows how to construct a language L_i such that $L_i \in \mathcal{C}_{i+1} - \mathcal{C}_i$, and shows how hard it is, given a machine M_i that satisfies the resource bounds of class $\mathcal{C}_i$, to find an input x on which M_i differs from L_i.

4.2 The Time Hierarchy theorem

The first hierarchy theorem in computational complexity was the Time Hierarchy theorem, proven by Hartmanis and Stearns [HS65] and made tighter by Hennie and Stearns [HS66]. They show that for all time-constructible[1] functions $f, g : \mathbb{N} \to \mathbb{N}$ such that

$$\lim_{n\to\infty} \frac{f(n) \log f(n)}{g(n)} = 0,$$

we have $\mathsf{DTIME}(f(n)) \subsetneq \mathsf{DTIME}(g(n))$. For any function $h(n)$, $\mathsf{DTIME}(h(n))$ denotes the set of languages that are decided by a deterministic Turing machine that runs in time $h(n)$.

The Time Hierarchy theorem is proven using diagonalization. This means that the proof shows how, given f and g, we can construct a machine M running in time g such that for every machine M' that runs in time f, there is some input x (where x is constructed from the encoding of M') on which M simulates M' on x and "does the opposite". Since the input x on which M "does the opposite" of M' is constructed from the encoding of M', it is easy to find this input given M', and hence the proof of the Time

[1] A function $f : \mathbb{N} \to \mathbb{N}$ is *time-constructible* if there exists a Turing machine M that halts in exactly $f(|x|)$ steps on every input x.

Hierarchy theorem is constructive. In general, proofs by diagonalization are constructive.

To elaborate on the ideas we have discussed, we examine the proof of the Time Hierarchy theorem for the case $f(n) = n^2$ and $g(n) = n^2 \log^2 n$.

Note that there exists a Turing machine U, known as the universal Turing machine, such that $L(U) = \{\langle x, y\rangle \mid M_x \text{ accepts } y\}$. Furthermore, as shown by Hennie and Stearns [HS66], for each machine M_x there is a constant c_x such that for every string y, if M_x runs in time t on input y then U runs in time $c_x t \log t$ on input $\langle x, y\rangle$. The constant c_x, which we will refer to as the simulation constant, depends on the number of states and the number of tapes that M_x has.

Consider the language L_1 defined as follows:

$$L_1 = \{1^t 0x \mid U \text{ does not accept } \langle x, 1^t 0x\rangle \text{ within } |1^t 0x|^2 \log^2 |1^t 0x| \text{ steps}\}.$$

It is clear that L_1 can be decided in time $n^2 \log^2 n$. Now, suppose we are given a machine M_z that runs in time n^2. Let c_z be the simulation constant for M_z. Let t be large enough so that $c_z|1^t 0z|^2 \log |1^t 0z|^2 < |1^t 0z|^2 \log^2 |1^t 0z|$. Since M_z runs in time $|1^t 0z|^2$ on input $|1^t 0z|$, U runs in time $c_z|1^t 0z|^2 \log |1^t 0z|^2$ on input $\langle z, 1^t 0z\rangle$. Then, by the definitions of L_1 and U, we have that M_z differs from L_1 on input $1^t 0z$. So, for every machine M that runs in time n^2, there is an input on which this machine differs from L_1, and hence L_1 cannot be decided in time n^2.

4.3 Tighter time hierarchy results

Existing hierarchy theorems can be made tighter using padding arguments. Suppose we have classes $\mathcal{C}_1 \subseteq \mathcal{C}_2 \subseteq \mathcal{C}_3$, we have proven $\mathcal{C}_1 \subsetneq \mathcal{C}_3$, and we want to show that $\mathcal{C}_1 \subsetneq \mathcal{C}_2$. In a padding argument proof of this, we derive a contradiction from the assumption that $\mathcal{C}_1 = \mathcal{C}_2$. Essentially, the padding argument allows us to take the "small collapse" $\mathcal{C}_1 = \mathcal{C}_2$ and derive a "larger collapse" $\mathcal{C}_1 = \mathcal{C}_3$. The main idea is that if we have a language $L \in \mathcal{C}_3$, then a sufficiently padded version of L will be in $\mathcal{C}_2$ and hence in $\mathcal{C}_1$.

But using a $\mathcal{C}_1$ machine for the padded language, we can get a $\mathcal{C}_2$ machine for L, and hence $L \in \mathcal{C}_1$. This argument is non-constructive.

More concretely, consider functions $f(n) = n^2$ and $g(n) = n^2 \log n$. Note that $\lim_{n\to\infty} \frac{f(n)\log f(n)}{g(n)} > 0$. It is not known how to use diagonalization to prove $\mathsf{DTIME}(n^2) \subsetneq \mathsf{DTIME}(n^2 \log n)$. However, recall that we proved $\mathsf{DTIME}(n^2) \subsetneq \mathsf{DTIME}(n^2 \log^2 n)$ using diagonalization in section 4.2. Then, using a padding argument, we can show $\mathsf{DTIME}(n^2) \subsetneq \mathsf{DTIME}(n^2 \log n)$.

Let L_1 be as defined in section 4.2. That is,

$$L_1 = \{1^t 0x \mid U \text{ does not accept } \langle x, 1^t 0x \rangle \text{ within } |1^t 0x|^2 \log^2 |1^t 0x| \text{ steps}\}.$$

Recall that $L_1 \in \mathsf{DTIME}(n^2 \log^2 n) - \mathsf{DTIME}(n^2)$.

Suppose for the sake of contradiction that $\mathsf{DTIME}(n^2) = \mathsf{DTIME}(n^2 \log n)$. We will show that $L_1 \in \mathsf{DTIME}(n^2)$.

Let L_2 be the language defined as follows:

$$L_2 = \{1^u 0y \mid y \in L_1 \text{ and } |1^u 0y| \geq |y|\sqrt{\log |y|}\}.$$

Note that $L_2 \in \mathsf{DTIME}(n^2 \log n)$, since we can construct a machine that on input $1^u 0y$ checks if $|1^u 0y| \geq |y|\sqrt{\log |y|}$ and, if so, runs the $\mathsf{DTIME}(n^2 \log^2 n)$ machine for L_1 on y. But then, by assumption, we have $L_2 \in \mathsf{DTIME}(n^2)$. This means that $L_1 \in \mathsf{DTIME}(n^2 \log n)$, since we can construct a machine that on input y computes u large enough so that $|1^u 0y| \geq |y|\sqrt{\log |y|}$ and then runs the $\mathsf{DTIME}(n^2)$ machine for L_2 on $1^u 0y$. Then, by assumption, we have $L_1 \in \mathsf{DTIME}(n^2)$, and this is a contradiction.

Note that this proof is non-constructive. While the proof tells us there is some language L such that $L \in \mathsf{DTIME}(n^2 \log n) - \mathsf{DTIME}(n^2)$, it does not tell us unconditionally what this language is. Instead, we are left with two cases.

If $L_1 \notin \mathsf{DTIME}(n^2 \log n)$, then we must have $L_2 \notin \mathsf{DTIME}(n^2)$, and hence $L_2 \in \mathsf{DTIME}(n^2 \log n) - \mathsf{DTIME}(n^2)$. However, it is not clear how hard it is, given a machine M that runs in time n^2, to find an input x on which M differs from L_2.

If $L_1 \in \mathsf{DTIME}(n^2 \log n)$, then we have $L_1 \in \mathsf{DTIME}(n^2 \log n) - \mathsf{DTIME}(n^2)$. From section 4.2, we know how, given a machine M that runs in time n^2, we can find an input x on which M differs from L_1.

4.4 A hierarchy theorem for probabilistic polynomial time with small advice

As we saw in section 4.2, a hierarchy theorem for deterministic polynomial time was proven four decades ago. However, no such theorem is known for bounded-error probabilistic polynomial time (BPP). This means that it may be the case that a bounded-error probabilistic Turing machine cannot solve more problems in polynomial time than it can solve in linear time. The main difficulty in proving a hierarchy theorem for BPP is that there is no known way of enumerating bounder-error probabilistic Turing machines, yet such an enumeration is necessary for diagonalization.

Hierarchy theorems have been proven for bounded-error probabilistic polynomial time with small advice. Barak [Bar02] shows that for every d, $\mathsf{BPTIME}(n^d)/\log\log n \subsetneq \mathsf{BPTIME}(n^{d+1})/\log\log n$. Fortnow and Santhanam [FS04] improve this by reducing the advice from $\log\log n$ bits to one bit. These proofs rely on the idea of an optimal algorithm. An algorithm for a language L is optimal if it is at most polynomially slower than any other algorithm for L. Languages that have instance checkers also have optimal bounded-error probabilistic algorithms. If such an algorithm runs in superpolynomial time, we immediately get a hierarchy for bounded-error superpolynomial time. Then, we can use a padding argument to get a hierarchy for BPP. However, the amount of padding needed is not known to be computable in uniform BPP, and hence some nonuniformity must be introduced.

We will present Fortnow and Santhanam's proof, and then discuss if this proof can be made more constructive.

We begin by formally defining uniform and nonuniform BPTIME.

Definition 4.4.1 (BPTIME). Let L be a language and let $t : \mathbb{N} \to \mathbb{N}$ be a function. We say that $L \in \mathsf{BPTIME}(t)$ if there exists a probabilistic Turing machine M such that on all inputs x, the probability that M halts within $t(|x|)$ steps and outputs $L(x)$ is at least $2/3$.

Note that in this definition of $\mathsf{BPTIME}(t)$, machine M is not required to always halt within time t. However, if t is time-constructible, we can add a "clock" to M and force it to halt within time t.

Definition 4.4.2 (Nonuniform BPTIME). Let L be a language and let $s : \mathbb{N} \to \mathbb{N}$ and $t : \mathbb{N} \to \mathbb{N}$ be functions. We say that $L \in \mathsf{BPTIME}(t)/s$ if there exists a probabilistic Turing machine M and an infinite sequence of strings $\{y\}_n$, $|y_n| \leq s(n)$, such that on all inputs $\langle x, y_{|x|} \rangle$, the probability that M halts within $t(|x|)$ steps and outputs $L(x)$ is at least $2/3$.

Note that in this definition of nonuniform BPTIME, machine M is not required to be a bounded-error machine when given advice strings other than the y_i.

Theorem 4.4.1 (Fortnow, Santhanam [FS04]). *Let $d \geq 1$. Then* $\mathsf{BPTIME}(n^d)/1 \subsetneq \mathsf{BPP}/1$.

Corollary 4.4.2 *Let $d \geq 1$. Then* $\mathsf{BPTIME}(n^d)/1 \subsetneq \mathsf{BPTIME}(n^{d+1})/1$.

We will give the proof of Theorem 4.4.1. The corollary can be proven using a padding argument.

4.4.1 Proof of Theorem 4.4.1

We will consider two cases, $\mathsf{BPP} = \mathsf{PSPACE}$ and $\mathsf{BPP} \subsetneq \mathsf{PSPACE}$. In the first case, we will be able to use diagonalization. In the second case, we will use the idea of an optimal algorithm for a PSPACE-complete language.

Case 1: $\mathsf{BPP} = \mathsf{PSPACE}$

Lemma 4.4.3 *If* $\mathsf{BPP} = \mathsf{PSPACE}$*, then* $\mathsf{BPP} \not\subseteq \mathsf{BPTIME}(n^d)/\log n$.

Proof. Consider the following language L_D:

$$L_D = \{x \mid \Pr[M_x \text{ accepts } x \text{ within } |x|^d \text{ steps}] < 1/2\}.$$

Note that $L_D \in \mathsf{PSPACE}$, and hence $L_D \in \mathsf{BPP}$. We need to show that $L_D \notin \mathsf{BPTIME}(n^d)/\log n$.

Suppose we are given a probabilistic machine M_y and an infinite sequence of strings $\{s\}_n$, $|s_n| \leq \log n$, such that on all inputs $\langle x, s_{|x|}\rangle$, we either have that M_y accepts within $|x|^d$ steps with probability at least $2/3$ or we have that M_y rejects within $|x|^d$ steps with probability at least $2/3$.

We need to find an input z on which $M_y(\langle z, s_{|z|}\rangle)$ differs from L_D. Now, for sufficiently large n, there exists a string z of length n such that M_z is M_y with the string s_n "hardwired" into it. Then, for all inputs x of length n, $M_z(x)$ behaves exactly like $M_y(\langle x, s_n\rangle)$. In particular, $M_z(z)$ behaves exactly like $M_y(\langle z, s_n\rangle)$. By the definition of L_D, $z \in L_D$ if and only if M_z rejects z with probability at least $1/2$. This means that $z \in L_D$ if and only if M_y rejects $\langle z, s_n\rangle$ with probability at least $1/2$. That is, $M_y(\langle z, s_{|z|}\rangle)$ differs from L_D. $\square$

It follows from Lemma 4.4.3 that $\mathsf{BPTIME}(n^d)/1 \subsetneq \mathsf{BPP}/1$. In fact, it also follows that $\mathsf{BPTIME}(n^d) \subsetneq \mathsf{BPP}$. That is, we get a hierarchy for uniform BPP in this case.

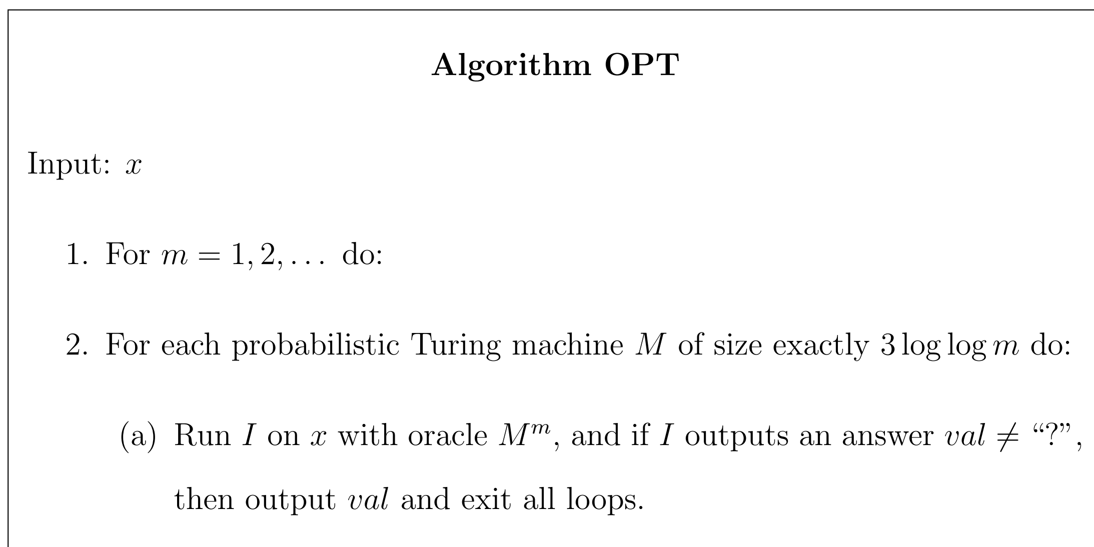
Case 2: $\mathsf{BPP} \subsetneq \mathsf{PSPACE}$

We will give an optimal bounded-error probabilistic algorithm for a PSPACE-complete language L. As noted in Chapter 3, every PSPACE-complete language L has an instance checker I. We will need L to have an instance checker I such that on any input x, I only makes queries of length $|x|$. Trevisan and Vadhan [TV02] show that there is a PSPACE-complete language $L \in \mathsf{DTIME}(2^{2n})$ that has such an instance checker I. We will further

require that I's error probability (property 3 in Definition 3.2.1) is less than $\frac{1}{2^n}$. We can always achieve this by running I some polynomial number of times and then taking the majority answer.

For a Turing machine M and a constant c, we will let M^c denote M restricted to c steps. This means that if M doesn't halt after c steps, we will consider M to have halted and rejected.

We will assume that the encoding of a Turing machine can be padded. This means that if a Turing machine has an encoding of length n, it also has an encoding of length m for each $m > n$.



Algorithm OPT

Input: x

1. For $m = 1, 2, \ldots$ do:
2. For each probabilistic Turing machine M of size exactly $3 \log \log m$ do:
   (a) Run I on x with oracle M^m, and if I outputs an answer $val \neq$ "?", then output val and exit all loops.

Figure 4.1: The optimal algorithm for L

The optimal probabilistic algorithm for L is given in Figure 4.1.

We will now give some notation for the running time of **OPT** and then bound this running time.

For each string x, let $t(x)$ be the smallest number such that on input x, the probability that **OPT** halts and outputs $L(x)$ within $t(x)$ steps is at least 2/3. Then, for each n, let $T(n) = \max_{x \in \{0,1\}^n} t(x)$.

Lemma 4.4.4 *For sufficiently large n, $T(n) < 2^{5n}$.*

Proof. Recall that $L \in \mathsf{DTIME}(2^{2n})$. That is, there exists a deterministic Turing machine M running in time 2^{2n} that decides L. If **OPT** runs long enough, it will run I with oracle $M^{2^{2n}}$, and at this point **OPT** will always halt and output the correct answer.

Let n be sufficiently large so that the smallest encoding of M is of length at most $3 \log\log 2^{2n}$. On an input of length n, consider bounding the number of steps needed by **OPT** to get to machine $M^{2^{2n}}$ (assuming it doesn't halt before this). Note that when m is the smallest integer $\geq 2^{2n}$ such that $\log\log m$ is an integer, **OPT** will run I with oracle M^m (equivalent to $M^{2^{2n}}$). By this point, **OPT** will have tried every machine of size less than $3 \log\log 2^n$ and possibly some other machines of size $\lceil 3 \log\log 2^n \rceil$. That is, **OPT** will have tried at most $16n^3$ machines. For each such machine, **OPT** will run I. I runs in polynomial time, so say I's running time is $p(n)$. This means that I will make at most $p(n)$ oracle queries to each machine. Since each machine tried by **OPT** is restricted to at most m steps, it will take at most m steps to answer each such query. Since m is the smallest integer $\geq 2^{2n}$ such that $\log\log m$ is an integer, we have $m < 2^{4n}$. Now, putting all of this together, we have that **OPT** gets to machine $M^{2^{2n}}$ within $16n^3 \cdot p(n) \cdot 2^{4n}$ steps. For sufficiently large n, this is less than 2^{5n} steps.

Now consider the probability that **OPT** outputs an incorrect answer before it tries machine $M^{2^{2n}}$. Again, **OPT** tries at most $16n^3$ machines before it gets to $M^{2^{2n}}$. Note that while I's error probability is defined with respect to oracles, we can view a probabilistic machine as selecting all the random bits it needs for $p(n)$ runs before I begins to query it. Then, as long as I doesn't make the same query twice, the probabilistic machine is equivalent to some oracle. So, for each such machine, I (and hence **OPT**) outputs an incorrect answer with probability less than $\frac{1}{2^n}$. Then, the probability that **OPT** outputs an incorrect answer before it tries machine $M^{2^{2n}}$ is less than $16n^3/2^n$. For sufficiently large n, this is less than $1/3$. □

Note that by the definition of T, we have $L \in \mathsf{BPTIME}(T)$. On the other hand, since L is PSPACE-complete and since we are assuming $\mathsf{BPP} \subsetneq \mathsf{PSPACE}$, we have $L \notin \mathsf{BPP}$.

This means that T is not bounded by any polynomial.

We next show that **OPT** really is an optimal algorithm.

Lemma 4.4.5 *Let $b > 0$. Then $L \notin \mathsf{BPTIME}(n^b + T^{1/10})/2 \log\log T$.*

Proof. Fix b.

Suppose for the sake of contradiction that $L \in \mathsf{BPTIME}(n^b + T^{1/10})/2 \log\log T$.

We will restrict our attention to those n such that $T(n)^{1/10} > \max(n^{b+2}, p(n)^3)$, where $p(n)$ is the running time of instance checker I. Since T is not bounded by any polynomial, there are infinitely many such n.

For sufficiently large n, there exists a probabilistic Turing machine M_1 whose encoding has size at most $2.1 \log\log T(n)$ such that on all inputs x of length n, the probability that M_1 halts within $n^b + T(n)^{1/10}$ steps and outputs $L(x)$ is at least $2/3$.

Furthermore, if n is sufficiently large, there exists a probabilistic Turing machine M_2 whose encoding has size at most $2.2 \log\log T(n)$ such that M_2 is M_1 with success probability amplified so that it is exponentially close to 1. We can think of M_2 as running M_1 in parallel cn times for some constant c until the majority of these runs output the same thing. So we have that on all inputs x of length n, the probability that M_2 halts within $cn(n^b + T(n)^{1/10})$ steps and outputs $L(x)$ is at least $1 - 1/2^n$.

Note that if n is sufficiently large, we have

$$\begin{aligned} cn(n^b + T(n)^{1/10}) &\leq n^{b+2} + n^2 T(n)^{1/10} \\ &< T(n)^{1/10} + T(n)^{2/10} \qquad \text{since } T(n)^{1/10} > n^{b+2} \\ &< 2T(n)^{2/10} \\ &< T(n)^{1/4} \end{aligned}$$

So on all inputs x of length n, the probability that M_2 halts within $T(n)^{1/4}$ steps and outputs $L(x)$ is at least $1 - 1/2^n$.

However, this means that on all inputs x of length n, the probability that **OPT** halts and outputs $L(x)$ within $(\log \sqrt{T(n)})^3 \cdot p(n) \cdot \sqrt{T(n)}$ steps is at least $1 - 1/n$.

To see this, suppose **OPT** is run on an input x of length n, and consider what happens in **OPT** when m is the smallest integer such that $m \geq T(n)^{1/4}$ and $\log\log m$ is an integer. **OPT** will run I with oracle ${M_2}^m$, since if n is sufficiently large we have $3\log\log m \geq 3\log\log T(n)^{1/4} > 2.2\log\log T(n)$. Now, I makes at most $p(n)$ queries to ${M_2}^m$. By the union bound, the probability that ${M_2}^m$ does not err on any of these queries is at least $1 - p(n)/2^n$. In this case, I will output $L(x)$, and so will **OPT**. Using reasoning similar to that used in the proof of Lemma 4.4.4, **OPT** will need at most $(\log\sqrt{T(n)})^3 \cdot p(n) \cdot \sqrt{T(n)}$ steps to get to machine ${M_2}^m$.

Again, using reasoning similar to that used in the proof of Lemma 4.4.4, the probability that **OPT** outputs an incorrect answer before it tries machine ${M_2}^m$ is less than $(\log\sqrt{T(n)})^3/2^n$. But recall from Lemma 4.4.4 that if n is sufficiently large, $T(n) < 2^{5n}$. So the probability that **OPT** outputs an incorrect answer before it tries machine ${M_2}^m$ is less than $(5n/2)^3/2^n$.

So, on all inputs x of length n, the probability that **OPT** halts and outputs $L(x)$ within $(\log\sqrt{T(n)})^3 \cdot p(n) \cdot \sqrt{T(n)}$ steps is at least $1 - (p(n) + (5n/2)^3)/2^n > 2/3$. Since $(\log\sqrt{T(n)})^3 \cdot p(n) \cdot \sqrt{T(n)} < T(n)$, this contradicts the definition of $T(n)$. □

We now define a language L_{pad} that is a padded version of L.

$$L_{pad} = \{x\#1^y \mid x \in L,\ y = 2^{2^z} \text{ for some } z,\ y > |x|,\ y + |x| + 1 \geq T(|x|)^{\frac{1}{30d}}\}$$

We will show that $L_{pad} \in \mathsf{BPP}/1$ but $L_{pad} \notin \mathsf{BPTIME}(n^d)/\log\log n$.

Lemma 4.4.6 $L_{pad} \in \mathsf{BPP}/1$.

Proof. Consider strings of the form $x\#1^y$ which have some fixed length n. In order for $y > |x|$, it must be the case that $n/2 \leq y \leq n-1$. But note that for any n, there is at most one z such that $n/2 \leq 2^{2^z} \leq n-1$. This means that for each n, there is at most one $y > |x|$ such that $y = 2^{2^z}$ for some z. If such a y exists, denote it y_n.

Now, consider the following probabilistic algorithm for L_{pad} that uses 1 bit of advice. On input $x\#1^y$, say the input has length n, the algorithm will receive advice bit 0 if both y_n exists and $n \geq T(n - y_n - 1)^{\frac{1}{30d}}$, and advice bit 1 otherwise. If the algorithm receives advice bit 1, then it immediately halts and rejects. Otherwise, the algorithm checks if $y > |x|$ and if not it rejects. Then, the algorithm checks if $\log \log y$ is an integer, and if not it rejects. If the algorithm hasn't rejected, we must have $y = y_n$. Then, by the advice bit, we have $n \geq T(|x|)^{\frac{1}{30d}}$ and hence we need to check if $x \in L$. We can do this by running **OPT** on x. If **OPT** accepts x then the algorithm halts and accepts, and otherwise the algorithm halts and rejects. We know that with probability at least 2/3, **OPT** halts within $T(|x|) \leq n^{30d}$ steps and correctly tells us whether $x \in L$. Furthermore, the initial checks can be done in time n^2. So, the probability that the algorithm halts within $n^{30d} + n^2$ steps and outputs $L_{pad}(x\#1^y)$ is at least 2/3. This means that $L_{pad} \in \mathsf{BPP}/1$. $\square$

Lemma 4.4.7 *$L_{pad} \notin \mathsf{BPTIME}(n^d)/\log \log n$.*

Proof. Suppose for the sake of contradiction that $L_{pad} \in \mathsf{BPTIME}(n^d)/\log \log n$. We will show that $L \in \mathsf{BPTIME}(n^b + T^{1/10})/2 \log \log T$ for some b, contradicting Lemma 4.4.5.

Consider a probabilistic algorithm for L that works as follows on an input x. Say $|x| = n$. Let z be the smallest integer such that $2^{2^z} > n$ and $2^{2^z} + n + 1 \geq T(n)^{\frac{1}{30d}}$. Such a z can be specified using $\log \log T$ bits of advice. Let $y = 2^{2^z}$. Let $x' = x\#1^y$. Note that $|x'| \leq \max(n^2 + n + 1, T(n)^{\frac{2}{30d}})$. We would like to run the $\mathsf{BPTIME}(n^d)/\log \log n$ machine for L_{pad} on x'. But this will require $\log \log |x'|$ bits of advice. So our algorithm will also need to be provided with these bits as advice. We then run the $\mathsf{BPTIME}(n^d)/\log \log n$ machine for L_{pad} on x', and accept or reject as that machine does. Running this machine takes time $|x'|^d \leq (n^2 + n + 1 + T(n)^{\frac{2}{30d}})^d$. The total advice our algorithm needs is at most $2 \log \log T$. Finally, there exists a b such that, with probability at least 2/3, our algorithm halts and outputs $L(x)$ within $n^b + T(n)^{1/10}$ steps. $\square$

It follows from Lemma 4.4.6 and Lemma 4.4.7 that $\mathsf{BPP}/1 \not\subseteq \mathsf{BPTIME}(n^d)/\log\log n$, and hence $\mathsf{BPTIME}(n^d)/1 \subsetneq \mathsf{BPP}/1$.

4.4.2 Making the proof of Theorem 4.4.1 more constructive

The proof of Theorem 4.4.1 is non-constructive. It does not unconditionally give a language L such that $L \in \mathsf{BPP}/1 - \mathsf{BPTIME}(n^d)/1$. Instead, it gives one such language for the case $\mathsf{BPP} = \mathsf{PSPACE}$, and another such language for the case $\mathsf{BPP} \subsetneq \mathsf{PSPACE}$. Now, for the language L_D given in the case $\mathsf{BPP} = \mathsf{PSPACE}$, the proof does show how, given a $\mathsf{BPTIME}(n^d)/\log n$ machine M, we can easily find an input x on which M differs from L_D. On the other hand, a similar result is not shown for the language L_{pad} given in the case $\mathsf{BPP} \subsetneq \mathsf{PSPACE}$.

We now consider whether the proof can be made more constructive. We first discuss whether the two cases can be replaced with a single case. We then discuss whether Case 2 can be made more constructive.

Replacing the two cases with a single case

In Case 1, diagonalization is used to prove a hierarchy. In general, we cannot use this technique with bounded-error probabilistic machines. Given an enumeration of probabilistic machines, we aren't able to determine if a particular machine has bounded error on any particular input. So, if the probabilistic machine we construct by diagonalization simply runs other probabilistic machines and "does the opposite", the machine we construct will not have bounded error. However, in Case 1 we are assuming that $\mathsf{BPP} = \mathsf{PSPACE}$. In PSPACE (and, in fact, in $\mathsf{P}^{\#\mathsf{P}}$), we can (exactly) compute the acceptance probability of any probabilistic polynomial time machine on any input. Therefore, in Case 1, when the machine we construct is diagonalizing against a particular probabilistic machine on a particular input, it can compute the machine's acceptance probability on that input and then behave in a way that ensures that its own acceptance probability is bounded away

from 1/2. However, without assuming that $\mathsf{BPP} = \mathsf{PSPACE}$ (or, alternatively, making a similar assumption like $\mathsf{BPP} = \mathsf{P}^{\#\mathsf{P}}$), it is not known how to construct a BPP machine that diagonalizes against probabilistic machines. Therefore, we must make such an assumption in order to use diagonalization.

In Case 2, an optimal algorithm is used to prove a hierarchy. All languages that have instance checkers also have optimal bounded-error probabilistic algorithms. An optimal bounded-error probabilistic algorithm A for a language L is an algorithm that is at most polynomially slower than any other bounded-error probabilistic algorithm for L. That is, there is some constant $\epsilon < 1$ such that if A runs in time t then any other algorithm for L runs in time at least t^ϵ. This means that L can be decided in time t but not in time $o(t^\epsilon)$, and hence we get a hierarchy. However, this hierarchy is trivial if t^ϵ is sublinear (since it is obvious that a non-trivial language L cannot be decided in sublinear time). To ensure that the hierarchy we get is non-trivial, we must make an assumption about the probabilistic hardness of L. That is, we must make an assumption like $L \notin \mathsf{BPP}$. However, it is not even known if $\mathsf{EXP} \not\subseteq \mathsf{BPP}$, and so, unconditionally, it is not known if there is an instance-checkable language $L \notin \mathsf{BPP}$. Therefore, we assume $\mathsf{BPP} \subsetneq \mathsf{PSPACE}$ (or, alternatively, $\mathsf{BPP} \subsetneq \mathsf{P}^{\#\mathsf{P}}$) in order to guarantee the existence of such a language L.

Since the techniques used in each case - diagonalization in Case 1 and the use of an optimal algorithm in Case 2 - appear to require the assumption made in each such case (or at least a similar assumption), some other technique is likely needed in order to get a proof that doesn't use cases. For example, in [FST05], Fortnow, Santhanam, and Trevisan use a *nonuniform* optimal algorithm to prove (without cases) a hierarchy for BPP with $\log^2 n$ bits of advice.

Making Case 2 more constructive

Consider the language L_{pad} defined in Case 2. In Lemma 4.4.7, it is shown that $L_{pad} \notin \mathsf{BPTIME}(n^d)/\log\log n$. However, this proof is non-constructive. It simply shows that

if $L_{pad} \in \mathsf{BPTIME}(n^d)/\log\log n$, then $L \in \mathsf{BPTIME}(n^b + T^{1/10})/2\log\log T$ for some b, contradicting Lemma 4.4.5. The proof of Lemma 4.4.5 is also non-constructive, as it simply shows that if $L \in \mathsf{BPTIME}(n^b + T^{1/10})/2\log\log T$ for some b, then the definition of T is contradicted. Finally, the definition of T is itself non-constructive. Recall that for each n, $T(n) = \max_{x \in \{0,1\}^n} t(x)$, where $t(x)$ is the smallest number such that on input x, the probability that **OPT** halts and outputs $L(x)$ within $t(x)$ steps is at least 2/3. The definition of $T(n)$ does not give any insight into how to find, for each n, an x such that $T(n) = t(x)$. In fact, it is precisely because $T(n)$ is not known to be easily computable that the BPP algorithm for L_{pad} needs an advice bit; that is, finding an efficient algorithm for computing $T(n)$ would allow us to prove a hierarchy for uniform BPP.

Note that if we make the proof of Lemma 4.4.5 more constructive, we can also make the proof of Lemma 4.4.7 more constructive. That is, if we know how, given a $\mathsf{BPTIME}(n^b + T^{1/10})/2\log\log T$ machine M, to find an input x on which M differs from L, we can use this approach to find an input x' on which a $\mathsf{BPTIME}(n^d)/\log\log n$ machine M' differs from L_{pad}. Given such a machine M', we construct (using the ideas given in the proof of Lemma 4.4.7) a $\mathsf{BPTIME}(n^b + T^{1/10})/2\log\log T$ machine M, and we find an input x on which M differs from L. Then, the input x' on which M' differs from L_{pad} will be some sufficiently padded version of x. Of course, the exact amount of padding needed will depend on the apparently difficult-to-compute value of $T(|x|)$.

We now focus on making the proof of Lemma 4.4.5 more constructive. Fix b, and suppose that we are given the encoding of a $\mathsf{BPTIME}(n^b + T^{1/10})/2\log\log T$ machine M and (oracle access to) an infinite sequence of advice strings y_n such that $|y_n| \leq 2\log\log T(n)$ for each n. Our goal is to find an input x on which M with advice $y_{|x|}$ differs from L. Consider the following probabilistic procedure to find such an x.

As in the proof of Lemma 4.4.5, we need to restrict our attention to those n such that $T(n)^{1/10} > \max(n^{b+2}, p(n)^3)$. Since $T(n)$ is not bounded by any polynomial, we know that there are infinitely many such n. We also know by Lemma 4.4.4 that $T(n) < 2^{5n}$

for sufficiently large n. However, we don't have an efficient procedure to find these n. Therefore, we will assume that we have an oracle O that on input m returns an n such that $n \geq m$ and $2^{n/2} > T(n)^{1/10} > \max(n^{b+2}, p(n)^3)$.

Now, we would like to find an n large enough so that there is a machine M' of encoding size $2.2 \log\log T(n)$ that is machine M with the advice string y_n "hardwired" into it and that has error probability reduced to $1/2^{2n}$ (on inputs of length n) at the expense of having its running time increase by a factor of cn for some constant c. M' is similar to the machine M_2 in the proof, except that we are requiring its error probability to be $1/2^{2n}$ rather than $1/2^n$ for reasons that we will see later. In order for M' to exist, we need n to be such that $|M|$ is at most, say, $0.1 \log\log T(n)$. But, again, we can't compute $T(n)$. So, instead, we will select an n that satisfies the stronger requirement that $|M| < 0.1 \log\log n$. That is, we will select an $n > 2^{2^{|M|}}$. We will query our oracle O on input $2^{2^{|M|}} + 1$ to find such an n. Now, as the proof argues for machine M_2, machine M' cannot correctly decide L for all inputs of length n, since otherwise **OPT** will need time less than $T(n)$ to decide L. So we know that there is an input x of length n on which M' (and also M with advice y_n) differs from L. Since we don't know anything about which input this is, we are left with having to iterate through every string x of length n, checking if M' differs from L on input x.

On each input x of length n, we will do the following. We will first check if $x \in L$. We can use the $\mathsf{DTIME}(2^{2n})$ machine for L to check this. Alternatively, we can assume that we have an oracle for PSPACE. The advantage of making such an assumption is that our analysis can then more easily be adapted to a proof that uses, say, a $\mathsf{P}^{\#\mathsf{P}}$-complete instance-checkable language L instead of a PSPACE-complete language. After checking if $x \in L$, we run M' on x. If M' outputs a value different from $L(x)$, we know that with high probability, x is an input on which M' differs from L.

Since we are running M' on each of the 2^n strings of length n, the probability that M' errs at least once is, by the union bound, at most $2^n/2^{2n} = 1/2^n$. Therefore, with

probability at least $1 - 1/2^n$, our procedure will correctly identify the input (or inputs) of length n on which M' differs from L.

Now, consider the running time of our procedure. We make a single call to our oracle O. We make 2^n calls to our PSPACE oracle (to check if each string of length n is in L). We run M' 2^n times on inputs of length n. On each such run, M' halts within $cn(n^b + T^{1/10})$ steps with probability at least $1 - 1/2^{2n}$. This means that with probability at least $1 - 1/2^n$, M' halts within $cn(n^b + T^{1/10})$ steps on every such run. So, with probability at least $1 - 1/2^n$, the time needed to run M' on every input of length n is at most $2^n \cdot cn(n^b + T^{1/10})$. This is at most 2^{2n}, since $2^{n/2} > T(n)^{1/10} > \max(n^{b+2}, p(n)^3)$. But recall that n is double exponential in $|M|$. So, our probabilistic procedure takes time triple exponential in the size of M to find an input x on which M with advice $y_{|x|}$ differs from L.

Chapter 5

Open questions

In this chapter, we summarize the open questions that are discussed in this thesis.

What are suitable approaches for showing that #P has polynomial-size circuits?

As we discussed, the interactive proofs for #P are the simplest known non-trivial interactive proofs for NP-hard languages. This makes the class #P an ideal starting point for investigating whether interaction can be converted into nonuniformity. Since EXP has two-prover interactive proofs, a successful approach, based on interactive proofs, for showing that #P has polynomial-size circuits could lead to an approach for showing that EXP has polynomial-size circuits, and hence that $\mathsf{P} \neq \mathsf{NP}$.

In Chapter 2, we considered two approaches for showing that #P has polynomial-size circuits. As in the interactive proof for #3-SAT, these approaches were based on the idea of arithmetizing propositional formulas. The $2n$-variable polynomial approach dealt with arbitrary polynomials of degree at most two in each variable. This approach turned out to be too general, as not every polynomial of degree at most two in each variable corresponds to the arithmetization of some propositional formula. On the hand, the small low-degree descriptor approach turned out to be too restrictive, since there exist

propositional formulas whose arithmetizations do not have small low-degree descriptors. Approaches that are more restrictive than the $2n$-variable approach yet more general than the small low-degree descriptor approach need to be identified and investigated.

Are there instance checkers for languages complete for any NP-hard natural class that is not known to contain $\mathsf{P}^{\#\mathsf{P}}$?

Since instance checkers and interactive proofs are closely related, any progress made toward answering this question may lead to new approaches for investigating whether interaction can be converted into nonuniformity. Furthermore, as we saw in Chapter 4, the existence of particular instance checkers has been recently used to prove results about probabilistic computation.

In Chapter 3, we noted that the class of $(1 + \frac{1}{n^{O(1)}})$-approximate counting problems is an NP-hard class that is not known to contain $\mathsf{P}^{\#\mathsf{P}}$. We identified obstacles that need to be overcome in order to construct instance checkers for $(1 + \frac{1}{n^{O(1)}})$-approximate counting problems. These obstacles need to be further examined. Other NP-hard natural classes not known to contain $\#\mathsf{P}$, such as NP and Σ_2^P, should also be considered.

How constructive can proofs of hierarchy theorems be made?

It seems natural to prefer constructive proofs to those that are non-constructive. As we saw in Chapter 4, proofs of hierarchy theorems that use padding arguments are non-constructive. Can these proofs be made more constructive? The proof of the hierarchy theorem for $\mathsf{BPP}/1$ is also non-constructive. We discussed the problems that are encountered when trying to make the existing proof more constructive. What other techniques can be used to prove this theorem? Do such techniques result in more constructive proofs?

Bibliography

[Bar02] Boaz Barak. A probabilistic-time hierarchy theorem for "slightly non-uniform" algorithms. In *RANDOM '02: Proceedings of the 6th International Workshop on Randomization and Approximation Techniques*, pages 194–208. Springer-Verlag, 2002.

[BF91] László Babai and Lance Fortnow. Arithmetization: A new method in structural complexity theory. *Computational Complexity*, 1:41–66, 1991.

[BFL91] László Babai, Lance Fortnow, and Carsten Lund. Non-deterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1:3–40, 1991.

[BK95] Manuel Blum and Sampath Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, 1995.

[BM88] László Babai and Shlomo Moran. Arthur-merlin games: a randomized proof system, and a hierarchy of complexity classes. *Journal of Computer and System Sciences*, 36(2):254–276, 1988.

[FS04] Lance Fortnow and Rahul Santhanam. Hierarchy theorems for probabilistic polynomial time. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS'04)*, pages 316–324. IEEE Computer Society, 2004.

[FST05] Lance Fortnow, Rahul Santhanam, and Luca Trevisan. Promise hierarchies. In *STOC '05: Proceedings of the 27th ACM Symposium on Theory of Computing*. ACM Press, 2005. To appear.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[GSTS03] Dan Gutfreund, Ronen Shaltiel, and Amnon Ta-Shma. Uniform hardness versus randomness tradeoffs for Arthur-Merlin games. *Computational Complexity*, 12(3-4):85–130, 2003.

[HS65] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[HS66] F. C. Hennie and R. E. Stearns. Two-tape simulation of multitape Turing machines. *Journal of the ACM*, 13(4):533–546, 1966.

[KL82] R. Karp and R. Lipton. Turing machines that take advice. *L'enseignement mathématique*, 28:191–209, 1982.

[KS03] Valentine Kabanets and Amir Shpilka. Personal communication, May 2003.

[LFKN92] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM*, 39(4):859–868, 1992.

[Sha92] Adi Shamir. $\mathsf{IP} = \mathsf{PSPACE}$. *Journal of the ACM*, 39(4):869–877, 1992.

[Sto76] Larry Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1976.

[Sto83] Larry Stockmeyer. The complexity of approximate counting. In *STOC '83: Proceedings of the 15th ACM Symposium on Theory of Computing*, pages 118–126. ACM Press, 1983.

[Tod91] Seinosuke Toda. PP is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, 20(5):865–877, 1991.

[TV02] Luca Trevisan and Salil Vadhan. Pseudorandomness and average-case complexity via uniform reductions. In *CCC '02: Proceedings of the 17th IEEE Annual Conference on Computational Complexity*, pages 129–138. IEEE Computer Society, 2002.

[Val79] Leslie G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.