Counting in Structural Complexity Theory

Lane Adrian Hemachandra
Ph.D. Thesis

87-840
June 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

Counting in Structural Complexity Theory

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Lane Adrian Hemachandra

May 31, 1987

COUNTING IN STRUCTURAL COMPLEXITY THEORY

Lane Adrian Hemachandra, Ph.D.
Cornell University 1987

Structural complexity theory is the study of the form and meaning of computational complexity classes. Complexity classes—P, NP, ProbabilisticP, PSPACE, etc.—are formalizations of computational powers—deterministic, nondeterministic, probabilistic, etc. By examining the structure of and the relationships between these classes, we seek to understand the relative strengths of their underlying computational paradigms.

This thesis studies counting in structural complexity theory. We are interested in complexity classes defined by counting and in the use of counting to explore the structure of these and other classes.

We consider the structure of the strong exponential hierarchy, an exponential time analogue of the polynomial time hierarchy. A careful investigation of the census functions of nondeterministic computation trees shows that the strong exponential hierarchy collapses.

Next, we move from computing census functions of computation trees to computing census functions of sets. The ranking problem for a fixed set is to determine the position of elements within the set. We give strong structural evidence that ranking of any type—uniform, nonuniform, strong, weak, or enumeratively approximate—is computationally complex. Indeed, we can believe that most types of ranking are computationally hard with at least the certainty with which we believe that $P \neq NP$.

Returning to the combinatorics of computation trees and their accepting paths, we study robust machines. A robust machine is a nondeterministic Turing machine that maintains certain computational properties in every relativized world. We show that, due to the limited combinatorial control of NP machines, robust machines accept only simple languages. A robust machine will accept, for every

oracle A, a language that can be accepted by a polynomial time Turing machine with oracles for NP and A.

Finally, we turn to the count "one," and its effect on computation trees, accepting paths, and the structure of the satisfiability problem. We prove a UP (unique polynomial time) converse to the Borodin-Demers Theorem, and, under a complexity-theoretic assumption related to uniqueness, we show that there is an algorithm that quickly finds satisfying assignments for satisfiable formulas with few satisfying assignments.

Throughout this thesis, our goal is to use counting as a tool in understanding the structure of feasible computations.

Biographical Sketch

Lane Adrian Hemachandra was born in New York on February 21, 1960. When he was ten years old he proved that the product of the least common multiple and the greatest common divisor of two positive integers equals the product of the integers.[1] After this first theorem, he spent many years reading books and listening to classical music. He eventually resurfaced at Yale University and in 1981 received a Bachelor of Science degree, *summa cum laude*, in Mathematics & Physics and Computer Science. In 1982 he received a Master of Science degree in Computer Science from Stanford University. Since then, he has been proving theorems as a graduate student at Cornell University and appreciating the scenic beauty of Ithaca.

[1]The author has been informed that this result has been proven, independently and almost simultaneously, by Euclid (personal communication). A joint paper is planned [HE].

To my mother.

Acknowledgements

Professor Juris Hartmanis, my advisor, has been a model of insight, warmth, integrity, and grace. From him I've learned not only the mechanics of theorem statement and proof but also a philosophy of scientific inquiry. His intuitive ability to understand results within the context of the broad themes and needs of our field is striking and inspiring.

Professor John Hopcroft's confidence and encouragement have been invaluable. From him I learned the value of seeking alternative approaches to difficult problems.

I thank Professors Robert Bland and David Gries for serving on my committee with interest and concern. During my undergraduate years at Yale, Professors Dana Angluin, Dan Gusfield, David Lichtenstein, and Alan Perlis ignited my love of computer science and provided wise advice and guidance during my first encounters in theoretical computer science.

For wonderful suggestions, comments, and ideas that contributed to the research described in this thesis, I'm grateful to Eric Allender, Jin-yi Cai, Joan Feigenbaum, Judy Goldsmith, Juris Hartmanis, Paris Kanellakis, Dexter Kozen, Jeffrey Lagarias, Robert McCurley, Mark Novick, Steven Rudich, Vijay Vazirani, and Osamu Watanabe. For reading and improving drafts of this thesis, I thank Lee Barford, Robert Bland, David Gries, Juris Hartmanis, John Hopcroft, Mark Novick, William Pugh, Kenneth Regan, Geoffrey Smith, and Bradley Vander Zanden.

The Hertz Foundation generously supported my graduate education with a Fannie and John Hertz Fellowship. NSF grants DCR-8301766 and DCR-8520597 also supported my thesis research.

I'm overwhelmed and overjoyed by the number of friends who have made my graduate school years a time not only of proving theorems but also of enjoying concerts, parks, skiing, restaurants, and a generally eclectic lifestyle.

From quiet evenings of reading to moonlit snowshoe treks through the North Maine Woods, the joys I've shared with Anne Freedman form the core of my Cornell years.

I'll remember with warmth the years of haggling with Lee Barford over where to eat lunch. The first round bids were always McDonalds and L'Auberge du Cochon Rouge; we never went to either. Thanks to my friendship with Lee, I not only understand carburetors [Ste79] and television screens but also almost understand the "computer" in computer science.

My friends Joan Feigenbaum, Judy Goldsmith, and Yoram Moses gave undeserved support, unasked but not unappreciated. Knowing that I'll see them at conferences forever is a pleasure.

My fellow members of the Birthday Dinner Club—Lee Barford, Anne Freedman, Mark Novick, Bill Pugh, and Brad Vander Zanden—have been friends on more than just our birthdays. Happy October 14th, February 24th, April 4th, June 14th, and February 3rd!

Jin-yi Cai, Bruce Esrig, and Jennifer Widom meet life with degrees of (respectively but not exclusively) reflection, kindness, and élan refreshingly different from the norm. My friendships with them have broadened the way I view the world.

For a lifetime of understanding and advice, I'm grateful to my family—Aunt Hattie and Uncle John, Uncle Bill, Grandmother Ruth, and above all my mother.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

> The form is the meaning, and indeed the classic Greek mind, with an integrity of perception lost by later cultures which separated the two, firmly identified them.
>
> —Vincent Scully, *The Earth, the Temple, and the Gods* [Scu62].

To the computer scientist, structure is meaning. Seeking to understand nature's diverse problems with man's pathetic resources, we simplify our task by grouping similarly structured problems.

The resulting complexity classes, such as P, NP, and PSPACE, are simply families of problems that can be solved with a certain underlying computational power. The range of interesting computational powers is broad—deterministic, nondeterministic, probabilistic, unique, table lookup, etc.—and an equally rich spectrum of classes symbolize the powers—P, NP, PP, UP, P/poly, etc. These structurally motivated classes can themselves be studied in terms of their internal structure and behavior. For example, we might seek to understand the nature of the class NP by asking if NP contains any hard sparse sets [HIS85] or by asking if there is only one NP-complete set, which appears in many p-isomorphic guises [BH77].

This thesis studies the interaction between counting and structural complexity theory. We study the structure of classes defined by counting and how combinatorics and counting give us insight into the structure of complexity classes.

Chapter 2 briefly reviews the definitions of, meanings of, and previous research on the complexity classes we study. Summary sheets are collected in the appendices.

Chapter 3 studies the strong exponential hierarchy, an exponential time analogue of the polynomial time hierarchy. We study the structure of the computations at the second level (NP^{NE}) of the strong exponential hierarchy. This work is related to the quantitative relativizations of the polynomial hierarchy of Book, Long, and Selman [BLS84,Lon85]. By carefully building up a profile of the number of "yes" answers the base NP machine receives (the "census profile") we see that $P^{NE} = NP^{NE}$. It follows that the strong exponential hierarchy collapses to P^{NE}.

Chapter 3 shows how to compute census functions of *computation trees*. Chapter 4 discusses the complexity of computing census functions of *sets*. For a fixed set, the ranking problem is to determine the position (rank) of input strings within the set. As an example, the ranking problem is easy for the set of odd numbers. We can quickly tell that 1001 is the 501st odd number. Ranking was first studied by Blum, Goldberg, and Sipser [GS85], who showed that if a certain P set can be ranked in polynomial time, then the counting functions of NP machines (which are the functions counting the number of accepting paths of NP machines) can be computed in polynomial time.

Do all easy (P) sets have easily computable ranking functions? Do all NP sets have easily computable ranking functions? The latter might seem less likely. However, we show that all sets in P have easy ranking functions if and only if all sets in NP have easy ranking functions. Thus, though it is likely that the membership problems for P and NP are of different complexities, their ranking problems stand or fall together.

We also ask if there are small circuits for the ranking problem and if we can enumeratively approximate ranking functions. We can not, if standard assumptions about complexity classes hold.

A machine that maintains a property for *every* oracle[1] is said to have the property robustly [Sch84]. For example, if two nondeterministic Turing machines accept complementary languages for every oracle we say they are robustly com-

[1]Oracles and relativized computations are defined and discussed in Section 2.3.

plementary. Maintaining a property robustly strains a machine's combinatorial control; it is hard for a machine to be flexible enough to show a certain behavior with all oracles and also to accept complex languages.

Chapter 5 studies robustness and shows that machines with robustness properties accept only simple languages. For example, if two machines are robustly complementary then for every oracle A each of the machines accepts a language in $\mathrm{P}^{\mathrm{NP}\oplus A}$, where $\oplus$ represents disjoint union. In particular, if $\mathrm{P}=\mathrm{NP}$ then for each oracle A all robustly complementary machines accept languages in polynomial time relative to A.

Chapter 6 studies properties of a unique count—one. Borodin and Demers [BD76] show that if $\mathrm{P}\neq\mathrm{NP}\cap\mathrm{coNP}$ then there exists a polynomial time set S of satisfiable boolean formulas such that no polynomial time machine computes satisfying assignments for all formulas in S—that is, for no polynomial time computable function g do we have $(\forall F\in S)[g(F)$ is a satisfying assignment of $F]$. Intuitively, we have a set of formulas that we can easily recognize as satisfiable, but we cannot easily determine *why* they are satisfiable.

It is not known if the converse of the Borodin-Demers Theorem holds. The first part of Chapter 6 proves a "uniqueness" version of the theorem and its converse: $\mathrm{P}\neq\mathrm{UP}\cap\mathrm{coUP}$ if and only if there is a polynomial time set S so that each element of S is a formula with exactly one solution but no polynomial time machine computes satisfying assignments for all formulas in S—that is, for no polynomial time computable function g do we have $(\forall F\in S)[g(F)$ is the unique satisfying assignment of $F]$. Intuitively, we have a set of formulas that we can quickly recognize as each having exactly one solution, but we cannot get our hands on these solutions quickly.

The second part of Chapter 6 shows, if a complexity-theoretic assumption holds, that fast algorithms exist to find satisfying assignments for satisfiable boolean formulas that have few satisfying assignments. Our complexity-theoretic assumption is that we can distinguish zero from one. Call $(**)$ the assumption that there is a polynomial time Turing machine that

- given any unsatisfiable formula prints "unsatisfiable," and

- given any formula with exactly one solution prints "satisfiable."

Note that given a formula with many solutions we don't know what this machine will say; it may lie.

Valiant and Vazirani [VV85] show that $(**)$ implies $\mathrm{P}=\mathrm{UP}$ and NP equals R, random polynomial time. Assumption $(**)$ does not appear to imply $\mathrm{P}=\mathrm{NP}$. We show that $(**)$ implies that we can find a satisfying assignment to a satisfiable formula f in time $O(|f|^k \cdot ||f|| \cdot \binom{\#var(f)}{\min(\log ||f||, \#var(f)/2)})$, where $||f||$ is the number of satisfying assignments of f, $|f|$ is the size of f, and $\#var(f)$ is the number of distinct variables in f, and k is a constant. In particular, $(**)$ implies that for formulas with few solutions we have witness finding algorithms faster than the known (exponential time) algorithms. We interpret this as strong evidence that $(**)$ is false.

All these chapters strive to understand counting and its effects on the structure of the complexity classes that formalize our view of the world of feasible computations.

Chapter 2

A Primer of Complexity Classes

This chapter briefly reviews the definitions, meanings, and histories of the complexity classes this thesis studies. Detailed discussions of previous work related to this thesis are included in each chapter.

2.1 P: Determinism

$$\mathrm{P} = \{L \mid L \text{ is accepted by a polynomial time deterministic Turing machine}\}.$$

P, deterministic polynomial time, is the class that is widely thought to embody the power of reasonable computation (Figure 2.1). In the 1930s, Gödel, Church, Turing, and Post [God31,Chu36,Tur36,Chu41,Pos46,Dav58] asked what could be *effectively* solved by computing machines—that is, what problems are recursive? Starting in the 1960s, computer scientists have asked which problems can be *efficiently* solved by computers. The theory of P and NP, and indeed structural complexity theory itself, sprang from this desire to understand the limits of feasible computation.

The notion that polynomial time, $\bigcup_k \mathrm{TIME}[n^k]$, is the right class to represent feasible computation is due to Cobham and Edmonds [Cob64,Edm65]. Polynomials grow slowly and are closed under composition (thus allowing subroutine calls). These features support the claim that P is a reasonable resource bound. The view that P loosely characterizes "feasibility" is widely accepted.

P – Polynomial Time

Power

Feasible computation.

Definition

$\mathrm{P} = \bigcup_k \mathrm{TIME}[n^k]$.

Background

P was described as embodying the power of feasible computation by Cobham [Cob64] and Edmonds [Edm65]. The field of design and analysis of algorithms attempts to place as many problems as possible in P.

Complete Languages

P has well-known complete languages under $\leq_{many-one}^{logspace}$ reductions, e.g., the emptiness for context-free grammars [HU79].

Sample Problem

In a fixed, reasonable proof system, asking if x is a proof of T is a polynomial time question. In particular, in polynomial time we can check if assignment x satisfies boolean formula F.

Figure 2.1: P

One might argue that an algorithm that runs for $10^{10^{10}} n^{10^{100}}$ steps on inputs of size n is not practical. Problems are known that provably require high degree polynomial algorithms (artificial problems [HS65][HU79, Theorem 12.9], cat-and-mouse and pebbling problems [KAI79,AIK81]), and natural problems are known that may require high degree polynomial algorithms (permutation group membership from generators [Hof82,FHL80], robotics configuration space problems [SS83]).[1]

Nonetheless, there is a widely held feeling that fundamental natural problems belonging to P will have polynomial time algorithms of low degree. The field of design and analysis of algorithms attempts to bring problems into P, and then show that they have algorithms of low time complexity [AHU74,Tar83].

2.2 NP: Nondeterminism

$$\mathrm{NP} = \bigcup_k \mathrm{NTIME}[n^k].$$

P contains the problems we can solve. NP symbolizes the problems man needs to solve to efficiently structure and optimize his world. The P=NP question asks if the computers built by man's ingenuity have the power to solve the problems formed by nature's complexity.

NP is the class of languages accepted by nondeterministic polynomial time bounded Turing machines [HU79]. Intuitively, a nondeterministic machine is one that is allowed to make guesses during its computation, and always guesses correctly.

In the early 1970s, the work of Cook and Karp [Coo71,Kar72] showed that NP had natural complete, or "hardest," languages—languages to which every other NP problem can be polynomial time many-one reduced. These problems stand or fall together: if one NP-complete problem is in P then all NP-complete problems are in P. During the last sixteen years, hundreds of problems from all areas of

[1] Many problems are known to have high lower bounds. Meyer and Stockmeyer [MS72] and Fischer and Rabin [FR74] show, respectively, problems that require exponential space and double exponential nondeterministic time. Our question is, are there natural, fundamental *polynomial time* problems that have high degree polynomial lower bounds?

NP – Nondeterministic Polynomial Time

Power Guessing. Nondeterminism.

Definition $\mathrm{NP} = \bigcup_k \mathrm{NTIME}[n^k]$.

Background

In the early 1970s, Cook [Coo71], Karp [Kar72], and Levin [Lev73] initiated the study of NP and its complete problems. Many NP-complete problems are now known, and the study of NP's structure is the unifying theme of structural complexity theory.

Complete Problems

NP has hundreds of $\leq_m^p$ (polynomial time many-one) complete problems [GJ79]. The most studied NP-complete problem is satisfiability. SAT=$\{f \mid$ boolean formula f is satisfiable$\}$ was shown to be Turing-complete for NP by Cook. Karp showed that SAT and many other problems were $\leq_m^p$-complete for NP.

Theorems

- If $(\exists$ sparse $S)[\mathrm{NP} \subseteq \mathrm{P}^S]$ then the polynomial hierarchy, PH, equals $\mathrm{NP}^{\mathrm{NP}}$. [KL80]
- NP has sparse complete sets if and only if $\mathrm{P} = \mathrm{NP}$. [Mah80]
- $\mathrm{NP} - \mathrm{P}$ contains sparse sets if and only if $\mathrm{E} \neq \mathrm{NE}$. [HIS85]
- All paddable sets are p-isomorphic to SAT. [BH77,MY85]
- If $\mathrm{P} = \mathrm{NP}$ and S is sparse then $[\mathrm{P}^S = \mathrm{NP}^S \Leftrightarrow (\exists k)[S \subseteq K^S[k \log n, n^k]]]$, where $K[]$ represents time bounded Kolmogorov complexity. [HH86b]
- $\mathrm{P} \neq \mathrm{NP} \Rightarrow \mathrm{NP} - \mathrm{P}$ contains sets that are not NP-complete. [Lad75]
- $(\exists A)[\mathrm{P}^A = \mathrm{NP}^A]$. $(\exists B)[\mathrm{P}^B \neq \mathrm{NP}^B]$. [BGS75]

Figure 2.2: NP

mathematics, computer science, and operations research have been shown NP-complete. If P=NP then these and many crucial optimization problems can be solved in polynomial time.

However, the implications of P = NP are even more profound. An NP machine can answer the question, in a fixed formal system, "Does this theorem have a proof (of reasonable size)?" Thus NP embodies the power of guessing, or creating, mathematical proofs. P embodies the mechanical process of verifying if a proof is correct. Asking if P $\neq$ NP is another way of asking if the creative process in mathematics rises above the complexity of mere mechanical verification. Since men are likely to create mathematical proof structures only of small size, asking if P = NP is one way of asking if machines can usurp man's role in mathematical discovery.

NP is the most extensively studied computational complexity class, and many insights into NP's structure have been found during the past decade (Figure 2.2). Nonetheless, our understanding of NP is fragmented, incomplete, and unsatisfying.

2.3 Oracles and Relativized Worlds

Oracles are defined and discussed in the literature [HU79,BGS75,Tor86]. We may think of an oracle B as a *unit cost subroutine* for the set B. For example, P^B (NP^B) is the class of languages computable by deterministic (nondeterministic) polynomial time Turing machines given unit cost subroutines (i.e., subroutines that return in one time unit) that test membership in B. We may think of such a subroutine as changing the ground rules of computation that the machines operate under.

We can also define what it means to relativize a complexity class not with a single set but with another complexity class:

$$\mathcal{C}^{\mathcal{D}} = \bigcup_{A \in \mathcal{D}} \mathcal{C}^A$$

For example, $\mathrm{NP}^{\mathrm{NP}} = \bigcup_{A \in \mathrm{NP}} \mathrm{NP}^A = \mathrm{NP}^{\mathrm{SAT}}$. We may think of $\mathcal{C}^{\mathcal{D}}$ as the class of languages recognized by $\mathcal{C}$ machines given free access to the power of some member of $\mathcal{D}$.

Oracles are a useful tool in certifying structural possibilities for complexity classes. If we show that some complexity result T holds in a relativized world (that is, with some oracle B), we know that relativizable proof techniques cannot disprove T. This is because a relativizable disproof of T would disprove T in *all* relativized worlds, but we know that T is true in the world relativized by B.

Many crucial results in structural complexity can be relativized in conflicting ways. For example, there are oracles A and B so that $\mathrm{P}^A = \mathrm{NP}^A$ yet $\mathrm{P}^B \neq \mathrm{NP}^B$ [BGS75]. Since most known mathematical proof techniques seem to relativize (for a discussion of this and of possible exceptions see (Hartmanis [Har85])), new techniques or novel uses of old techniques will be needed to resolve such central questions as $\mathrm{P} = \mathrm{NP}$.

Oracles exist to certify unlikely situations—e.g., there is an oracle A for which $\mathrm{P}^A = \mathrm{NP}^A = \mathrm{PSPACE}^A$. We should not think of oracles as telling us what is the case in the world of computation. We should think of oracles as suggesting the limitations of our mastery of mathematical proof techniques.

2.4 The Polynomial Hierarchy and Polynomial Space: The Power of Quantifiers

2.4.1 The Polynomial Hierarchy

The polynomial hierarchy was defined by Stockmeyer [Sto77] as a time bounded analogue of the Kleene (arithmetic) hierarchy of recursive function theory [Rog67]. The definitions of the polynomial hierarchy appear in Figure 2.3. In particular, $\Sigma_0^p = \mathrm{P}$, $\Sigma_1^p = \mathrm{NP}$, $\Pi_1^p = \mathrm{coNP}$, $\Delta_2^p = \mathrm{P}^{\mathrm{NP}}$, and $\Sigma_2^p = \mathrm{NP}^{\mathrm{NP}}$. Figure 2.4 diagrams the structure of the polynomial hierarchy.

The levels of the polynomial hierarchy have natural descriptions in terms both of Turing machines and logical formulas. Just as the Kleene hierarchy's levels are characterized by quantifier alternation, so also are the levels of the polynomial hierarchy characterized by alternating polynomial bounded quantifiers [Sto77]. For example:

$$
\begin{aligned}
\mathrm{NP} &= \{L \mid (\exists k)(\exists \textit{ polynomial predicate } P)[x \in L \Leftrightarrow (\exists y)[|y| \leq |x|^k \wedge P(x,y)]]\} \\
\Pi_2^p &= \{L \mid (\exists k)(\exists \textit{ polynomial predicate } P) \\
&\qquad [x \in L \Leftrightarrow (\exists y)(\forall z)[|y| \leq |x|^k \wedge [|z| \leq |x|^k \Rightarrow P(x,y,z)]]]\}.
\end{aligned}
$$

This characterization by alternating quantifiers is handy. When asked the complexity of MINIMAL-FORMULAS= $\{F \mid F$ is a boolean formula and no equivalent boolean formula is shorter than $F\}$, we can reflect for a moment on the underlying quantifier structure and quickly note that MINIMAL-FORMULAS$\in \Pi_2^p$. That is, MINIMAL-FORMULAS is the set of all F such that *for every* shorter formula F' *there exists* a variable assignment on which F and F' differ.

The work of Chandra, Kozen, and Stockmeyer [CKS81] develops machines that accept the languages at each level of the polynomial hierarchy. Known as alternating Turing machines, the action of these machines alternates between existential and universal blocks, and mirrors the underlying quantifier structure of the classes.

We say that the polynomial hierarchy *collapses* if, for some k, $\Sigma_k^p = \Pi_k^p$ (thus $\Sigma_k^p = \mathrm{PH}$). A crucial open question is, does the polynomial hierarchy collapse? That is, is some fixed number of quantifiers powerful enough to simulate all fixed arrangements of quantifiers? Oracles are known for which the hierarchy collapses [BGS75] and for which the hierarchy does not collapse [Yao85]. An exponential analogue of the polynomial hierarchy collapses (Chapter 3).

2.4.2 Polynomial Space

PSPACE is the class of languages accepted by polynomial space bounded Turing machines. PSPACE embodies the power of polynomial bounded quantifiers. A quantified boolean formula is an expression of the form

$$(\exists x_1)(\forall x_2)(\exists x_3)\cdots[f(x_1, x_2, x_3, \ldots)],$$

where f is a quantifier-free boolean formula. QBF, the set of true quantified boolean formulas, is a well-known PSPACE-complete problem, and shows how PSPACE embodies the power of alternating quantifiers [Sto77].

PH, PSPACE, P, NP, coNP, $\mathbf{P^{NP}}$, $\mathbf{NP^{NP}}$, ... – The Polynomial Hierarchy and Polynomial Space

Power Alternating polynomial bounded existential and universal quantifiers.

Definition

$$\begin{aligned}
\Sigma_0^p &= \Pi_0^p = \mathrm{P} \\
\Delta_{i+1}^p &= \mathrm{P}^{\Sigma_i^p} \qquad i \geq 0 \\
\Sigma_{i+1}^p &= \mathrm{NP}^{\Sigma_i^p} \qquad i \geq 0 \\
\Pi_{i+1}^p &= \mathrm{co}\Sigma_{i+1}^p = \{L \mid \overline{L} \in \Sigma_{i+1}^p\} \qquad i \geq 0 \\
\mathrm{PH} &= \bigcup_i \Sigma_i^p \\
\mathrm{PSPACE} &= \bigcup_k \mathrm{SPACE}[n^k].
\end{aligned}$$

Background The polynomial hierarchy was defined in (Stockmeyer [Sto77]).

Complete Languages Canonical complete languages exist for each level of the hierarchy [Wra77] and for PSPACE [Sto77].

Theorems

- $(\exists A)[\mathrm{P}^A = \mathrm{PH}^A]$. [BGS75]
- $(\exists A)[\mathrm{P}^A \neq \mathrm{NP}^A \neq \mathrm{NP}^{\mathrm{NP}^A} \neq \cdots \neq \mathrm{PSPACE}^A]$. [Yao85]
- $\mathrm{PH} = \mathrm{PSPACE} \Leftrightarrow (\forall \text{ sparse } S)[\mathrm{PH}^S = \mathrm{PSPACE}^S]$. [BBL*84]
- $\Sigma_k^p = \Pi_k^p \Rightarrow \Sigma_k^p = \mathrm{PH}$ (Downward Separation). [Sto77]
- $\mathrm{Prob}_A(\mathrm{PH}^A \neq \mathrm{PSPACE}^A) = 1$. [Cai86]
- PSPACE = NPSPACE = Probabilistic-PSPACE. [Sav70,Sim77]

Open Problems

- Does the polynomial hierarchy collapse?
- $\mathrm{Prob}_A(\mathrm{P}^A \neq \mathrm{NP}^A \neq \mathrm{NP}^{\mathrm{NP}^A} \neq \cdots) = 1$?
- For which j can we construct oracles so $(\Sigma_j^p)^A \neq (\Sigma_{j+1}^p)^A = \mathrm{PH}^A$?
- $(\exists k, A)[(\Sigma_k^p)^A = (\Pi_k^p)^A \neq \mathrm{PSPACE}^A]$?

Figure 2.3: The Polynomial Hierarchy and PSPACE

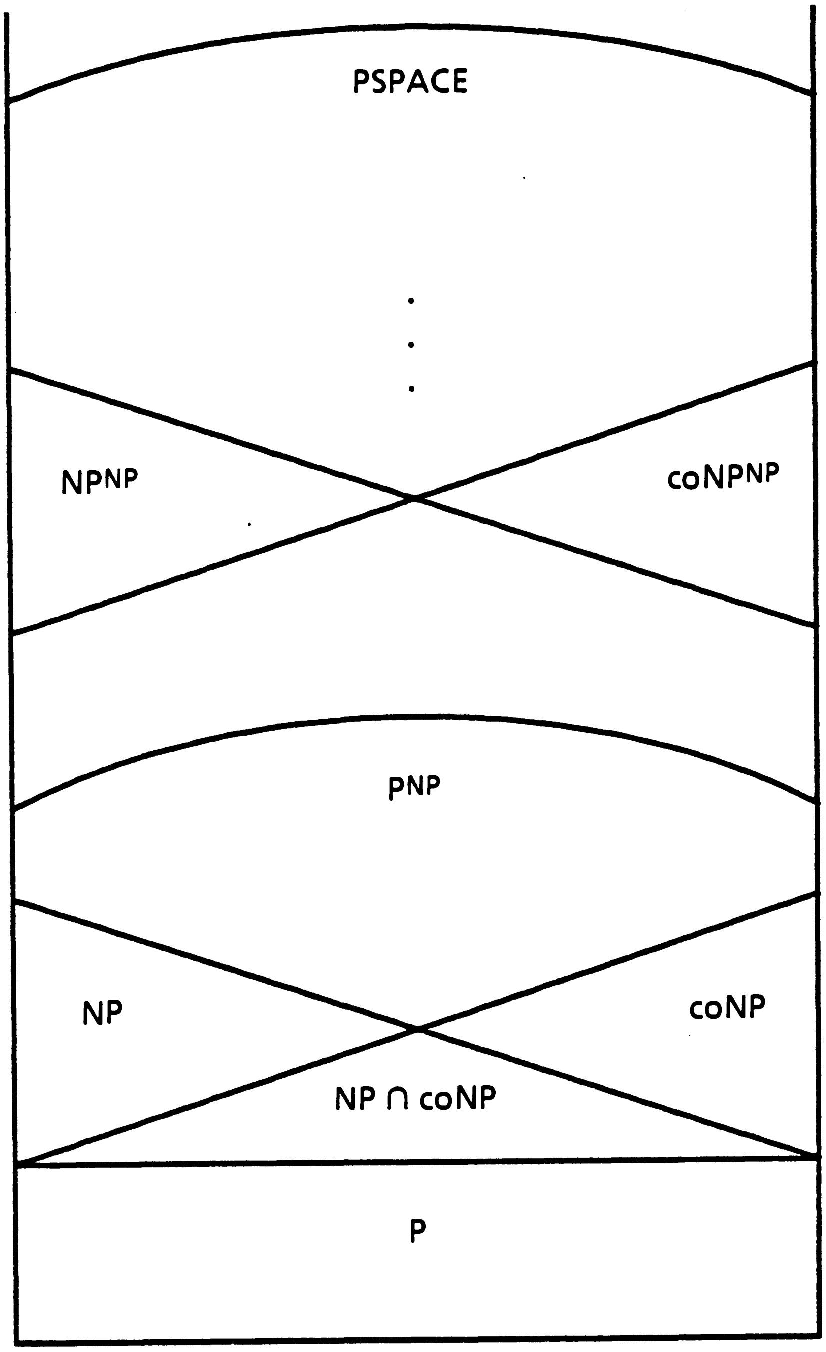


Figure 2.4: The Structure of The Polynomial Hierarchy

There are many PSPACE-complete problems. Adversary (game) problems are often PSPACE-complete. The generalized versions of checkers [FGJ*78] and go [LS78] are PSPACE-complete. In a fixed formal system, whether a theorem has a polynomial proof presentation can be determined in PSPACE [HY84].

2.5 E, NE, and the Strong Exponential Hierarchy

E and NE are exponential time analogues of P and NP. The structure of these exponential time classes is linked to the structure of polynomial time classes. In particular, the complexity of tally[2] and sparse sets within NP is tied to the structure of E and NE (Book, Cai, Hartmanis, Hemachandra, Hunt, Immerman, Sewelson, and Yesha [Boo74,HH74,HY84,HIS85,CH86a,CHHS86b]).

The strong exponential hierarchy (Figures 2.5 and 2.6) is a natural exponential time analogue of the polynomial time hierarchy. Its high levels at first seem to have great computational power. Thus it is somewhat of a surprise that the hierarchy collapses to its P^{NE} level (Chapter 3).

2.6 P/Poly: Small Circuits

$$\mathrm{P/poly} = \{L \mid (\exists \text{ sparse } S)[L \in \mathrm{P}^S]\}$$

L is in P/poly (Figure 2.7) if and only if L has small circuits, i.e., there is a family of representations of boolean circuits [Sav72,Sch86] $c_1, c_2, \ldots$ and an integer k so that:

- $|c_i| \leq i^k + k$, and
- $x \in L \Leftrightarrow c_{|x|}$ accepts x [KL80].

Equivalently, a language L is in P/poly if and only if there is a sparse[3] set S so

[2] T is a tally set if $T \subseteq 1^* = \{\epsilon, 1, 11, 111, \ldots\}$.

[3] A set S is sparse if there are at most polynomially many elements of length $\leq n$ in S, i.e., $(\exists k)(\forall n \geq 1)[|\{x \mid x \in S \wedge |x| \leq n\}| \leq n^k]$.

E, NE, P^{NE}, $\mathrm{NP}^{\mathrm{NE}}$, ... – The Strong Exponential Hierarchy

Power Exponential computation hierarchy.

Definitions

$$
\begin{aligned}
\mathrm{E} &= \Sigma_0^{\mathrm{SEH}} = \bigcup_c \mathrm{TIME}[2^{cn}] \\
\mathrm{NE} &= \Sigma_1^{\mathrm{SEH}} = \bigcup_c \mathrm{NTIME}[2^{cn}] \\
\Delta_i^{\mathrm{SEH}} &= \mathrm{P}^{\Sigma_{i-1}^{\mathrm{SEH}}} \qquad i \geq 2 \\
\Sigma_i^{\mathrm{SEH}} &= \mathrm{NP}^{\Sigma_{i-1}^{\mathrm{SEH}}} \qquad i \geq 2 \\
\mathrm{SEH} &= \mathrm{E} \cup \mathrm{NE} \cup \mathrm{NP}^{\mathrm{NE}} \cup \mathrm{NP}^{\mathrm{NP}^{\mathrm{NE}}} \cup \cdots \\
\mathrm{EXPSPACE} &= \bigcup_k \mathrm{SPACE}[2^{n^k}]
\end{aligned}
$$

Background The complexity of sparse sets in the polynomial hierarchy is closely related to the structure of exponential time classes [Boo74,HH74, HY84,HIS85,CH86a,CHHS86b].

Complete Languages All these classes have straightforward canonical complete languages that capture the actions of generic machines (see the techniques of [Har78]).

Theorems

- $\mathrm{E} = \mathrm{NE}$ if and only if there are no tally sets in $\mathrm{NP} - \mathrm{P}$. [Boo74,HH74]
- $\mathrm{E} = \mathrm{NE}$ if and only if there are no sparse sets in $\mathrm{NP} - \mathrm{P}$. [HIS85]
- $\mathrm{E} = \mathrm{NE}$ if and only if all capturable sets in the boolean hierarchy are in P. [CH86a]
- $\mathrm{NE} = \mathrm{coNE}$ if and only if every sparse set in NP is NP-printable. [HY84]
- $\mathrm{P}^{\mathrm{NE}} = \mathrm{E} \cup \mathrm{NE} \cup \mathrm{NP}^{\mathrm{NE}} \cup \mathrm{NP}^{\mathrm{NP}^{\mathrm{NE}}} \cup \cdots$. (Chapter 3)
- $\mathrm{E} = \mathrm{NE} \Rightarrow \mathrm{EXP} = \mathrm{SEH}$ (Downward Separation). (Chapter 3)

Figure 2.5: E, NE, and the Strong Exponential Hierarchy

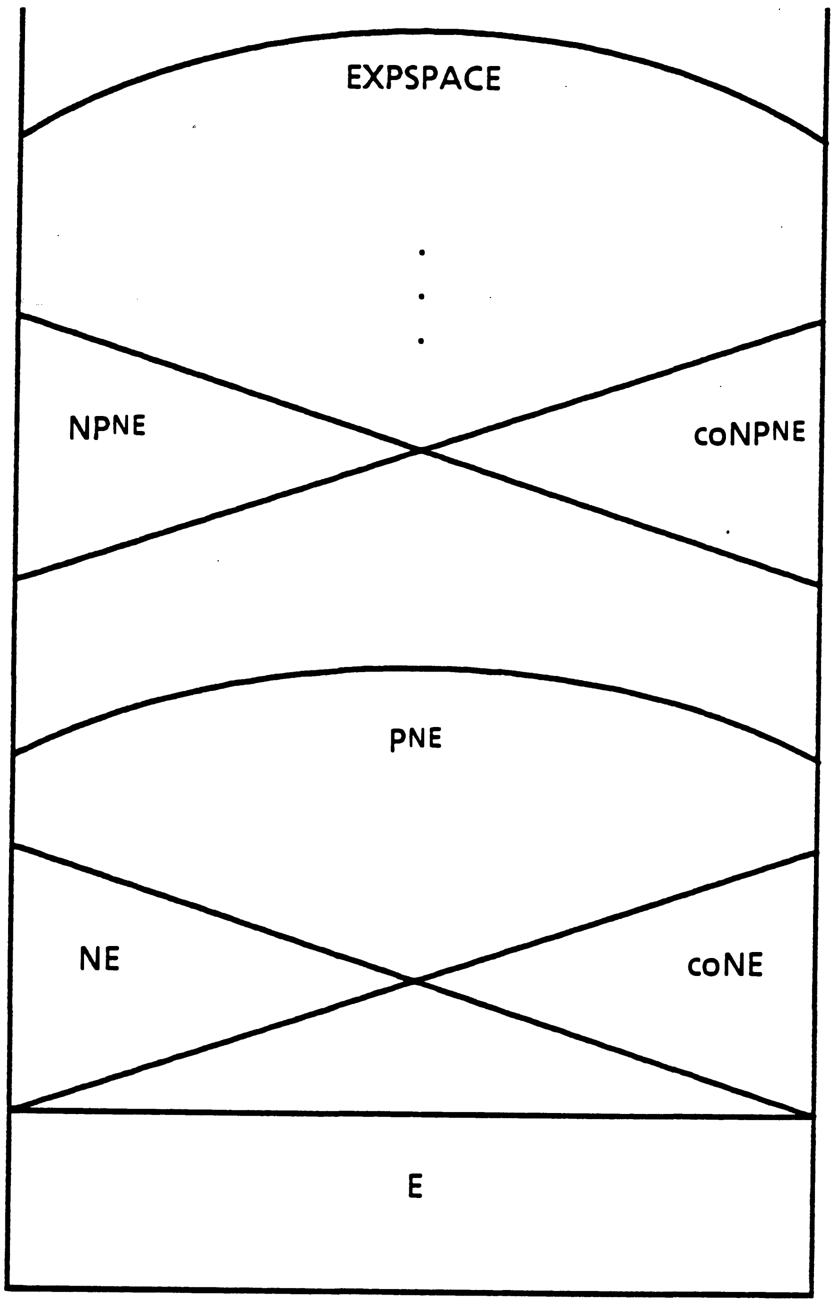


Figure 2.6: The Structure of the Strong Exponential Hierarchy

P/poly – Nonuniform Polynomial Time

Power

Small circuits. Table lookup.

Definition

$\mathrm{P/poly} = \{L \mid (\exists \text{ sparse } S)[L \in \mathrm{P}^S]\}$

Theorems

- $\mathrm{NP} \subseteq \mathrm{P/poly} \Rightarrow \mathrm{PH} = \mathrm{NP}^{\mathrm{NP}}$. [KL80]
- $(\exists S)[\mathrm{NP} \subseteq \mathrm{P}^S \wedge S \text{ sparse} \wedge S \in \mathrm{NP}] \Rightarrow \mathrm{PH} = \mathrm{P}^{\mathrm{NP[log]}}$. [Kad87]
- There is a relativized world A and a sparse set S so $\mathrm{PSPACE}^A \subseteq \mathrm{P}^{A \oplus S}$ yet the boolean hierarchy relative to A is infinite. [CH86a]
- If P has small ranking circuits then $\mathrm{P}^{\#\mathrm{P}} \subseteq \mathrm{PH} = \mathrm{NP}^{\mathrm{NP}}$. (Chapter 4)
- If P has small ranking circuits then P has small ranking circuits that can be printed by a Δ_3^p machine. (Chapter 4)

Figure 2.7: P/Poly

that $L \in P^S$ (due to Meyer, see [BH77, p. 307] and [KL80]).

Intuitively, sets in P/poly are "close" to being in polynomial time. With a small amount of advice (e.g., the circuit description), a polynomial machine can recognize these sets. However, the advice may be terribly hard to compute; thus it is not surprising that P/poly contains sets arbitrarily high in the Kleene hierarchy.

Some sets that are not known to be in P are known to have small circuits. For example, the set of primes is not known to be in P, but has small circuits [Rab76,Adl78,APL80,GK86]. More generally, any set in the probabilistic class R has small circuits.

Karp and Lipton show it unlikely that all NP sets have small circuits: if NP has small circuits (i.e., if $NP \subseteq P^S$ for some sparse set S) then the polynomial hierarchy collapses to its second level. In the wake of their result, a flurry of related research has extended our knowledge of the implications of "$NP \subseteq P^S$, S sparse," and of "$NP \subseteq P^S$, S sparse, $S \in NP$" (Cai, Hemachandra, Mahaney, Immerman, Kadin, and Long [MI82,CH86a,Lon82,Mah82,Kad87]).

2.7 UP and FewNP: Uniqueness

$$\begin{aligned} \mathrm{UP} &= \{L \mid \text{ there is a nondeterministic polynomial time Turing machine } N \\ &\qquad \text{so } L = L(N)\text{, and for all } x\text{, the computation of } N(x) \text{ has at most} \\ &\qquad \text{one accepting path}\}. \\ \mathrm{US} &= \{L \mid \text{ there is a polynomial predicate } P \text{ and integer } k \text{ so for all } x, \\ &\qquad x \in L \Leftrightarrow |\{y \mid |y| \leq |x|^k \wedge P(x,y)\}| = 1\}. \end{aligned}$$

The classes UP and US (Figures 2.8, 2.9, and 2.10) capture the power of uniqueness. Given a boolean formula f a typical US question would be, "Does f have exactly one solution?"

UP has a related but subtly different nature. UP is the class of problems that have (on some NP machine) unique witnesses. That is, if there is an NP machine M accepting L and for every input x the computation $M(x)$ has at most one accepting path, then we say $L \in \mathrm{UP}$. We call NP machines that accept on at most

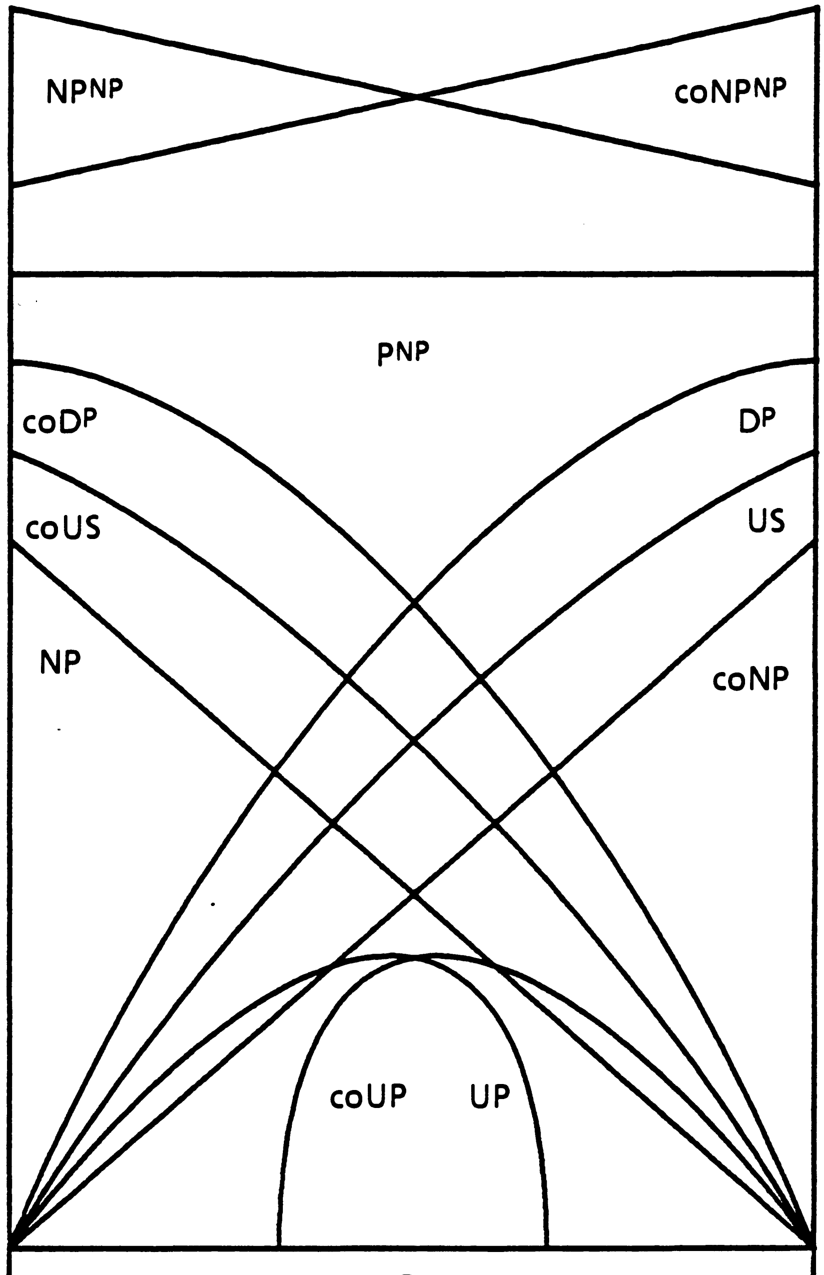


Figure 2.8: The Structure of UP and US within the Polynomial Hierarchy

UP, US – Unique Polynomial Time

Power

Categorical acceptance. Uniqueness.

Definition

$$\begin{aligned}\mathrm{UP} &= \{L \mid \text{there is a nondeterministic polynomial time Turing machine } N \\ &\quad \text{so } L = L(N)\text{, and for all } x\text{, the computation of } N(x) \text{ has at most} \\ &\quad \text{one accepting path}\}.\\ \mathrm{US} &= \{L \mid \text{there is a polynomial predicate } P \text{ and integer } k \text{ so for all } x,\\ &\quad x \in L \Leftrightarrow |\{y \mid |y| \leq |x|^k \wedge P(x,y)\}| = 1\}.\end{aligned}$$

Background

UP is defined in (Valiant [Val76]). US is studied in (Blass and Gurevich [BG82]). UP is related to cryptography [GS84] and to central conjectures in structural complexity theory [JY85,HH87].

Complete Problems

USAT$=\{f \mid f$ has exactly one satisfying assignment$\}$ is complete for US.

UP may not have complete languages. There are relativized worlds where it does not have complete languages and relativized worlds where $\mathrm{P}^A \neq \mathrm{UP}^A \neq \mathrm{NP}^A$ yet UP^A does have complete languages [HH86a].

Figure 2.9: UP—Part I

UP, US – Unique Polynomial Time

Theorems

- $P \neq UP \Leftrightarrow$ one-way functions exist. [GS84]
- $P \neq UP \cap coUP \Leftrightarrow$ one-way functions whose range is in P exist. [GS84]
- $UP \subseteq NP \cap US$
- If UP has complete languages then it has a complete language of the form $L = SAT \cap A$, $A \in P$. [HH86a]
- There is an oracle so $USAT^A$ is not $(D^P)^A$-complete. [BG82]
- $(\exists A)[P^A = UP^A \neq NP^A]$. $(\exists B)[P^B \neq UP^B = NP^B]$. [Rac82]
- $(\exists A)[UP^A$ has no complete languages$]$. [HH86a]
- $(\exists A)[P^A \neq UP^A \neq NP^A$ and UP^A has complete languages $]$. [HH86a]
- $(\forall A)[N_i^A$ is categorical $] \Rightarrow (\forall A)[L(N_i^A) \in P^{NP \oplus A}]$. (Chapter 5) and [HH87]
- There is a reasonable (i.e., $P^A \neq NP^A$) oracle A for which $P^A = UP^A$ (that is, there are no one-way functions) yet there are sets that are $\leq_m^{p,A}$-complete for NP^A and are non-p^A-isomorphic. [HH87]
- $P \neq UP \cap coUP$ if and only if there is a set S so (1) $S \in P$ and $S \subseteq SAT$, and (2) $f \in S \Rightarrow f$ has exactly one solution, and (3) no P machine can find solutions for all formulas in S—that is,

$$g(f) = \begin{cases} 0 & f \notin S \\ \text{the unique satisfying assignment of } f & f \in S \end{cases}$$

is not a polynomial time computable function. (Chapter 6)

Open Problems $\text{Prob}_A(P^A = UP^A) = 1$? Does UP have complete languages [HH86a]? Do one-way functions exist [GS84]?

Figure 2.10: UP—Part II

one path for all inputs *categorical* machines. Valiant started the study of UP and categorical machines in [Val76].

Recently, UP has come to play a crucial role in both cryptography and structural complexity theory. In cryptography theory, Grollmann and Selman [GS84] prove that one-way functions[4] exist if and only if $P \neq UP$, and one-way functions whose range[5] is in P exist if and only if $P \neq UP \cap coUP$. Thus we suspect that $P \neq UP$ because we suspect that one-way functions exist.

A central question in structural complexity theory, first asked by Berman and Hartmanis [BH77], is "how many NP-complete problems are there?" Berman and Hartmanis conjectured that there is only one NP-complete problem, which appears in many guises. That is, they conjectured that all NP-complete sets are polynomial time isomorphic (p-isomorphic). Indeed, they showed that all then-known and all paddable NP-complete sets are p-isomorphic ([BH77] and Mahaney and Young [MY85]). Note that the conjectured p-isomorphism of NP-complete sets implies $P \neq NP$.

Joseph and Young found NP-complete "k-creative" sets that are *not* obviously p-isomorphic to SAT. However, if no one-way functions exist then these sets are isomorphic to SAT. This led to the following conjecture [JY85,KMR86]. Since one-way functions exist if and only if $P \neq UP$, this conjecture links $P = UP$ to the structure of NP.

One-way Conjecture One-way functions exist if and only if non-p-isomorphic NP-complete sets exist.

This coupling between UP and NP has been weakened. Hartmanis and Hemachandra [HH87] show that there is a relativized world in which the One-way Conjecture fails. That is, there is a world in which there are no one-way functions yet there are non-p-isomorphic NP-complete sets. Their oracle consists of a powerful collapsing component (PSPACE) unioned with an extraordinarily sparse diagonalizing component.

[4] A function f is *honest* if $(\exists k)(\forall x)[|f(x)|^k + k \geq |x|]$. A *one-way function* is a total, single-valued, one-to-one, honest, polynomial time computable function f such that f^{-1} (which will be a partial function if range$(f) \neq \Sigma^*$) is not computable in polynomial time [GS84].

[5] *Range*$(f) = \bigcup_{i \in \Sigma^*} f(i)$.

Theorem 2.1 There is a reasonable (i.e., $P^A \neq NP^A$) oracle A for which $P^A = UP^A$ (that is, there are no one-way functions) yet there are sets that are $\leq_m^{p,A}$-complete for NP^A and are non-p^A-isomorphic.

This does not imply that the one-way conjecture is false, though it does open that possibility. This theorem, however, suggests that the conjecture is unlikely to be proved by standard techniques.

The class FewNP (Allender [All86]) is an analogue of UP that restricts machines not to one accepting path but to at most polynomially many accepting paths. Clearly, $P \subseteq UP \subseteq FewNP \subseteq NP$, and Allender [All86] shows that $P = FewNP$ if and only if all sparse sets in P are P-printable.[6]

Definition 2.2 $L \in FewNP$ if there is a nondeterministic polynomial time Turing machine N so that N accepts language L and for some polynomial q, $(\forall x)[N(x)$ has at most $q(|x|)$ accepting paths].

2.8 #P: Counting

$$\#P = \{f \mid (\exists \text{ nondeterministic polynomial time Turing machine } N)(\forall x) [f(x) = \text{ number of accepting paths of } N(x)]\}.$$

#P is the class of *functions* that count the accepting paths of nondeterministic polynomial time Turing machines (Figure 2.11). For example, the function that maps any boolean formula to its number of satisfying assignments is a #P function. To create a language, as opposed to a function class, we usually discuss $P^{\#P}$ or $P^{\#P[1]}$ ($P^{\#P}$ with at most one oracle call allowed).

Little is known about the complexity of $P^{\#P}$. Since the number of solutions of a boolean formula tells if it is satisfiable, we have $P^{NP[1]} \subseteq P^{\#P[1]}$ and $P^{NP} \subseteq P^{\#P}$. Also, PSPACE can count accepting paths by brute force so $P^{\#P} \subseteq PSPACE$ (Figure 2.12). Angluin displays a relativized world in which $P^{\#P} \supsetneq \Sigma_2^p \cup \Pi_2^p$ [Ang80].

#P is closely related to PP, probabilistic polynomial time: $P^{PP} = P^{\#P}$ [Gil77, Sim75].

[6]A set S is *P-printable* if there is a polynomial time Turing machine M such that for each n, $M(1^n)$ prints all elements of S of length at most n [HY84].

#P – Sharp P (Counting Functions)

Power

Counting.

Definition

$$\#\mathrm{P} = \{f \mid (\exists \text{ nondeterministic polynomial time Turing machine } N)(\forall x) \\ [f(x) = \text{number of accepting paths of } N(x)]\}.$$

Background

#P was first studied by Valiant [Val79a], who showed that counting versions not only of NP-complete problems but also of P problems can be #P-complete.

Complete Problems

#SAT is a representative #P function: $\mathrm{P}^{\#\mathrm{P}[1]} = \mathrm{P}^{\#\mathrm{SAT}[1]}$.

Theorems

- $(\exists A)[\mathrm{P}^{\#\mathrm{P}^A} - ((\Sigma_2^p)^A \cup (\Pi_2^p)^A) \neq \emptyset]$. [Ang80]
- If #SAT has a $\mathrm{O}(n^{1-\epsilon})$ enumerative approximator then $\mathrm{P} = \mathrm{P}^{\#\mathrm{P}}$. [CH86a]
- $\mathrm{P} = \mathrm{P}^{\#\mathrm{P}}$ if and only if every P set has a polynomial time computable ranking function. [GS85], [Rud87], and (Chapter 4)

Open Problems

- $\mathrm{P}^{\#\mathrm{P}} \subseteq \mathrm{PH}$?
- $\mathrm{PH} \subseteq \mathrm{P}^{\#\mathrm{P}}$?

Figure 2.11: #P

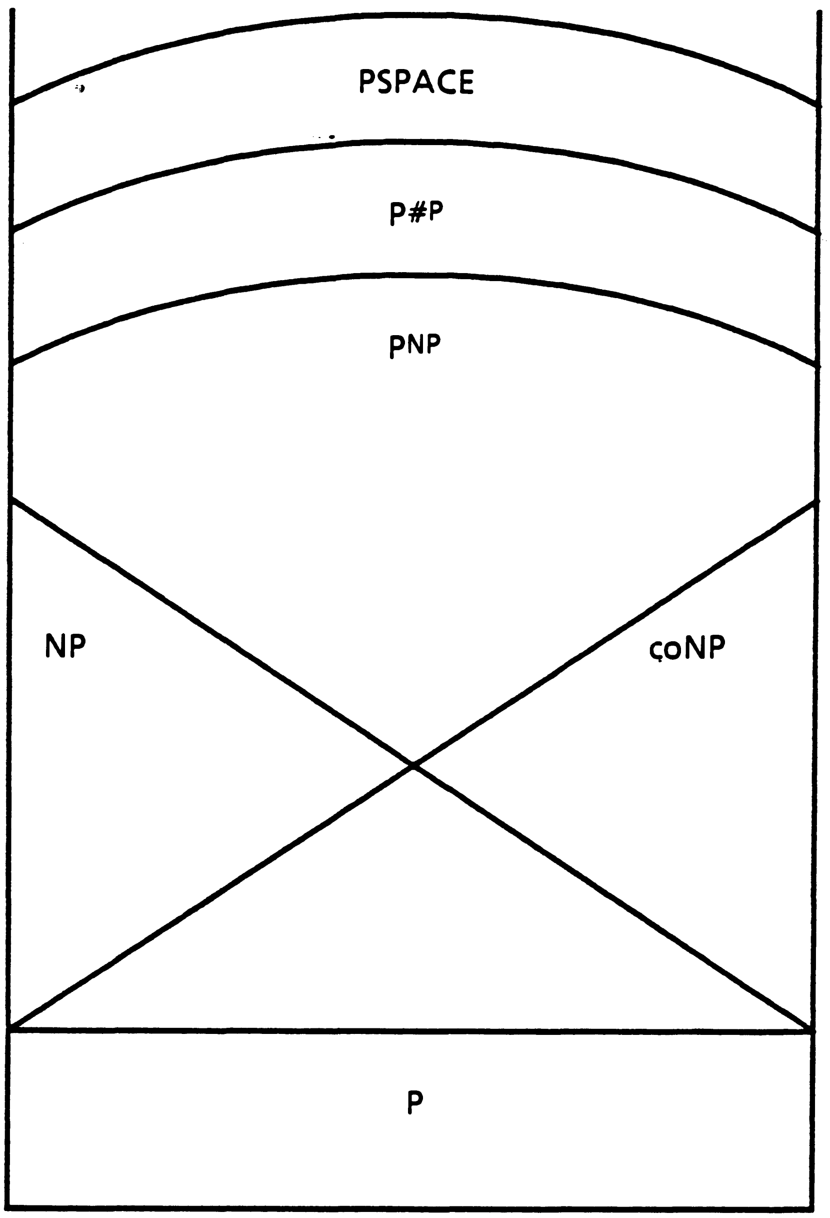


Figure 2.12: The Structure of #P and the Polynomial Hierarchy

Recently, the possibility of approximating #P functions has been studied. Stockmeyer [Sto85] shows that Δ_3^p machines can approximate #P functions within a tight factor. Cai and Hemachandra [CH86b] show that the range of #P functions cannot be reduced to a sublinear size unless $\mathrm{P} = \mathrm{P}^{\#\mathrm{P}}$.

#P is intimately connected to the complexity of ranking—determining the position of elements in a set (Goldberg and Sipser [GS85], Rudich [Rud87], and (Chapter 4)).

Chapter 3

The Strong Exponential Hierarchy Collapses

3.1 Chapter Overview

The polynomial hierarchy, composed of the levels P, NP, P^{NP}, $\mathrm{NP}^{\mathrm{NP}}$, etc., plays a central role in classifying the complexity of feasible computations. It is not known whether the polynomial hierarchy collapses.

This chapter resolves the question of collapse for an exponential time analogue of the polynomial time hierarchy. Composed of the levels E (i.e., $\bigcup_c \mathrm{DTIME}[2^{cn}]$), NE, P^{NE}, $\mathrm{NP}^{\mathrm{NE}}$, etc., the strong exponential hierarchy collapses to P^{NE}, its Δ_2 level.

$$\mathrm{E} \neq \mathrm{P}^{\mathrm{NE}} = \mathrm{NP}^{\mathrm{NE}} \cup \mathrm{NP}^{\mathrm{NP}^{\mathrm{NE}}} \cup \cdots.$$

Our proof stresses the use of partial census information and the exploitation of nondeterminism.

Extending our techniques, we also derive new quantitative relativization results. We show that if the weak exponential hierarchy's Δ_{j+1} and Σ_{j+1} levels, respectively $\mathrm{E}^{\Sigma_j^p}$ and $\mathrm{NE}^{\Sigma_j^p}$, do separate, this is due to the large number of queries NE makes to its Σ_j^p database.[1]

[1] Σ_j^p is the j'th level of the polynomial hierarchy [Sto77].

Our techniques provide a successful method of proving the collapse of certain complexity classes.

3.2 Introduction

We wish to know if the high Δ and Σ levels of complexity hierarchies are computationally complex, and if so, *why* they are complex.

Section 3.3 looks at the strong exponential hierarchy:

$$\mathrm{E} \cup \mathrm{NE} \cup \mathrm{NP}^{\mathrm{NE}} \cup \mathrm{NP}^{\mathrm{NP}^{\mathrm{NE}}} \cup \cdots.$$

We show that the strong exponential hierarchy collapses to its Δ_2 level, P^{NE}. Our proof is based on a careful inspection of the computation tree involved in an $\mathrm{NP}^{\mathrm{NE}}$ computation. We show how P^{NE} can construct increasingly accurate partial census information about the number of "yes" responses NE makes to queries from NP in the action of $\mathrm{NP}^{\mathrm{NE}}$. Finally, we have the correct census and collapse the classes.[2]

The main result of Section 3.3 is:

Lemma 3.2 $\mathrm{P}^{\mathrm{NE}} = \mathrm{NP}^{\mathrm{NE}}$.

It follows that

$$\mathrm{E} \neq \mathrm{P}^{\mathrm{NE}} = \mathrm{NP}^{\mathrm{NE}} \cup \mathrm{NP}^{\mathrm{NP}^{\mathrm{NE}}} \cup \cdots.$$

Section 3.3.2 shows that this result does not follow simply from the fact that NE is a hard set. The section also notes that the combinatorics involved prevents this technique from collapsing the polynomial hierarchy [Sto77] to P^{NP}.

Section 3.4 uses the census techniques of Section 3.3 to prove new results on quantitative relativization—relativization with restrictions placed on oracle access. We first review the work on quantitative relativization of Book, Long, and Selman [BLS84,Lon85]. Then we show how our method of computing *partial census functions*, instead of the *names of strings* used in previous work, collapses complexity classes and unifies previous results.

[2] The author has been informed that recent work of Book, Schöning, Toda, Wagner, and Watanabe provides an alternative way of proving Lemma 3.2 [Wat87].

For the main result of Section 3.4, we study the weak exponential hierarchy, which is NE given a rich database:

$$\mathrm{NE} \cup \mathrm{NE}^{\mathrm{NP}} \cup \mathrm{NE}^{\mathrm{NP}^{\mathrm{NP}}} \cup \cdots.$$

We show that if the weak exponential hierarchy's Δ_i and Σ_i levels do separate, this is due not to the power of the database but to the large number of queries NE makes to the database.

Theorem 3.12 $\mathrm{E}^{\Sigma_k^p} = \{L \mid L \in \mathrm{NE}^{\Sigma_k^p}$ and some $\mathrm{NE}^{\Sigma_k^p}$ machine accepting L, for some c and every x, queries its Σ_k^p oracle at most $2^{c|x|}$ times in its entire computation tree on input $x\}$.

We also show that if $\mathrm{EXP}^{\mathrm{NP}}$ is to dominate P^{NE}, it must use the answers to early queries to help it pose later ones.

Theorem 3.19, Part 2 $\{S \mid S \leq^{exp}_{\text{truth-table}} \mathrm{NP}\} \subseteq \mathrm{P}^{\mathrm{NE}}$.

Finally, Section 3.5 lists open problems and summarizes the implications of our results.

Thus the high levels of the strong exponential hierarchy are no harder than the low levels—the strong exponential hierarchy collapses. The high levels of the weak exponential hierarchy separate completely only if NE floods its database with queries.

3.3 On the Strong Exponential Hierarchy

3.3.1 The Strong Exponential Hierarchy Collapses

This section proves that the strong exponential hierarchy collapses to its P^{NE} level. It suffices to collapse the strong exponential hierarchy's P^{NE} and $\mathrm{NP}^{\mathrm{NE}}$ levels. Then downward separation gives us a quick proof of the hierarchy's collapse.

Both $\bigcup_c \mathrm{DTIME}[2^{cn}]$ and $\bigcup_k \mathrm{DTIME}[2^{n^k}]$ are commonly referred to as exponential time (compare [CT86] with [BH77]), though the former is more common in the literature of structural complexity [Sel86,HY84]. We always make clear which exponential time we are speaking of. Our main result—the strong exponential hierarchy collapses—holds under either definition.

Definition 3.1 [3]

$$\begin{aligned}
\mathrm{E} &= \bigcup_c \mathrm{DTIME}[2^{cn}] \\
\mathrm{EXP} &= \bigcup_k \mathrm{DTIME}[2^{n^k}] \\
\mathrm{NE} &= \bigcup_c \mathrm{NTIME}[2^{cn}] \\
\mathrm{NEXP} &= \bigcup_k \mathrm{NTIME}[2^{n^k}] \\
\mathrm{SEH} &= \text{strong exponential hierarchy} \\
&= \mathrm{E} \cup \mathrm{NE} \cup \mathrm{NP}^{\mathrm{NE}} \cup \mathrm{NP}^{\mathrm{NP}^{\mathrm{NE}}} \cup \cdots \\
\mathrm{SEXPH} &= \mathrm{EXP} \cup \mathrm{NEXP} \cup \\
&\qquad \mathrm{NP}^{\mathrm{NEXP}} \cup \mathrm{NP}^{\mathrm{NP}^{\mathrm{NEXP}}} \cup \cdots .
\end{aligned}$$

Lemma 3.2 $\mathrm{P}^{\mathrm{NE}} = \mathrm{NP}^{\mathrm{NE}}$.

Theorem 3.3 $\mathrm{P}^{\mathrm{NE}} = \mathrm{SEH}$.

Corollary 3.4 $\mathrm{E} \neq \mathrm{P}^{\mathrm{NE}} = \mathrm{SEH} = \mathrm{P}^{\mathrm{NEXP}} = \mathrm{SEXPH}$.

Proof of Theorem 3.3 Define $\Sigma_1^{\mathrm{SEH}} = \mathrm{NE}$, $\Sigma_{k+1}^{\mathrm{SEH}} = \mathrm{NP}^{\Sigma_k^{\mathrm{SEH}}}$ for $k \geq 1$, and $\Delta_2^{\mathrm{SEH}} = \mathrm{P}^{\mathrm{NE}}$. By Lemma 3.2, $\Delta_2^{\mathrm{SEH}} = \Sigma_2^{\mathrm{SEH}}$. Inductively assume (for some $k \geq 2$) that $\Delta_2^{\mathrm{SEH}} = \Sigma_k^{\mathrm{SEH}}$. Now

$$\Sigma_{k+1}^{\mathrm{SEH}} = \mathrm{NP}^{\Sigma_k^{\mathrm{SEH}}} = \mathrm{NP}^{\Delta_2^{\mathrm{SEH}}} = \mathrm{NP}^{\mathrm{P}^{\mathrm{NE}}} = \mathrm{NP}^{\mathrm{NE}}.$$

The last equality holds because there is an NP machine that takes over the job of the P machine (in $\mathrm{NP}^{\mathrm{P}^{\mathrm{NE}}}$) and does all the NE queries itself.

Thus, by induction, $\Sigma_k^{\mathrm{SEH}} = \Delta_2^{\mathrm{SEH}}$ for all k, so $\mathrm{SEH} = \mathrm{P}^{\mathrm{NE}}$. □

Corollary 3.4 is proven in Section 3.3.3.

[3]It might seem natural to define an exponential hierarchy as $\mathrm{E} \cup \mathrm{NE} \cup \mathrm{NE}^{\mathrm{NE}} \cup \mathrm{NE}^{\mathrm{NE}^{\mathrm{NE}}} \cup \cdots$. However, by asking long queries to their oracles, these machines can unnaturally boost their power. For example, $\mathrm{NE}^{\mathrm{NE}}$ contains double exponential time and $\mathrm{NE}^{\mathrm{NE}^{\mathrm{NE}}}$ contains triple exponential time. Thus we can obtain trivial separations in this hierarchy by using the Hartmanis-Stearns time hierarchy theorem [HS65]. Even worse, an exponential hierarchy defined this way would not even be contained in EXPSPACE.

What causes this strange behavior is that a polynomial composed with a polynomial yields a polynomial, but an exponential function composed with an exponential function does not yield an exponential function. To avoid these anomalous behaviors, exponential hierarchies are defined by composing a single exponential function with many polynomial functions.

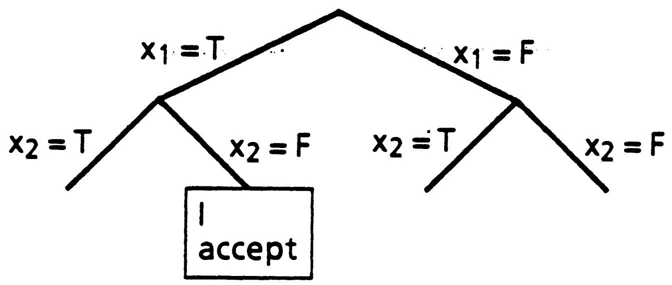


Figure 3.1: Nondeterministic Computation Tree

All the work of our result lies in the proof of Lemma 3.2. We make extensive use of the power of NE to guess query strings, witnesses, and paths in trees. We now give a sketch of the proof of Lemma 3.2; on page 35 we give a formal proof.

First, we wish to place graphically in mind our image of an $\mathrm{NP}^{\mathrm{NE}}$ computation. An NP computation tree has branches for each nondeterministic guess made by the NP machine. The machine is said to accept if any branch accepts [HU79]. For example, Figure 3.1 shows an NP machine checking the satisfiability of the formula $x_1 \wedge \overline{x_2}$. The NP machine has nondeterministically guessed all possible assignments and has found one that satisfies the formula. Throughout this chapter we assume, for simplicity of presentation, that our nondeterministic machines have at most two successor states for any given state.

We view an NP^{NE} computation similarly, except the NP machine can pose queries to an NE oracle. Each of the nondeterministic paths may, of course, pose different queries than its brothers do (Figure 3.2). We label the depth of the nodes in computation trees in the standard way (Figure 3.2).

Now we describe our strategy. Figure 3.3 shows the computation tree of an NP^{NE} machine. Our goal is to accept, with a P^{NE} machine, the same language the NP^{NE} machine accepts. The P^{NE} machine computes *the number of query strings receiving yes answers from NE at each depth of the tree.* For example, there are two yes strings at depth two in Figure 3.3.

We cannot simply jump in and compute the number of yes answers deep in the tree. To know which strings are even queried deep in the tree, we must first know the answers to queries more shallow in the tree. Thus we must be patient and first find the number of yes responses in the first level of the tree. Then using this knowledge, we find the number of yes responses at the second level, and so on. At each level we use knowledge of the previous levels to help us do a binary search for the number of strings at the current level.

For concreteness, suppose our NP^{NE} language is accepted by NP machine N_{17} with NE machine NE_{21} as its oracle. At a typical stage we know, for example, that the computation tree for ${N_{17}}^{NE_{21}}(x)$ has exactly 1,1,0,4, and 11 yes answers (queries of strings accepted by NE_{21}) at levels 0,1,2,3,4, respectively, and between 8 and 16 at level 5. Our question (asked by P to NE) is: given x and assuming 1,1,0,4,11 are the correct number of yes answers at levels 0,1,2,3,4, are there at least 12 yes strings queried at level 5?

This is the kind of question that NE can answer. NE guesses the first six levels of N_{17}'s computation tree, checks that the tree and queries written are really the actions that N_{17} would take given the query answers guessed, checks that there are 1,1,0,4,11 alleged yes strings at levels 0,1,2,3,4 and ≥ 12 alleged yes strings at level 5, and guesses the proofs that these strings really are yes strings (i.e., are accepted by NE_{21}).

If 1,1,0,4,11 is correct, then the first 5 levels correspond to the first 5 levels of the actual computation tree of ${N_{17}}^{NE_{21}}(x)$, and if there are ≥ 12 yes strings at

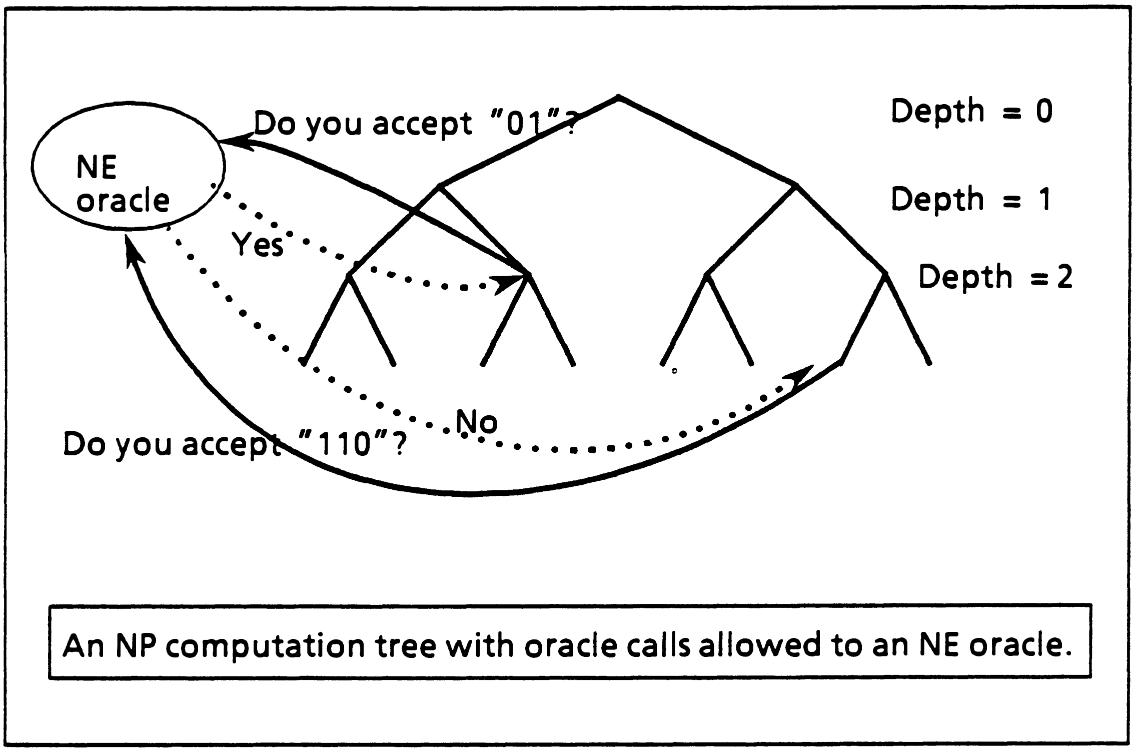


Figure 3.2: NP^{NE} Computation Tree

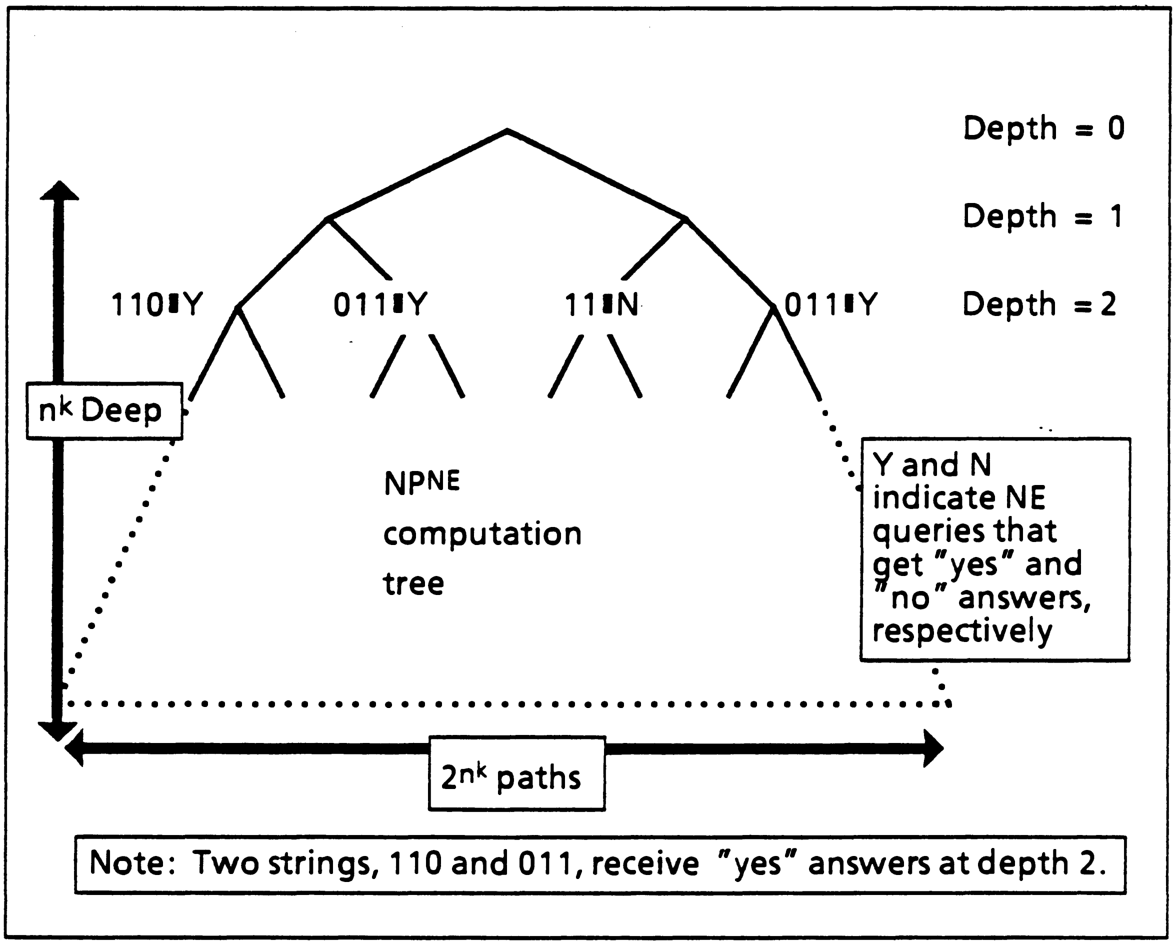


Figure 3.3: Our Strategy

level 5 level we will have guessed them.

Eventually we know the number of yes answers at each depth, and a final call of P to its NE oracle lets NE guess and check the correct computation tree of ${N_{17}}^{NE_{21}}(x)$.

Crucially, we do not need to verify that the no strings deserve "no" responses. We know the total number of yes answers at each level, and have proofs that strings do indeed get yes answers. Thus, in a guessed tree that gets all desired yes proofs, our no strings must indeed deserve "no" answers, as we have insured that our tree agrees with the action of N_{17}.

Let's refer to the P and NE machines we use to simulate NP^{NE} as $P_\star$ and $NE_\star$, i.e., we insure that $L({P_\star}^{NE_\star}) = L({N_{17}}^{NE_{21}})$. Figure 3.4 shows how the trees $NE_\star$ guesses increase in the (literal) depth of their accuracy at reflecting the tree of ${N_{17}}^{NE_{21}}(x)$. Crucially, $P_\star$ learns only the *number* of yes answers at each depth of ${N_{17}}^{NE_{21}}(x)$'s tree. This is fortunate; there is no way that $P_\star$ could remember all the yes *names* since deep in the ${N_{17}}^{NE_{21}}$ tree there may be as many as 2^{n^k} yes strings on a single level. Our use of increasingly accurate censuses of the number of queries per level is central to the success of our result.

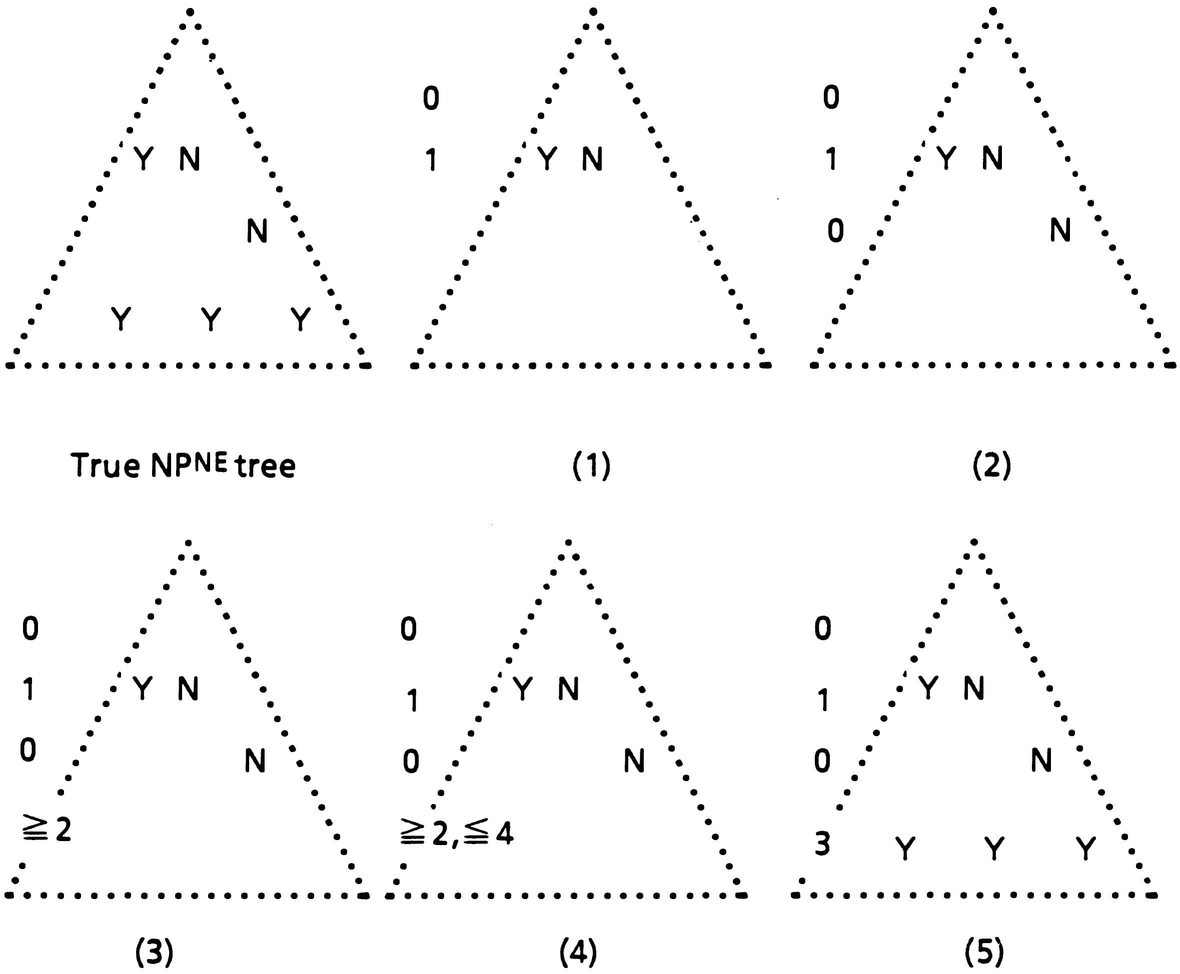
We now make precise the proof just sketched.

Proof of Lemma 3.2 ($P^{NE} = NP^{NE}$) Let L be an arbitrary language in NP^{NE}. For concreteness, suppose $L = L({N_{17}}^{NE_{21}})$, where N_{17} is an NP machine running in NTIME$[n^{17}]$ and NE_{21} is an NE machine. We describe machines $P_\star$ and $NE_\star$, respectively P and NE machines, so that $L = L({P_\star}^{NE_\star})$. Thus $P^{NE} = NP^{NE}$.

Let us first describe $NE_\star$.

$$L(NE_\star) = \{\mathit{final}\#1^{|x|^{18}}\#c_0\#c_1\#c_2\#\cdots\#c_l\#c_l \mid \text{There exist sets } C_0, C_1 \ldots, C_{l-1}, C_l \text{ of strings so:}$$

1. $|C_i| = c_i$ and $\bigcup_i C_i \subseteq L(NE_{21})$,
2. if we simulate ${N_{17}}^{()}(x)$, answering each oracle query q at depth i with a yes if and only if $q \in C_i$, then each y in C_i is really queried at level i in this simulation, and
3. if *final* is 1, there is an accepting path in the simulation mentioned in 2 above.$\}$



The true NP^{NE} tree and five snapshots showing a few of the increasingly correct images of the tree and its partial census information created in NE's mind. The numbers denote the number of queries on that level receiving a "yes" answer from the NE oracle.

Figure 3.4: The True NP^{NE} Tree and Five Snapshots

Table 3.1: Binary Search over Calls to $\mathrm{NE}_\star$ Discovers that there are Three Yes Strings at Level 3 of the Computation Tree of ${\mathrm{N}_{17}}^{\mathrm{NE}_{21}}(11101)$

Query	$\mathrm{NE}_\star$*'s Answer*
$\{0\#11101\#1^{5^{18}}\#0\#0\#1\#1\}$	*accept*
$\{0\#11101\#1^{5^{18}}\#0\#0\#1\#2\}$	*accept*
$\{0\#11101\#1^{5^{18}}\#0\#0\#1\#4\}$	*reject*
$\{0\#11101\#1^{5^{18}}\#0\#0\#1\#3\}$	*accept*

Given that we know the number of yes answers that appear in the first k levels of the computation tree of ${\mathrm{N}_{17}}^{\mathrm{NE}_{21}}(x)$, we can use binary search on $\mathrm{NE}_\star$ to find the number of yes answers that appear at level $k+1$.

Note that $L(\mathrm{NE}_\star)$ *is* in NE. The *value* of each c_i is at most $2^{|x|^{17}}$, since this is the maximum width of the computation tree ${\mathrm{N}_{17}}^{\mathrm{NE}_{21}}(x)$. So by guessing (at most) $|x|^{17}\cdot 2^{|x|^{17}}$ strings and then guessing proofs that each is in $L(\mathrm{NE}_{21})$ and guessing the paths by which each occurs in the simulation, we can implement $\mathrm{NE}_\star$ easily in $\mathrm{NTIME}[2^{|x|^{18}}]$. Note that we've padded so our input size to $\mathrm{NE}_\star$ is greater than $|x|^{18}$, so $\mathrm{NE}_\star$ runs in $\mathrm{NTIME}[2^{c\cdot\mathrm{Inputsize}}]$, and thus is in NE. Thus $L(\mathrm{NE}_\star)\in\mathrm{NE}$.

Now we describe the action of our machine $\mathrm{P}_\star$ (on input x). $\mathrm{P}_\star$ uses $\mathrm{NE}_\star$ to find the correct number of yes strings at each level of ${\mathrm{N}_{17}}^{\mathrm{NE}_{21}}(x)$.

Stage i: Inductively, we have numbers $c_0, c_1, \ldots, c_{i-1}$, so $c_0, c_1, \ldots, c_{i-1}$ are the correct number of yes strings at levels $0, 1, \ldots, i-1$ of the actual computation tree of ${\mathrm{N}_{17}}^{\mathrm{NE}_{21}}(x)$ (Figure 3.2). During this stage we find c_i, the actual number of yes strings at level i. This is easy—just perform binary search (varying z) using calls to $\mathrm{NE}_\star$ of the form

$$\{0\#x\#1^{|x|^{18}}\#c_0\#\cdots\#c_{i-1}\#z\}$$

to find the value of c_i.

Recall that, when $c_0, c_1, \ldots, c_{i-1}$ are correct, $\mathrm{NE}_\star$ says yes if z is a lower bound for the number of yes answers at level i. Table 3.1 gives a sample binary search run. At the end of it $\mathrm{P}_\star$ has learned that there are exactly 3 yes strings at level 3.

It is crucial to notice that when $c_0, \ldots, c_{i-1}$ are correct, the c_i we find is correct. This is because some branch B of $\mathrm{NE}_\star$ will guess the true yes strings $C_0, \ldots, C_{i-1}$. The queries asked at level i depend only on the fact that we know the right oracle replies at levels 0 through $i-1$; thus the queries asked at level i on branch B will be the queries that are asked in the tree of $\mathrm{N_{17}}^{\mathrm{NE}_{21}}(x)$ at level i. Thus if there are at least z yes strings queried at level i of the tree of $\mathrm{N_{17}}^{\mathrm{NE}_{21}}(x)$, an extension of B will guess them and accept.

On the other hand, any branch that does not guess the sets $C_0, \ldots, C_{i-1}$ correctly (it guesses, say, $C_0', \ldots, C_{i-1}'$) will certainly not accept. At the first level it errs from the true set of C_i's (say level m) it will not be able to find all the strings of its incorrect C_m' in the simulation tree. Why? Since $C_m' \neq C_m$, yet $|C_m'| = |C_m| = c_m$, some string w in C_m' is not a yes string of $\mathrm{N_{17}}^{\mathrm{NE}_{21}}(x)$ at level m. Since the C_i' are correct at levels $0, \ldots, m-1$, the strings queried at level m in the simulation are exactly those queried at level m in the tree of $\mathrm{N_{17}}^{\mathrm{NE}_{21}}(x)$. So if w is queried at level m, it is not in $L(\mathrm{NE}_{21})$ (if it were it would have to be in C_m); thus condition 1 of the definition of $L(\mathrm{NE}_\star)$ is violated and the branch won't accept. If w is not queried at level m, condition 2 of the definition of $L(\mathrm{NE}_\star)$ is violated and the branch won't accept.

End of Stage i

Since c_i can be at most $2^{|x|^{17}}$ in value (this is as wide as the tree of $\mathrm{N_{17}}^{\mathrm{NE}_{21}}(x)$ gets), the binary search process at stage i takes at most around $|x|^{17}$ steps, each requiring writing a string of length at most about $|x|^{17} \cdot |x|^{17} + |x|^{18} + |x|$. There are at most $|x|^{17}$ stages, so the total run time of $\mathrm{P}_\star^{(\cdot)}$ is easily polynomial; it runs in $\mathrm{TIME}[n^{4 \cdot 17+2}]$. Returning to the general case, if $\mathrm{N_{17}}$ runs in $\mathrm{NTIME}[n^k]$, then $\mathrm{P}_\star$ runs in deterministic $\mathrm{TIME}[n^{4k+2}]$.

After stage $|x|^{17}$, we know the correct values $c_0, \ldots, c_{|x|^{17}}$. At this point, a single call to $\mathrm{NE}_\star$ suffices. $\mathrm{P}_\star$ accepts if and only if $\mathrm{NE}_\star$ accepts $1\#x\#1^{|x|^{18}}\#c_0\#\cdots\#c_{|x|^{17}}$ (thus stating that $\mathrm{N_{17}}^{\mathrm{NE}_{21}}(x)$ accepts). □

3.3.2 Relationship of Our Collapse to Other Collapses and to the Polynomial Hierarchy

We have just shown that $\mathrm{P}^{\mathrm{NE}} = \mathrm{NP}^{\mathrm{NE}}$, and thus the strong exponential hierarchy collapses to P^{NE}. Is this collapse of the strong exponential hierarchy trivial in the same sense that $\mathrm{P}^{\mathrm{PSPACE}} = \mathrm{NP}^{\mathrm{PSPACE}} = \mathrm{PSPACE}$ is trivial? PSPACE is so powerful that both $\mathrm{P}^{\mathrm{PSPACE}}$ and $\mathrm{NP}^{\mathrm{PSPACE}}$ are equal to PSPACE. Do we similarly have $\mathrm{P}^{\mathrm{NE}} = \mathrm{NE}$? Relativization techniques help us here. There is a relativized world where $\mathrm{P}^{\mathrm{NE}^A} \supsetneq \mathrm{NE}^A$. That this is not a side effect of the ability of P^{NE} to reach length 2^{n^k} strings (compared with NE's reach of 2^{cn}) is shown by Theorem 3.5. Section 3.3.3 discusses this "reach" anomaly in detail.

Theorem 3.5 There is a recursive oracle A for which

$$\mathrm{P}^{\mathrm{NE}^A} \supsetneq \mathrm{NEXP}^A \supsetneq \mathrm{NE}^A.$$

Proof Sketch This is a straightforward diagonalization using the techniques of Baker, Gill and Solovay [BGS75]. We separate $\mathrm{P}^{\mathrm{NE}^A}$ from NEXP^A by forcing a coNE^A language out of NEXP^A. In particular, we diagonalize so that

$$L_A = \{0^n \mid (\forall y)[|y| = 2^n \Rightarrow y \notin A]\} \notin \mathrm{NEXP}^A.$$

Interlaced with this, we separate NEXP^A from NE^A simply using the fact that the former class is sensitive to length 2^{n^k} oracle strings. □

More generally, the collapse does not occur simply because NE is hard. Even though $\mathrm{P}^{\mathrm{NE}} = \mathrm{NP}^{\mathrm{NE}}$, there are many NE-hard sets, A, for which $\mathrm{P}^A \neq \mathrm{NP}^A$. This follows from direct diagonalization, and is related to Hartmanis's [Har85] results on the re-relativization of PSPACE.

On the other hand, our techniques do not collapse the polynomial hierarchy ($\mathrm{NP} \cup \mathrm{NP}^{\mathrm{NP}} \cup \cdots$) to P^{NP}. Suppose we tried to show that $\mathrm{P}^{\mathrm{NP}} = \mathrm{NP}^{\mathrm{NP}}$ using the above methods. $\mathrm{NP}^{\mathrm{NP}}$ may have exponentially many yes replies given to its lower NP machine by the upper one. The P machine *can* record an exponential *count*, but the NP machine sitting over it (in P^{NP}) certainly cannot guess an exponentially large object: the names of the 2^{n^k} yes strings in the tree of $\mathrm{NP}^{\mathrm{NP}}$.

Of course, if we change the game so that, in $\mathrm{NP}^{\mathrm{NP}}$, few queries are made to the oracle, then the same argument works. This "quantitative relativization" approach is what is done by Book, Long, and Selman in [BLS84], where they show that a hierarchy of quantified relativizations collapses. Section 3.4 of this thesis studies quantified relativizations of the exponential hierarchy, where we'll see the partial census technique of this section put to further use.

3.3.3 The Strong Exponential Hierarchy and Sensitivity to Padding

Definition 3.6 1. The weak exponential hierarchy, EH [HIS85], is $\mathrm{NE} \cup \mathrm{NE}^{\mathrm{NP}} \cup \mathrm{NE}^{\mathrm{NP}^{\mathrm{NP}}} \cup \cdots$.

2. The EXP hierarchy, EXPH, is $\mathrm{NEXP} \cup \mathrm{NEXP}^{\mathrm{NP}} \cup \mathrm{NEXP}^{\mathrm{NP}^{\mathrm{NP}}} \cup \cdots$.

SEH is "strong" because in a relativized world A, SEH^A is not contained in EH^A. This is simply because from its Δ_2^A level ($\mathrm{P}^{\mathrm{NE}^A}$) on up, SEH^A can query strings in A of length 2^{n^k}, but EH^A can only query strings of length 2^{cn}. To understand this, just reflect on the fact that

$$\mathrm{P}^{\mathrm{E}} \supsetneq \mathrm{E}^{\mathrm{P}},$$

since $\mathrm{E}^{\mathrm{P}} = \mathrm{E}$ but $\mathrm{P}^{\mathrm{E}} = \mathrm{EXP}$, and $\mathrm{EXP} \supsetneq \mathrm{E}$ by the time hierarchy theorem of Hartmanis and Sterns [HS65].

Theorem 3.7 There is a relativized world A so that $\mathrm{SEH}^A - \mathrm{EH}^A \neq \emptyset$. Indeed, in this world $\mathrm{P}^{\mathrm{NE}^A} - \mathrm{EH}^A \neq \emptyset$.

Proof We make $L_A \in \mathrm{P}^{\mathrm{NE}^A} - \mathrm{EH}^A$, where

$$L_A = \{0^n \mid (\exists y)[y \in A \wedge |y| = 2^{n^2}]\}.$$

L_A is clearly in $\mathrm{P}^{\mathrm{NE}^A}$, but since no EH^A machine can reach strings of length 2^{n^2} on inputs of length n, we can easily diagonalize against each EH machine. □

Similarly we get the following separation that is due wholly to this padding anomaly.

Fact 3.8 $\Sigma_0^{\mathrm{SEH}} \neq \Delta_2^{\mathrm{SEH}}$, i.e., $\mathrm{E} \neq \mathrm{P}^{\mathrm{NE}}$.

This sensitivity of E and NE oracles to polynomial padding of their input strings has another consequence. The hierarchy SEXPH (Definition 3.1) equals SEH. Why? Clearly $\mathrm{P}^{\mathrm{NE}} = \mathrm{P}^{\mathrm{NEXP}}$, $\mathrm{NP}^{\mathrm{NEXP}} = \mathrm{NP}^{\mathrm{NE}}$, and so forth, since if P (in P^{NE}) just adds polynomial padding to each query string, its NE oracle can simulate the NEXP calls of $\mathrm{P}^{\mathrm{NEXP}}$. Thus we have extended the collapsing result of Section 3.3.

Corollary 3.4
$\mathrm{E} \neq \mathrm{P}^{\mathrm{NE}} = \mathrm{SEH} = \mathrm{P}^{\mathrm{NEXP}} = \mathrm{SEXPH} =_{\mathrm{def}} \mathrm{NEXP} \cup \mathrm{NP}^{\mathrm{NEXP}} \cup \mathrm{NP}^{\mathrm{NP}^{\mathrm{NEXP}}} \cup \cdots$.

3.3.4 Downward Separations

If we collapse the polynomial hierarchy at any level, the entire hierarchy collapses to that level; $\Sigma_i^p = \Pi_i^p \Rightarrow \Sigma_i^p = \mathrm{PH}$ [Sto77]. This is known as *downward separation*. One troubling feature of the exponential hierarchy is that it does not have downward separation. Hartmanis, Immerman, and Sewelson [HIS85] display a relativized world A where $\mathrm{E}^A = \mathrm{NE}^A \neq \mathrm{NE}^{\mathrm{NP}^A}$.

On the other hand, the strong exponential hierarchy *does* have downward separation of a sort. A subtlety is that we must account for the sensitivity to padding discussed in the previous section.

Theorem 3.9 (Downward Separation)

1. $\mathrm{E} = \mathrm{NE} \Rightarrow \mathrm{EXP} = \mathrm{SEH}$
2. $\mathrm{NE} = \mathrm{coNE} \Rightarrow \mathrm{NEXP} = \mathrm{SEH}$.
3. $\mathrm{EXP} = \mathrm{NEXP} \Rightarrow \mathrm{EXP} = \mathrm{SEXPH}$.
4. $\mathrm{NEXP} = \mathrm{coNEXP} \Rightarrow \mathrm{NEXP} = \mathrm{SEXPH}$.

Proof of Theorem 3.9 (Using Theorem 3.3)

1. Using Theorem 3.3 and our assumption that $\mathrm{E} = \mathrm{NE}$,

$$\mathrm{EXP} \subseteq \mathrm{SEH} = \mathrm{P}^{\mathrm{NE}} = \mathrm{P}^{\mathrm{E}} = \mathrm{EXP}.$$

2. When NE = coNE (and thus even "no" answers to NE queries have certificates), P^{NE} can be simulated by NEXP, which just guesses the correct oracle answers along with their certificates of correctness.

3. See proof of 1.

4. See proof of 2. □

Alternate Direct Proof of Theorem 3.9

1. $NP^{E} = EXP$ ($\subseteq$ by brute force tree simulation, $\supseteq$ by the padding trick of Section 3.3.3). Also, $NP^{E} = NP^{EXP}$ by padding (see Section 3.3.3). Thus if $E = NE$ we have:

$$\begin{aligned} EXP &= NP^{E} = NP^{NE} \\ EXP &= NP^{E} = NP^{EXP} = NP^{NP^{E}} = NP^{NP^{NE}} \\ &\text{and so on} \end{aligned}$$

2. $NP^{NE \cap coNP} \subseteq NEXP$, since NEXP simulates the NP machine and guesses the correct oracle answers with their certificates. Also, $NP^{NEXP} = NP^{NE}$ by padding. Thus if NE = coNE the hierarchy collapses as follows.

$$\begin{aligned} NEXP &\supseteq NP^{NE \cap coNE} \\ &= NP^{NE} \\ NEXP &\supseteq NP^{NE} \\ &= NP^{NEXP} \\ &\supseteq NP^{NP^{NE \cap coNE}} \\ &= NP^{NP^{NE}} \\ &\text{and so on} \end{aligned}$$

□

There is no point in stating more general downward separation results. Since we already know that $P^{NE} = SEH$ with no assumptions needed, the results above are the only nontrivial downward separations possible in the strong exponential hierarchy.

3.4 Quantitative Relativization Results

3.4.1 Definitions

Quantitative relativization means relativization in which the power of the base machine to query its oracle is restricted. Before studying quantitative relativization, we define our notation. Since varied notations have previously been used, we take this opportunity to define a notation that is transparent.

Definition 3.10 Let A and B be complexity classes. The class

$$A^{B[f(n)]_{path}[g(n)]_{tree}[h(n)]_{named-restriction}}$$

is the class of languages accepted by an oracle Turing machine from A with a set from B as its oracle, under the restrictions that:

1. on input of size n, each path of A's computation tree (A may be nondeterministic) queries B about at most $f(n)$ different strings, and
2. the total number of strings B is queried about throughout A's entire computation tree is $g(n)$, and
3. there is an $h(n)$ bound on whatever is named by "named-restriction."

Examples and Notes

1. $\mathrm{NE}^{\mathrm{NP}} = \bigcup_c \mathrm{NE}^{\mathrm{NP}[2^{2^{cn}}]_{tree}}$. Since the NE computation tree is 2^{cn} deep it only has $2 \cdot 2^{2^{cn}}$ nodes, so it cannot make more than $2 \cdot 2^{2^{cn}}$ queries (Figure 3.5a).

2. $\mathrm{NE}^{\mathrm{NP}[2^{cn}]_{tree}}$ *does* put some restriction on the querying action of NE (Figure 3.5b). Note that $\mathrm{NE}^{\mathrm{NP}[1]_{path}}$ (Figure 3.5c) may query $2^{2^{cn}}$ many strings—one on each of NE's $2^{2^{cn}}$ computation paths (Figure 3.5d).

3. In classes such as $\mathrm{NEXP}^{\mathrm{NP}}$, NEXP may query NP about strings of length exponential in the input size. Thus it is not obvious and probably not true that NEXP equals $\mathrm{NEXP}^{\mathrm{NP}}$, even though $\mathrm{NP} \subseteq \mathrm{NEXP}$.

Table 3.2: Nomenclature of Quantitative Relativization

Class $= A^{B\,\text{restrictions}}$	
Restrictions	
$[\textit{time bound}]_{path}$	*limit on the number of strings queried on each of A's computation paths*
$[\textit{time bound}]_{tree}$	*limit on the number of strings queried throughout A's computation tree*
$[\textit{time bound}]_{Y,tree}$	*limit on the number of strings queried throughout A's computation tree that are in B*
Time Bounds	
log	$\bigcup_c \mathrm{TIME}[c \log n]$
poly	$\bigcup_k \mathrm{TIME}[n^k]$
e	$\bigcup_c \mathrm{TIME}[2^{cn}]$
exp	$\bigcup_k \mathrm{TIME}[2^{n^k}]$

4. Our quantitative classes count strings—not oracle calls. That is, $[n^2]_{tree}$ means that for all x we have

$$|\{y \mid y \text{ is queried in the tree on input } x\}| \leq |x|^2.$$

For example, an NE computation tree that queries string y on each of its $2^{2^{cn}}$ branches would be charged just "1" in the $[\cdot]_{tree}$ measure for this whole set of queries—*not* $2^{2^{cn}}$ (Figure 3.5d).

Finally, we make it harder to prove our theorems.

Notation 3.11 Adding a "Y" (e.g., $\mathrm{NE}^{\mathrm{NP}[2^{cn}]_{Y,path}}$) means we are counting just the number of strings queried that get a "yes" answer from the oracle (and "no" answers don't count). This makes our theorems harder to prove and stronger; any theorem we prove with a "Y" also holds without the "Y." Table 3.2 summarizes our notation.

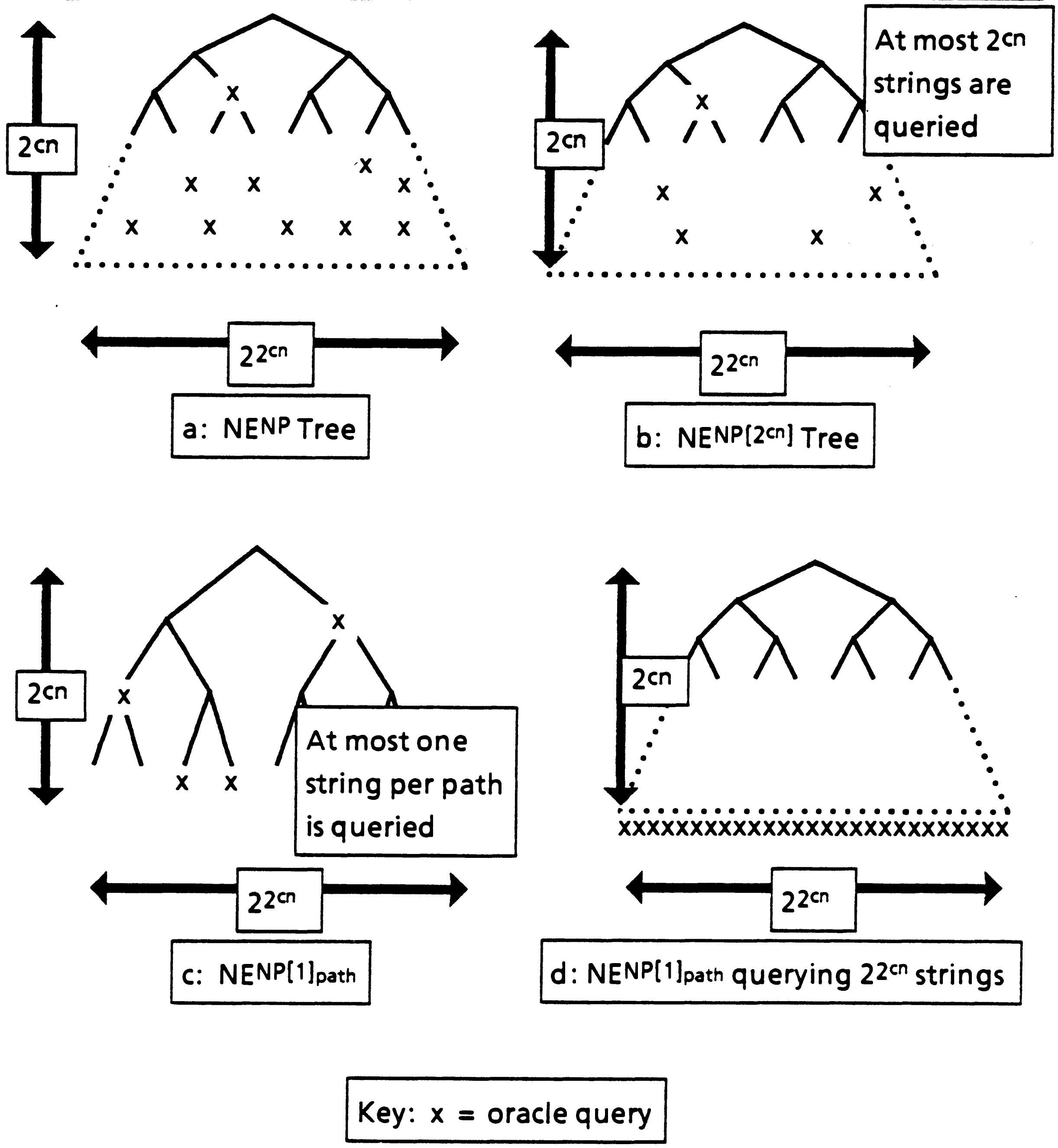


Figure 3.5: Quantitative Relativizations

3.4.2 Introduction and Background

The census techniques of Section 3.3.1 will be used to prove new theorems about quantitative relativizations. We'll see, for example, that:

$$\mathrm{E}^{\mathrm{NP}} = \mathrm{NE}^{\mathrm{NP}[e]_{tree}}.$$

Thus the only way $\mathrm{NE}^{\mathrm{NP}}$ can avoid collapsing to E^{NP} is by using its oracle often. Similarly, we'll see that:

$$\mathrm{P}^{\mathrm{NE}} \supseteq \mathrm{NE}^{\mathrm{NP}[poly]_{tree}}.$$

Note that the P machine here cannot write down even one of the long queries (length 2^{cn}) the NE machine makes to NP, yet P^{NE} can nonetheless simulate the action of $\mathrm{NE}^{\mathrm{NP}[poly]_{tree}}$.

This highlights the difference between our techniques and those found in previous work. Previous quantitative relativization results, centered in the polynomial hierarchy, require that the base machine obtain the *names* of all strings queried. However, our base class P can only use a polynomial amount (n^j) of tape. So previous methods did not prove that $\mathrm{P}^{\mathrm{NE}} \supseteq \mathrm{NE}^{\mathrm{NP}[poly]_{tree}}$, as each queried name is too long (length 2^{cn}). Previous methods did not prove that $\mathrm{P}^{\mathrm{NE}} = \mathrm{NP}^{\mathrm{NE}}$, as there are too many names (the NP tree has 2^{n^k} nodes).

The body of previous work in quantitative relativization contains many gems [Boo81][SMB83][BLS84][Lon85]. Book, Long, and Selman [BLS84] prove that $\mathrm{P}^{\mathrm{NP}} = \mathrm{NP} \cup \mathrm{NP}^{\mathrm{NP}[poly]_{tree}} \cup \mathrm{NP}^{(\mathrm{NP}^{\mathrm{NP}[poly]_{tree}})[poly]_{tree}} \cup \cdots$, which provides a polynomial analogue of our $\mathrm{P}^{\mathrm{NE}} = \mathrm{NP}^{\mathrm{NE}}$ result. Their paper provides a detailed study of quantitative relativization in a polynomial setting. Recently, Long [Lon85] extended this work by noting that many quantitative relativization results still hold when we restrict our counting to yes queries.

3.4.3 Quantitative Relativization Theorems

This section proves theorems that give insight into what makes hierarchies nontrivial. The first and most important theorem shows that the Σ_{k+1} and Δ_{k+1} levels of EH (Definition 3.6) collapse unless the Σ_{k+1} level, $\mathrm{NE}^{\Sigma_k^p}$, uses its Σ_k^p

oracle extensively. This is related to the polynomial time results of Long [Lon85, page 595].

Theorem 3.12 $\mathrm{E}^{\Sigma_k^p} = \mathrm{NE}^{\Sigma_k^p[e]_{Y,\,tree}}$, $k > 0$.

Corollary 3.13

1. $\mathrm{E}^{\mathrm{NP}} = \mathrm{NE}^{\mathrm{NP}[e]_{Y,\,tree}}$.
2. $\mathrm{E}^{\mathrm{NP}} = \mathrm{NE}^{\mathrm{NP}[e]_{tree}}$.

Proof Sketch of Theorem 3.12 The method of Section 3.3.1 works with a subtle but important modification. In that section, we could guess a whole computation tree and check it. Here, the computation tree of NE is too huge to do this; it has around $2^{2^{cn}}$ nodes. Luckily, since there is a $[e]_{Y,\,tree}$ bound we are interested in checking at most 2^{cn} nodes of this tree. Our new trick is that we just guess and check the parts of the tree that are of interest to us.

So now, our Σ_k^p machine called by E will just (1) guess the yes strings, (2) guess the paths in the computation tree of $\mathrm{NE}^{\Sigma_k^p}$ that lead to them, and (3) verify that the strings really are accepted by NE's oracle. Note that (3) is subtle; Σ_k^p is closed under intersection, so checking if all our guessed strings are accepted by NE's oracle is a Σ_k^p question. We use the same existential block (i.e., the first existential block of E's oracle) that tackles (1) and (2) to start on the first existential block of the membership checking question.

This can all be done in $2^{c'n}$ steps, and thus allow E's Σ_k^p oracle to guess and check all the portions of the huge $\mathrm{NE}^{\Sigma_k^p}$ tree that are of interest to it. □

The following theorem shows that $\mathrm{NE}^{\mathrm{NP}}$ is dominated by P^{NE} unless $\mathrm{NE}^{\mathrm{NP}}$ makes many oracle calls. The result is surprising as P^{NE} makes only polynomially many nondeterministic queries, but $\mathrm{NE}^{\mathrm{NP}[poly]_{path}[exp]_{tree}}$ makes exponentially many queries, any one of which may be far too long for P to record.

Theorem 3.14 $\mathrm{P}^{\mathrm{NE}} \supseteq \mathrm{NE}^{\mathrm{NP}[poly]_{path}[exp]_{Y,\,tree}}$.

Corollary 3.15

1. $\mathrm{P}^{\mathrm{NE}} = \mathrm{NEXP}^{\mathrm{NP}[poly]_{path}[exp]_{Y,\,tree}}$.

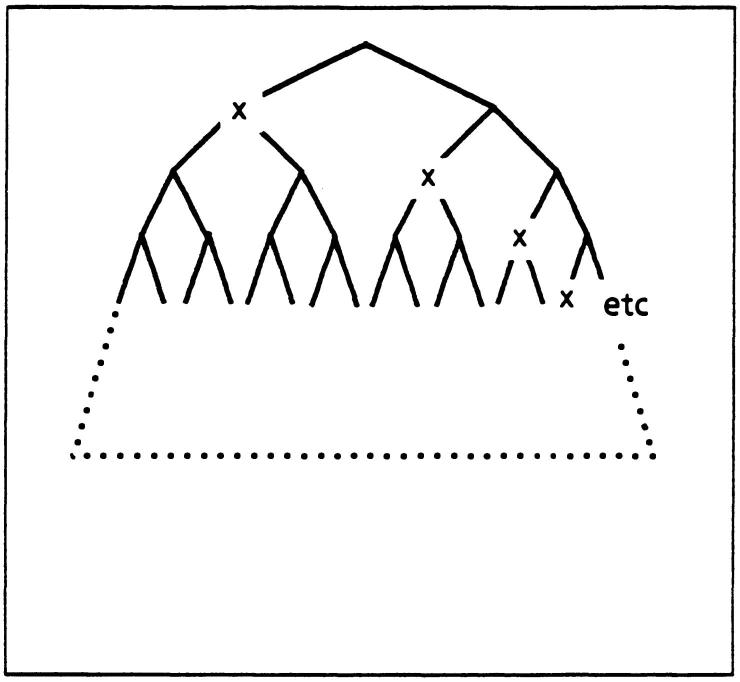


Figure 3.6: One Query per Path, but Many Levels Have Queries

2. $\mathrm{P}^{\mathrm{NE}} = \mathrm{NEXP}^{\mathrm{NP}[poly]_{tree}}$.

Proof Sketch of Theorem 3.14

We need a new trick here. Our problem is that P^{NE} can only store the query census for polynomially many levels. However, though $\mathrm{NE}^{\mathrm{NP}[poly]_{path}[exp]_{Y,\,tree}}$ has only polynomially many queries per path, it might have queries at exponentially many levels (Figure 3.6).

Our trick is to associate with each query its *query depth*: how many queries are made before it on its path (Figure 3.7). P^{NE} will have polynomially many rounds. In each round i it will find, by binary search, the number of yes answers

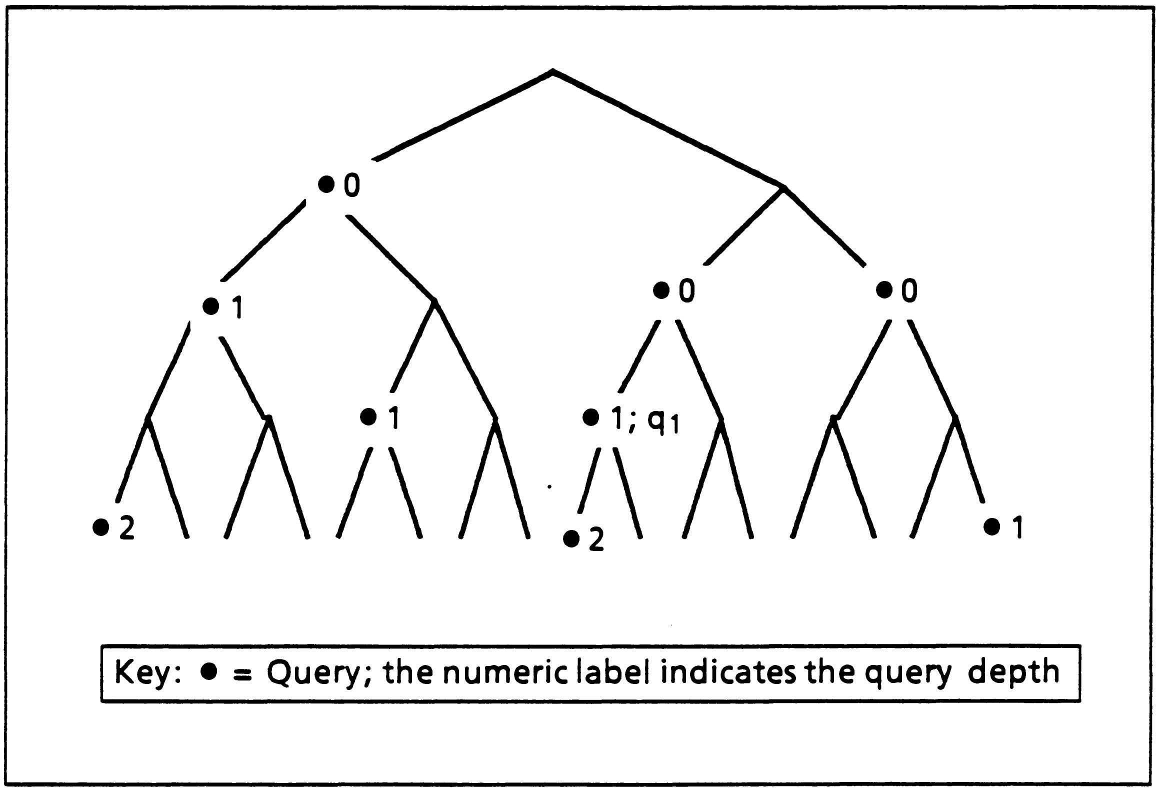


Figure 3.7: Query Depths

received by queries at query depth i.

Since the query depth of the tree of $\mathrm{NE}^{\mathrm{NP}[poly]_{path}[exp]_{Y,\,tree}}$ is polynomial, and since getting the right string to ask for a query depth i query depends only on having queries of query depth less than i correctly answered, the method of Section 3.3.1 now works. After polynomially many rounds of binary search the P machine knows the census of yes answers at each query depth and can finish off its simulation with one final query to NE. □

Proof of Corollary 3.15 These results use the same argument as Theorem 3.14, but since $\mathrm{P}^{\mathrm{NE}} \subseteq \mathrm{NEXP}^{\mathrm{NP}[poly]_{tree}} \subseteq \mathrm{NEXP}^{\mathrm{NP}[poly]_{path}[exp]_{Y,\,tree}}$, we get equality in this corollary. □

It is easy to see from Corollary 3.15 that $\mathrm{NEXP}^{\mathrm{NP}}$ is a delicate class. If you restrict its query access too much, it is weakened to the point of collapsing to P^{NE}.

3.4.4 More Quantitative Relativization Theorems

We can tailor and extend the techniques we've developed to a range of problems. This section presents three such problems and briefly outlines the modifications needed.

3.4.4.1 Paying Only for the First Occurrence

Note that, within a computation tree, a query consists of both a string queried, and the location in the tree at which the query is made. The *first occurrence depth* (Figure 3.8 gives examples) of an oracle query in a computation tree is 0 if there are no queries made along the path between it and the root. Inductively, an oracle query is of *first occurrence depth* $i+1$ if (1) the string associated with the query has not been already been assigned a query depth less that $i+1$ and (2) the largest query depth of a query along the path from the query to the root is i. (Figure 3.8). Intuitively, the first occurrence depth of a query string is how many times we must expand from the root our "frontier" of queries to reach the query string.

Simply put, the following theorem says that once a query is paid for by a nondeterministic branch, its brother branches get to query the same string for

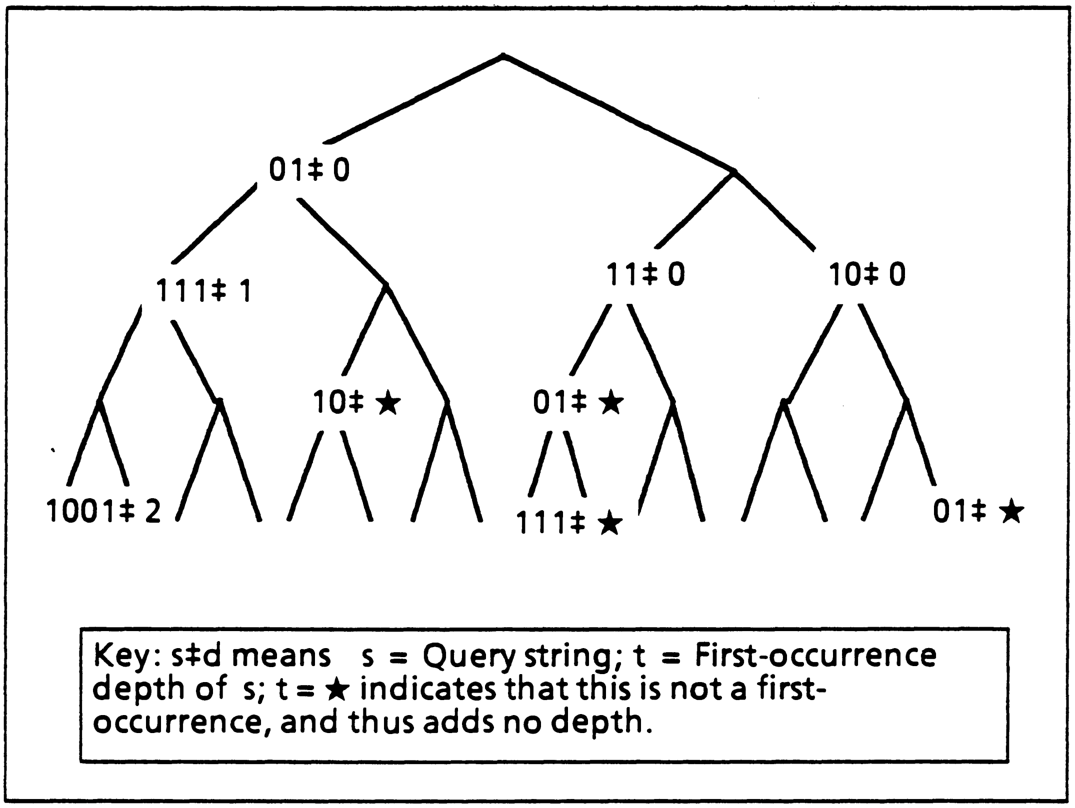


Figure 3.8: First-Occurrence Depths

free.

Theorem 3.16 $\mathrm{P}^{\mathrm{NEXP}} = \mathrm{NEXP}^{\mathrm{NP}[exp]_{tree}[poly]_{first\ occurrence\ depth}}$.

3.4.4.2 Many "No" Strings

P^{NE} can query its oracle only a polynomial number of times. Is it possible that P^{NE} can contain $\mathrm{NE}^{\mathrm{NP}[poly]_{Y,\,tree}}$, a class that may make up to $2^{2^{cn}}$ oracle calls (most receiving "no" answers)? The following theorem shows that it does.

Theorem 3.17 $\mathrm{P}^{\mathrm{NE}} \supseteq \mathrm{NE}^{\mathrm{NP}[poly]_{Y,\,tree}}$.

Corollary 3.18 $\mathrm{P}^{\mathrm{NE}} = \mathrm{NEXP}^{\mathrm{NP}[poly]_{Y,\,tree}}$.

Our trick is for P^{NE} to do binary search to find the *depth* of the first "yes" response in the run of $\mathrm{NE}^{\mathrm{NP}[poly]_{Y,\,tree}}(\cdot)$. Then as before it does binary search to find the number of yes strings at that level. P^{NE} repeats this level-finding/census-finding alternation until it has gone through all the levels that have yes responses (it can detect this).

3.4.4.3 Truth-Table Classes

Finally, we mention some easy but surprising results, probably part of the folklore, about truth-table classes [Yes83]. They follow not from the method of Section 3.3.1, but simply from the weakness of truth table reductions.

Theorem 3.19

1. $\{S \mid S \leq^{p}_{\mathbf{truth-table}} \mathrm{NP}\} \subseteq \mathrm{P}^{\mathrm{NP}[log]}$.
2. $\{S \mid S \leq^{exp}_{\mathbf{truth-table}} \mathrm{NP}\} \subseteq \mathrm{P}^{\mathrm{NE}}$.

Proof We prove the second part. $S \leq^{exp}_{\mathbf{truth-table}} \mathrm{NP}$ means there is a machine that answers "$x \in S$?" by making exponentially many queries to SAT [GJ79], such that the queries asked of SAT are independent of the answers received (see Ladner, Lynch, and Selman [LLS75] for a discussion of $\leq^{p}_{\mathbf{truth-table}}$).

To prove part 2 of the theorem, we have P perform binary search, using its NE oracle, to find the *number* of yes answers, and with one final query to the oracle

have NE guess the yes answers and simulate the action of the truth-table reducer.
□

Theorem 3.19 says that if $\mathrm{EXP}^{\mathrm{NP}}$ is to strictly contain P^{NE}, it *must* use the answers to early queries to help it pose later ones. Of course, we don't hope to show $\mathrm{EXP}^{\mathrm{NP}} \supsetneq \mathrm{P}^{\mathrm{NE}}$, as this implies $\mathrm{P} \neq \mathrm{NP}$. However, we suspect that $\mathrm{P} \neq \mathrm{NP}$ and even $\mathrm{EXP}^{\mathrm{NP}} \supsetneq \mathrm{P}^{\mathrm{NE}}$.

3.5 Conclusions and Open Problems

- The weak exponential hierarchy, EH, has characterizations in terms of the alternating Turing machines of Chandra, Kozen, and Stockmeyer [CKS81] and in terms of quantified formulas. EH is exactly the languages L of the form that for some k and c:

  $$L = \{x \mid (\exists_{\mathrm{E}} y_1)(\forall_{\mathrm{E}} y_2) \cdots (Q_{k,\mathrm{E}} y_k)[R(x, y_1, \ldots, y_k)]\},$$

  where R is a polynomial time predicate and an "E" subscript denotes a 2^{cn} bound on the quantifier size (e.g., "$(\exists_{\mathrm{E}} y)[$" is short for "$(\exists y)[|y| \leq 2^{cn} \wedge$") [HIS85]. In terms of alternating Turing machines, EH models alternating Turing machines with a bounded number of 2^{cn}-sized alternation blocks. Similarly, EXPH models alternating Turing machines with a bounded number of 2^{n^k}-sized alternation blocks. Does the strong exponential hierarchy have similar representations via quantifiers and alternating Turing machines?

- Is there a relativized world where $\mathrm{EXP}^{\mathrm{NP}} \supsetneq \mathrm{P}^{\mathrm{NE}}$ (see the discussion following the proof of Theorem 3.19)? We conjecture that there is.

Our goal was to find out if exponential hierarchies collapse, or if not, why they might separate. We've seen that the strong exponential hierarchy collapses, via a tight analysis of the query census of $\mathrm{NP}^{\mathrm{NE}}$. We've viewed the weak exponential hierarchy as NE with a rich database, and seen that the Δ and Σ levels of the weak exponential hierarchy separate only if NE floods its database with queries.

More generally, the census techniques of this chapter can be used as a general procedure for collapsing complexity classes that meet certain counting conditions [Hem86b].

Chapter 4

The Complexity of Ranking

4.1 Chapter Overview

This chapter structurally characterizes the complexity of ranking.

A set A is (strongly) P-rankable if there is a polynomial time computable function f so that for all x, $f(x)$ computes the number of elements of A that are lexicographically $\leq x$, i.e., the *rank* of x with respect to A [GS85]. This is the strongest of three notions of P-ranking we consider in this chapter. We say a class $\mathcal{C}$ is P-rankable if all sets in $\mathcal{C}$ are P-rankable. Our main results show that with the same certainty with which we believe counting to be complex, and thus with at least the certainty with which we believe $\mathrm{P} \neq \mathrm{NP}$, we believe that P has no uniform, strong, weak, or enumeratively approximate ranking functions.

We show that:

- P and NP are equally likely to be P-rankable, i.e., P is P-rankable if and only if NP is P-rankable.
- P is P-rankable if and only if $\mathrm{P} = \mathrm{P}^{\#\mathrm{P}}$. This extends work of Blum, Goldberg, and Sipser [GS85].
- Even the two weaker notions of P-ranking that we study are hard if $\mathrm{P} \neq \mathrm{P}^{\#\mathrm{P}}$.
- PSPACE is P-rankable if and only if $\mathrm{P} = \mathrm{PSPACE}$.

- If P has small ranking circuits, then it has small ranking circuits of relatively low complexity.
- If P has small ranking circuits, then the power of counting falls into the polynomial hierarchy (i.e., $\mathrm{P}^{\#\mathrm{P}} \subseteq \Sigma_2^p = \mathrm{PH}$).
- P/poly, the class of sets with small circuits, is not P-rankable.
- P/poly has small ranking circuits if and only if $\mathrm{P}^{\#\mathrm{P}}/\mathrm{poly} = \mathrm{P}^{\#\mathrm{P}/\mathrm{poly}} = \mathrm{P}/\mathrm{poly}$.
- If P is P-rankable, then P/poly has small ranking circuits. This links the ranking complexity of uniform and nonuniform classes.
- The ranks of some strings in easy sets are of high relative time-bounded Kolmogorov complexity unless $\mathrm{P} = \mathrm{P}^{\#\mathrm{P}}$. It follows that even a type of approximate ranking, enumerative ranking, is hard unless $\mathrm{P} = \mathrm{P}^{\#\mathrm{P}}$. This partially resolves a question posed by Goldberg and Sipser [GS85, pp. 447–448].

4.2 Introduction

This chapter characterizes the complexity of ranking sets from standard complexity classes such as P, NP, PSPACE, and P/poly. Over the past decade, much effort has been spent studying the complexity of the *membership problem* for languages from NP and PSPACE [GJ79]. At the same time, Valiant [Val79a,Val79b], Angluin [Ang80], Stockmeyer [Sto85], Valiant and Vazirani [VV85], and Jerrum, Valiant, and Vazirani [JVV86] have studied the complexity of the *counting problem* for NP: how hard is it to determine the number of satisfying assignments that a given boolean formula has?

The counting problem is formalized by Valiant's counting class #P [Val79a], the class of functions that count the accepting paths of a nondeterministic polynomial time (NP) Turing machine. A function f is in #P if and only if there is an NP machine N so that for every x, $f(x)$ equals the number of accepting paths of $N(x)$. $\mathrm{P}^{\#\mathrm{P}}$ is the *language* class of sets accepted by some polynomial time machine with a #P oracle.

This chapter studies the *ranking problem*: given a fixed set, how hard is it to determine the rank of elements within that set. Intuitively put, the ranking problem seems harder than the membership problem[1] and is closely related to the counting problem.

Studying the ranking problem gives a new perspective on the determinism-vs-nondeterminism question. It is likely that $P \neq NP$, and thus the membership problem for NP is harder than the membership problem for P. Nonetheless, we shall show that the ranking problems of NP and P stand or fall together: either both are easy or both are hard.

Another reason that ranking is of interest is that it is a natural generalization to non-sparse sets of the concept of P-printability.[2] A set S is *P-printable* if there is a polynomial time function that on input 1^n prints all strings in S of length $\leq n$ (Hartmanis and Yesha [HY84]). Recent work by Allender [All86], Balcazar and Book [BB86], and Hartmanis and Hemachandra [HH86b] has shown P-printability to be intimately related to Kolmogorov complexity and to the $P = NP$ question. [BB86] and [HH86b] show that in worlds where $P = NP$, the sparse sets S that separate NP^S from P^S are exactly those that are P^S-printable. It is well known that a sparse set is strongly P-rankable (defined below) if and only if it is P-printable.

Thus rankability provides one natural generalization to dense sets of the notion of P-printability, which applies only to sparse sets. Other generalizations of printability to dense sets are self-encodability and self-P-producibility, essentially equivalent notions, which have been explored by Ko, Long, and Selman (see [Ko85, page 228]), Balcazar and Book [BB86], and Hartmanis and Hemachandra [HH86b].

Sipser [GS85] proves that 1-way-Logspace sets have easy ranking problems and Allender [All85] proves that certain classes of automata accept only sets with easy ranking functions. Nonetheless, Blum, Goldberg, and Sipser [GS85] show that some simple sets may have hard ranking problems: if a certain set in P can be strongly P-ranked then #P functions can be computed by polynomial time

[1] Given a language A and a string x, one can answer "$x \in A$?" if one can compute the ranks if x and $x-1$.

[2] Dr. Osamu Watanabe introduced the author to this view of ranking.

machines. We prove the converse of this result and thus completely characterize when P is strongly P-rankable, and we study some novel facets of the ranking problem—ranking with circuits, the complexity of ranking circuits, etc.

Definition 4.1

1. For any set $A \subseteq \Sigma^*$, the *strong rank function* f_A is defined for all $x \in \Sigma^*$ by:

$$f_A(x) = |A^{\leq x}|,$$

   where $A^{\leq x}$ indicates all strings in A that are lexicographically $\leq x$. A set A is *strongly P-rankable* if its strong rank function f_A is polynomial time computable. In this case, we call f_A a strong P-ranker. A strong ranking function tells us the number of elements in A up to a given string.

2. For any set $A \subseteq \Sigma^*$, the *rank function* g_A is defined by:

   (a) $(\forall x \in A)[g_A(x) = |A^{\leq x}|]$, and

   (b) $(\forall x \notin A)[g_A(x)$ prints "not in A"$]$.

   A set A is *P-rankable* if its rank function g_A is polynomial time computable. A P-ranking function ranks elements that are in the set but merely detects the nonmembership of those outside the set.

3. A function h is a *weak rank function* of set A if

$$(\forall x \in A)[h(x) = |A^{\leq x}|].$$

   A set is *weakly P-rankable* if it has a weak rank function that is polynomial time computable. A weak P-ranking function may (deviously) output false values on inputs not in A.

4. We use *P-rankable*, *weak-P-rankable*, and *strong-P-rankable* to denote the classes of languages having these properties.

5. We say a language class $\mathcal{C}$ is P-rankable if all languages in $\mathcal{C}$ are P-rankable, i.e., if $\mathcal{C} \subseteq$ P-rankable (similarly for weakly P-rankable and strongly P-rankable).

Section 4.3 proves our first ranking result, which

- characterizes the complexity of ranking in structural terms,

- shows that even weak types of ranking are hard unless complexity classes collapse, and
- shows that the P-ranking problems for P and NP are equally likely to be easy.

Theorem 4.3 The following are equivalent:

1. $\mathrm{P} = \mathrm{P}^{\#\mathrm{P}}$.
2. $\mathrm{P} = \mathrm{P}$-*rankable.*
3. $\mathrm{P} =$ *strong-*P*-rankable.*
4. $\mathrm{P} \subseteq$ *weak-*P*-rankable.*
5. $\mathrm{P} \subsetneq$ *weak-*P*-rankable.*
6. $\mathrm{NP} = \mathrm{P}$-*rankable.*
7. $\mathrm{NP} =$ *strong-*P*-rankable.*
8. $\mathrm{NP} \subseteq$ *weak-*P*-rankable.*
9. $\mathrm{NP} \subsetneq$ *weak-*P*-rankable.*
10. $\mathrm{PH} = \mathrm{P}$-*rankable.*

Corollary 4.5 P is P-rankable if and only if NP is P-rankable.

Steven Rudich has, independently, proven the equivalence of (1) and (2) [Rud87].

Given our belief that sets in P are hard to rank, it is natural to ask whether sets in P have *small ranking circuits.* That is, does each set A in P have a family of circuits c_i, so (1) $|c_i|$ grows polynomially in i and (2) on any x of length i, c_i computes the rank of x in A? (Here each c_i will be a circuit of logical gates [Sav72], with enough output wires to carry any possible binary value of the rank.)

Section 4.4 shows that if P has small ranking circuits then the power of counting is dominated by the polynomial hierarchy:[3]

Theorem 4.11 If P has small ranking circuits, then $\mathrm{P}^{\#\mathrm{P}} \subseteq \Sigma_2^p = \mathrm{PH}$.

Our proof has two steps.

[3]The levels of the polynomial hierarchy (Stockmeyer [Sto77]) are defined in Section 2.4.

1. We show that if a set has small (but possibly complex) ranking circuits, then it has small ranking circuits that are relatively easily computed, i.e., they are printed by a Δ_4^p machine.[4] This technique of pulling circuits and sets down into the polynomial hierarchy has been mined by Hartmanis and by Balcazar, Book, Long, Selman, and Schöning [BBL*84] in their work on sparse oracles. Our application of their technique is surprising. Given our assumption of small circuits, we show that a boolean predicate with shallow quantifier nesting can verify that a given circuit is a valid ranking circuit for a given set in P.

2. We combine our result with the useful small circuit theorem of Karp and Lipton [KL80] to obtain Theorem 4.11.

Section 4.5 turns from studying the ranking problem of uniform classes such as P and NP to studying the ranking problem of the nonuniform complexity class P/poly (see [Sch86] for a discussion of results about P/poly). P/poly, the class of sets with small circuits, is an important class. Sets in P/poly are near P in the sense that with a sparse amount of additional information for each input length a P machine can accept any given P/poly set. The central result on P/poly is that if P/poly$\supseteq$NP, then the polynomial hierarchy collapses to its second level [KL80].

Some classes that we discuss, in particular P/poly, contain a nondenumerable number of sets; thus many standard techniques no longer apply. For example, these classes lack many-one complete languages. Thus we must modify the proof techniques of previous sections to eliminate their reliance on complete languages.[5]

By simple counting we show that P/poly is not P-rankable. As a nonuniform analogue of the results of Section 4.3, we show that P/poly has small ranking circuits if and only if P/poly$=$P$^{\#\mathrm{P}}$/poly. A connection between the rankability

[4] As a corollary of Theorem 4.11, we'll see that the circuits are much simpler than this (Corollary 4.12).

[5] A study of other classes that are suspected (for different reasons) not to have many-one complete languages and of the techniques that apply to these classes appears in recent papers of Hartmanis and Immerman [HI85] and Hartmanis and Hemachandra [HH86a]. They note that many probabilistic, intersection, and counting classes may not have complete languages (and do not, in some relativized worlds), and that if these classes do have complete languages they are of a particularly simple form.

of P and P/poly follows from this. If the uniform class P can be uniformly ranked (i.e., P-ranked), then the nonuniform class P/poly can be nonuniformly ranked.

Theorem 4.16 P/poly has small ranking circuits $\Leftrightarrow$ P/poly $=$ P$^{\#P}$/poly.

Corollary 4.17 If P is P-rankable then P/poly has small ranking circuits.

Finally, having shown in the previous sections that even weak or nonuniform ranking of P causes unlikely collapses of complexity classes, we naturally ask whether it is easy to approximate the rank functions of sets in P. This is an open question posed by Goldberg and Sipser [GS85].

It follows directly from the fact that the ranking functions of P sets belong to #P (see "(1)$\Rightarrow$(7)" of the proof of Theorem 4.3, page 62) and from Stockmeyer's result that approximate counting can be done in Δ_3^p, that Δ_3^p machines can compute "approximate ranks" for P sets. The notion here is that the "approximate rank" will be related to the true rank by a multiplicative factor that is around $1+O(n^{-d})$, where d is arbitrary but fixed.

Section 4.6 studies a different notion of approximate ranking, which we call *enumerative ranking.* Cai and Hemachandra [CH86b] introduce a notion of enumerative counting in which they ask whether one can quickly compute a small set of candidate solutions to a #P problem, one of which is guaranteed to be correct. They show that notion of their enumerative counting is Turing equivalent to exact counting. Thus this type of enumerative counting is as hard as exact counting.

Let's adopt a similar approach to approximation of ranks.

Definition 4.2 We say a set A is *k-enumeratively-rankable* if there is a polynomial time computable function f so for every x, $f(x)$ prints a set of k numbers, one of which is the rank of x with respect to A.

Similarly we can define *$g(n)$-enumeratively-rankable.* As a corollary of the above-mentioned result of [CH86b], we show that enumerative ranking is impossible unless P $=$ P$^{\#P}$.

Theorem 4.18 P $=$ P$^{\#P}$ if and only if for each set $A \in$ P there exists some positive integer k such that A is k-enumerative-rankable.

With some care, we can show that P $=$ P$^{\#P}$ if and only if each set A in P for some ϵ_A is $n^{1/2-\epsilon_A}$-enumeratively-rankable, where n is the input length.

Section 4.7 summarizes the above results and presents related open problems.

This chapter structurally characterizes the complexity of uniform, weak, nonuniform, and enumerative ranking. It follows from the characterizations that if our intuition about the inequality of certain complexity classes (such as P and $P^{\#P}$) is correct, then ranking lies beyond the grasp of polynomial time.

4.3 When Can Uniform Complexity Classes be Ranked?

This section answers the questions:

- Is NP harder to rank than P?
- When can P be ranked?
- When can NP be ranked?
- When can PSPACE be ranked?

Theorem 4.3 The following are equivalent:

1. $\mathrm{P} = \mathrm{P}^{\#\mathrm{P}}$.
2. $\mathrm{P} =$ P-*rankable.*
3. $\mathrm{P} =$ *strong*-P-*rankable.*
4. $\mathrm{P} \subseteq$ *weak*-P-*rankable.*
5. $\mathrm{P} \subsetneq$ *weak*-P-*rankable.*
6. $\mathrm{NP} =$ P-*rankable.*
7. $\mathrm{NP} =$ *strong*-P-*rankable.*
8. $\mathrm{NP} \subseteq$ *weak*-P-*rankable.*
9. $\mathrm{NP} \subsetneq$ *weak*-P-*rankable.*
10. $\mathrm{PH} =$ P-*rankable.*

Blum, Goldberg, and Sipser [GS85] show that if a certain P set is strongly P-rankable then all #P functions can be computed in polynomial time. Their method is to construct a set[6] combining formulas and their assignments, so that

[6]Loosely, this set is $\{f\#\mathit{assign} \mid |\mathit{assign}| = |f| \text{ and } |\mathit{assign}| \text{ is a vector of assignments to all variables of } f \text{ that satisfies } f\}$.

all the variable assignments to a formula are lexicographically adjacent. Thus computing a #P question reduces to the problem of subtracting two ranks, i.e., computing how many variable assignments are satisfying (and thus in the set).

This method fails for weak P-rankers; the extreme assignments may be out of the sets, and thus the ranker's answers about them may be lies. We modify of the argument of Blum, Goldberg, and Sipser to show that if all P sets are even weakly P-rankable, then $\mathrm{P}=\mathrm{P}^{\#\mathrm{P}}$.

A weak P-ranker may give false ranks for elements out of the set, but is correct for elements in the set. We take advantage of the ranker's correctness on these values by inserting easily located *beacons* into the set. The weak P-ranker of course is compelled to be truthful on these beacons, and we force it to solve a #P query (this is done in the proof below that (4) $\Rightarrow$ (1)).

Going the other way, (1) $\Rightarrow$ (7) below shows how to strongly rank NP sets when $\mathrm{P}=\mathrm{P}^{\#\mathrm{P}}$. Simply put, when $\mathrm{P}=\mathrm{P}^{\#\mathrm{P}}$, we can bring down from the #P oracle, one bit at a time, the correct ranks of strings from our NP language.

Notation 4.4 $Rank_A(x) =_{def} |A^{\leq x}|$, i.e., the number of strings in A that are lexicographically $\leq x$.

Proof of Theorem 4.3 Obvious implications that we use are (3) $\Rightarrow$ (2) $\Rightarrow$ (4), (7) $\Rightarrow$ (6) $\Rightarrow$ (8), (8) $\Rightarrow$ (4), (10) $\Rightarrow$ (2), (1) & (2) $\Rightarrow$ (10), and (7) $\Rightarrow$ (3). We prove that (4) $\Rightarrow$ (1), (1) $\Rightarrow$ (7), (4) $\equiv$ (5), and (8) $\equiv$ (9). Thus (1) $\Rightarrow$ (7) $\Rightarrow$ (6) $\Rightarrow$ (8)[$\equiv$(9)] $\Rightarrow$ (4)[$\equiv$5] $\Rightarrow$ (1), (1) $\Rightarrow$ (7) $\Rightarrow$ (3) $\Rightarrow$ (2) $\Rightarrow$ (4) $\Rightarrow$ (1), and (10) $\Rightarrow$ (2) $\Rightarrow$ (2) & (1) $\Rightarrow$ (10). Thus (1) through (10) are all equivalent.

[(4) $\Rightarrow$ (1)]:

To show that $\mathrm{P}=\mathrm{P}^{\#\mathrm{P}}$ it suffices to show that #SAT (i.e., computing the number of solutions to a boolean formula F) can be done by a polynomial time machine. Let

$$L = Satisfiers \cup Beacons,$$

where $Satisfiers = \{F\#01\#z \mid |z| = 3|F| \text{ and } z \text{ is a vector that assigns all variables of boolean formula } F \text{ (followed by padding ones) and } F \text{ is satisfied by the assignment coded by } z\}$ and $Beacons = \{y\#00\#1^{3|y|} \mid y \in \Sigma^*\} \cup \{y\#11\#1^{3|y|} \mid y \in \Sigma^*\}$. Clearly $L \in \mathrm{P}$ and by hypothesis (4), L has a weak P-ranker.

Asked the #SAT question about boolean formula F, we use our weak P-ranker to compute in polynomial time the integer:

$$\left(\mathit{Rank}_L(F\#11\#1^{3|F|}) - \mathit{Rank}_L(F\#00\#1^{3|F|}) \right) - 1. \tag{4.3.1}$$

This equals the number of solutions of F, as the beacons are in L and get truthful responses from the weak P-ranker. A difference of one between the above ranks is due to the beacons' rank difference, and any additional difference comes from satisfying assignments to F that are lexicographically sandwiched between the beacons. Thus with a weak P-ranker for L we compute #SAT in polynomial time, so $\mathrm{P} = \mathrm{P}^{\#\mathrm{P}}$.

[(1)$\Rightarrow$(7)]:

Clearly every strongly P-rankable language is in P, and hence *a fortiori* in NP. Assuming $\mathrm{P} = \mathrm{P}^{\#\mathrm{P}}$, we must show that an arbitrary language $L \in \mathrm{NP}$ can be strongly P-ranked. Since $\mathrm{NP} \subseteq \mathrm{P}^{\#\mathrm{P}}$, hypothesis (1) implies NP = P, and so L is accepted by some polynomial time machine M. Then there is an NP machine N such that, on any input x, N nondeterministically prints a string y as a "guess" on its tape, subject to the restriction that y is lexicographically $\leq x$. N then simulates M on input y deterministically, and accepts if and only if $M(y)$ accepts. Since N is an NP machine, its counting function c_N ($c_N(x) =$ the number of accepting paths of $N(x)$) is in #P. Clearly, by our choice of N, we have $\mathit{Rank}_L(x) = c_N(x)$. Thus the language $L_{bits} = \{x\#i\#b \mid c_N(x) \text{ has } \geq i \text{ bits and the } i\text{th bit is } b\}$ is in P. By repeatedly using L_{bits} to get one more bit of $c_N(x)$, we can compute in polynomial time $c_N(x) = \mathit{Rank}_L(x)$, and thus we have strongly P-ranked the language L.

[(4)$\equiv$(5)], [(8)$\equiv$(9)]:

That (5) implies (4) and (9) implies (8) follow immediately.

Every language having exactly one string of each length $n \geq 0$ is weakly P-rankable by the function $x \mapsto |x|$, which is polynomial time computable. Since there clearly are nonrecursive languages L with this property (via diagonalization or Kolmogorov complexity), we have (4)$\Rightarrow$(5) and (8)$\Rightarrow$(9). □

Aside from showing that ranking is complex, Theorem 4.3 reveals a striking contrast between the determinism-vs-nondeterminism questions for membership and counting. We believe that $\mathrm{P} \neq \mathrm{NP}$ and thus the membership problems of NP and P are of *different complexities*, respectively hard and easy. Corollary 4.5 says that either the ranking problems of P and NP are both easy, or are both hard.

Corollary 4.5 P is P-rankable if and only if NP is P-rankable.

Though at first surprising, this is not hard to understand. The ranking problems of P and NP are so complex that if either is easy it causes a major collapse of complexity classes, which in turn destroys any difference between P and NP.

Only a powerful complexity class can avoid being pulled into polynomial time if $\mathrm{P} = \mathrm{P}^{\#\mathrm{P}}$. PSPACE may be such a class.

Theorem 4.6 PSPACE is P-rankable if and only if P = PSPACE [Hem87a].

It is possible that the ranking problem of PSPACE may be hard even if the ranking problems of P and NP are easy. However, no oracle is known to certify this possibility (i.e., for no known oracle A is $\mathrm{P}^A = \mathrm{P}^{\#\mathrm{P}^A} \neq \mathrm{PSPACE}^A$), and any such oracle would resolve the major open relativization question about the polynomial hierarchy left in the aftermath of Yao's breakthrough [Yao85]: Is there an oracle for which the polynomial hierarchy *collapses* yet differs from PSPACE?

4.4 Small Ranking Circuits and P

In this section, we answer the questions:

- What happens if P has small strong-ranking circuits?
- What happens if P has small ranking circuits?
- What occurs if P has small weak-ranking circuits?

Having small circuits for a problem is the next best thing to having a polynomial time algorithm. If a problem has small circuits, there are relatively small machines that solve the problem for all non-enormous instances.

The previous section gives strong evidence that not all languages $L \in \mathrm{P}$ can be ranked with P machines. A natural weaker goal is to find small circuits for the ranking problem of L. For example, the set of primes has small circuits ([Rab76, Adl78], see [Sch86, pp. 33,40]), but is not known to belong to P [APL80,GK86].

Again, we prove that ranking is probably hard. If all sets in P can be ranked with small circuits, then the power of counting falls into the polynomial hierarchy.

We first show that if P has small strong-ranking circuits, then $\mathrm{P}^{\#\mathrm{P}} \subseteq \Sigma_2^p = \mathrm{PH}$. Then we sketch how to modify the proof to obtain the more powerful result that if P has small ranking circuits, then $\mathrm{P}^{\#\mathrm{P}} \subseteq \Sigma_2^p = \mathrm{PH}$.

Indeed, we can show that if P has small weak-ranking circuits, the same conclusion holds: $\mathrm{P}^{\#\mathrm{P}} \subseteq \Sigma_2^p = \mathrm{PH}$.

4.4.1 If P has Small Strong-Ranking Circuits then $\mathrm{P}^{\#\mathrm{P}} \subseteq \Sigma_2^{\mathrm{P}}$

The crucial step in our proof is to show that if P has small strong-ranking circuits (possibly complex ones), then it has small strong-ranking circuits that are relatively simple, i.e., printable by a Δ_3^p machine.

Balcazar, Book, Long, Schöning, and Selman [BBL*84] and Hartmanis use this paradigm of pulling down complexities in their work on sparse oracles. Our application to ranking of "pulling complexities down into the polynomial hierarchy" is a bit subtle. Our goal is to show that if small strong-ranking circuits exist then we can *compute* them using only a few polynomially bounded quantifiers.

Simply put, we guess a small circuit and make sure that the function that it computes:

1. starts at zero,

2. increases by one at each (small enough) string in L, and

3. stays the same at each (small enough) string not in L.

If so, the circuit is a small strong-ranking circuit.

This is done in Lemma 4.8 below. Theorem 4.9 follows after an appeal to a result of Karp and Lipton on the consequences of NP having small circuits.

Definition 4.7 We say a set L has *small strong-ranking circuits* if there is a family $\{c_i\}$ of circuits such that:

1. $|c_i|$ is polynomial in i, and
2. for all x with $|x| \leq i$, $c_i(x)$ outputs $Rank_L(x)$.

Comments on Definition 4.7:

- When we say $c_i(x)$, we of course assume that x is prefixed by enough leading special padding characters (#) to make its length exactly i. (Equivalently our circuits could have two wires for each input, one to say if there was a digit in that place and another to give the digit's value. Alternatively, we could use $\log i$ extra inputs to specify the length.)

- Typical definitions of a family of small circuits [Sch86] have circuit c_i perform correctly on all length i strings. Our definition (which will shorten the proofs) says it performs correctly on all length $\leq i$ strings. It is clear that a set L has small strong-ranking circuits under the first definition if and only if it has small strong-ranking circuits under the second definition (one direction is trivial, in the other direction our circuit first determines the length of the (unpadded) input and uses the appropriate circuit).

Lemma 4.8 If P has small strong-ranking circuits, then every P set has a family of small strong-ranking circuits that is printable by a $\Delta_3^p (= \mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}})$ machine.

Theorem 4.9 If P has small strong-ranking circuits, then $\mathrm{P}^{\#\mathrm{P}} \subseteq \Sigma_2^p = \mathrm{PH}$.

Proof of Lemma 4.8 Let $L \in \mathrm{P}$ have small strong-ranking circuits. W.l.o.g. suppose that $|c_i| \leq i^k$ for some fixed k and all i.

Note that it suffices to show that the bottom NP machine of an $\mathrm{NP}^{\mathrm{NP}}$ machine can find (if one exists) a small strong-ranking circuit consistent with some prefix, for L up to a given length. This is because then a $\mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}} (= \Delta_3^p)$ machine with this $\mathrm{NP}^{\mathrm{NP}}$ oracle can print a small circuit. It does so by getting one bit of the circuit at a time from the oracle with queries of the form, "Is there a strong-ranking circuit for L up to length i whose representation starts with bits 001101?"

More rigorously, let k be the degree of the polynomial growth of the small circuits. Let $L_{circ} = \{1^i\#\sigma \mid$ There exists a circuit representation c so that $|c| \leq i^k$ and c has prefix σ and c is a strong-ranking circuit for L up to length $i\}$.

Here is the $\mathrm{NP}^{\mathrm{NP}}$ procedure that accepts L_{circ}. x_- denotes the string that lexicographically precedes x and ϵ denotes the empty string. Let σ be the prefix we are extending and $c_i \sqsupseteq \sigma$ indicate that (the representation of circuit) c_i starts with the bits σ. Recall that the expression below simply says we start at zero, increase by one at elements of L, and stay the same at nonmembers of L. $x \in L_{circ} \Leftrightarrow$

$$\Big(\exists c_i . |c_i| \leq i^k\Big)\Big(c_i \sqsupseteq \sigma \wedge [(\epsilon \in L \wedge c_i(\epsilon) = 1) \vee (\epsilon \notin L \wedge c_i(\epsilon) = 0)] \wedge$$
$$(\forall x . |x| \leq i)\Big[(x \in L \wedge c_i(x) - c_i(x_-) = 1) \vee (x \notin L \wedge c_i(x) - c_i(x_-) = 0)\Big]\Big).$$

This depth-two-polynomially-bounded-quantifier language, L_{circ}, can be accepted by an $\mathrm{NP}^{\mathrm{NP}}$ machine (the base machine does the $\exists$ and the NP oracle does the $\forall$, see Stockmeyer [Sto77]). This, as noted at the start of this proof, suffices to prove our lemma. $\square$

Proof of Theorem 4.9 By Lemma 4.8, every P set has small strong-ranking circuits computable in Δ_3^p. Thus we can solve #SAT, as in the proof of Theorem 4.3, with two calls to this Δ_3^p-computable object, and thus place $\mathrm{P}^{\#\mathrm{P}} \subseteq \mathrm{P}^{\Delta_3^p} = \Delta_3^p$. But it is easy to see that if P has small strong-ranking circuits then NP has small membership circuits (via the language L_B of page 63). So by the Karp-Lipton result [KL80], $\Sigma_2^p = \Pi_2^p = \mathrm{PH}$. Thus $\mathrm{P}^{\#\mathrm{P}} \subseteq \Delta_3^p \subseteq \mathrm{PH} \subseteq \Sigma_2^p$. $\square$

4.4.2 If P has Small Weak-Ranking Circuits or Small Ranking Circuits then $\mathrm{P}^{\#\mathrm{P}} \subseteq \Sigma_2^{\mathrm{P}}$

This follows the same argument as the previous section, except we must account for the change from strong-ranking circuits to ranking circuits. The potential problem is that a ranking circuit, on input $x \notin L$, merely says "this is not in L" (unlike a strong-ranker, which ranks nonmembers).

Given that a collection of small ranking circuits exists, we guess a small circuit and make sure that the function it computes meets conditions which guarantee that it really is a ranking circuit of L, as argued before in the proof of Lemma 4.8.

Simply put, our conditions are that the function starts at zero, is correct about membership and nonmembership, and increases its rank exactly one between adjacent members of L. These conditions bear contrast with the simpler conditions used to check strong-ranking circuits on page 66.

Spelled out in full, our conditions on the function f that the guessed circuit computes are that f:

1. starts at zero,
2. at each (small enough) string out of L function f proclaims nonmembership,
3. at each (small enough) string in L function f proclaims membership by announcing (its version of) the string's rank.
4. for every pair of (small enough) strings x, y (w.l.o.g. y is lexicographically greater than x), such that:
   (a) f thinks that x and y are in L and thus gives them ranks, and
   (b) there exists no string z lexicographically between x and y in L,

   the rank f assigns y is exactly one greater than the rank f assigns x.

Looking at the quantifier structure implicit in the above conditions, they can be written as a Σ_3^p predicate; so we can *print* a small ranking circuit with a Δ_4^p machine. Theorem 4.11 now follows as in the previous section.

Interestingly, as a corollary of the theorem, we conclude that the circuits can be printed by a Δ_3^p machine.[7]

Lemma 4.10 If P has small ranking circuits, then every P set has a family of small ranking circuits that is printable by a Δ_4^p machine.

Theorem 4.11 If P has small ranking circuits, then $\mathrm{P}^{\#\mathrm{P}} \subseteq \Sigma_2^p = \mathrm{PH}$.

Corollary 4.12 If P has small ranking circuits, then each P set has a family of small ranking circuits that is printable by a Δ_3^p machine.

[7]Even though we conclude that all *languages* in the polynomial hierarchy are in Σ_2^p, some *functions* computable by polynomial hierarchy machines may need Δ_3^p machines, which use Σ_2^p oracles to get longer and longer prefixes of the functions' outputs.

By slightly modifying the four conditions above[8] we can strengthen our results to apply to small *weak*-ranking circuits.

Theorem 4.13 If P has small weak-ranking circuits, then $P^{\#P} \subseteq \Sigma_2^p = PH$.

4.5 Ranking P/Poly

This section answers the questions:

- Can P/poly be P-ranked?
- When can P/poly have small ranking circuits?
- What techniques can be used in proofs about uncountable classes, which necessarily lack polynomial many-one complete sets?

The classes P/poly and $P^{\#P}$/poly contain uncountably many languages. Because of this, they are immune to many techniques we use on standard complexity classes such as P and NP. In particular, they have no many-one complete languages, at least with respect to uniform many-one reductions, so the technique of Theorem 4.3, which used the fact that rendering #SAT polynomial time computable makes all of #P polynomial time computable, does not apply.

We prove an analogue of Theorem 4.3 *without* using any complete language. Thus complete sets, usually a crucial tool, seem to have been thrown aside. However, when we look carefully, we see what has happened. Before, our proof centered on a complete set and the set dragged its class with it. Now, our proofs must take an arbitrary member of the nondenumerable class and operate on it. Thus instead of working with a simple set like #SAT, we work directly with an arbitrary $P^{\#P}$/poly set by formalizing in the standard way [Coo71][Har78] the computations of its underlying machine.

Also, note that by simple counting, P/poly is not P-rankable; this is because P/poly contains sets spanning an uncountable number of ranking functions, and there are only a countable number of P-rankers.

[8]**Eliminate condition 2. Change condition 1 so that it checks that the first string in the set is indeed assigned rank one.**

Theorem 4.14 P/poly is not P-rankable.

Definition 4.15 A language L is in $\mathrm{P}^{\#\mathrm{P}}$/poly if there is a polynomial time Turing machine M, a #P function f, and an advice function g such that:

1. $(\exists k)(\forall x)[|g(x)| \leq |x|^k + k]$, and
2. $(\forall x)[M^f(x\#g(1^{|x|}))$ accepts if and only if $x \in L]$.

Put another way, L is in $\mathrm{P}^{\#\mathrm{P}}$/poly if a $\mathrm{P}^{\#\mathrm{P}}$ machine, given a bit of free advice that depends only on the length of the input, can compute L. It is easy to prove that

$$\mathrm{P}^{\#\mathrm{P}}/\mathrm{poly} = \mathrm{P}^{\#\mathrm{P}/\mathrm{poly}},$$

where the #P/poly oracle counts the accepting paths of NP machines that are given a bit of advice.

Theorem 4.16 P/poly has small ranking circuits if and only if

$$\mathrm{P}/\mathrm{poly} = \mathrm{P}^{\#\mathrm{P}}/\mathrm{poly} = \mathrm{P}^{\#\mathrm{P}/\mathrm{poly}}.$$

Remark: In this proof we revert to the standard notion of families of circuits accepting languages, rather than computing functions as in the previous proof, and we use the standard definition of circuits—each circuit accepts all strings of the set of some single length.

Proof of Theorem 4.16

$\Longleftarrow$: Given $L \in \mathrm{P}/\mathrm{poly}$, we know that L has small circuits. We wish to show that L has small *ranking* circuits.

Let N be an NP machine that, on any input of the form $x\#c_0\#\ldots\#c_{|x|}$, makes a nondeterministic path for each string y lexicographically $\leq x$. The path associated with y will accept if and only if the circuit $c_{|y|}$ accepts y. Clearly, if the circuits given to N are correct circuits for L, then $Rank_L(x)$ equals the number of accepting paths of $N(x)$.

A $\mathrm{P}^{\#\mathrm{P}}$/poly machine whose #P oracle is the counting function of N and whose advice function is the circuit family of L can compute the ranking function, and in particular, can accept the following languages L_k, for each fixed $0 \leq k \leq |x|$.

$$L_k = \{x\#b \mid \text{ the } k\text{'th bit of } Rank_L(x) \text{ is } b\}.$$

By assumption, $\mathrm{P}^{\#\mathrm{P}}/\mathrm{poly} = \mathrm{P}/\mathrm{poly}$, so we have small circuits for $L_1, L_2, \ldots, L_{|x|}$, i.e., each circuit computes one bit of the ranking function. We can easily fan out the inputs and combine these circuits, thus obtaining a small ranking circuit for our arbitrary P/poly language L.

$\Longrightarrow$: Let L be an arbitrary language in $\mathrm{P}^{\#\mathrm{P}}/\mathrm{poly}$. We show that, if P/poly has small ranking circuits, then L is in P/poly. W.l.o.g., let the #P oracle of the $\mathrm{P}^{\#\mathrm{P}}/\mathrm{poly}$ machine accepting L be the counting function (computing the number of accepting paths) of the machine N. Since N is an NP machine, we may suppose that N runs in nondeterministic time $\mathrm{O}(n^l)$ for some fixed integer l. Define:

$$\begin{aligned} L' \;=\; & \Big\{x\#01\#path\#1^{pad} \;\Big|\; |path\#1^{pad}| = |x|^{l+1} \text{ and } N(x) \text{ has } path \\ & \qquad \text{as an accepting path}\Big\} \cup \\ & \Big\{x\#00\#1^{|x|^{l+1}} \mid x \in \Sigma^*\Big\} \cup \\ & \Big\{x\#11\#1^{|x|^{l+1}} \mid x \in \Sigma^*\Big\}. \end{aligned}$$

Then $L' \in \mathrm{P}$, so from our assumption L' has small ranking circuits.

Here is how to accept L with a P/poly machine. Our advice will consist of the advice given to L by Definition 4.15 plus the small ranking circuits of L'. On input x, the P/poly machine simulates the run of the $\mathrm{P}^{\#\mathrm{P}}/\mathrm{poly}$ machine accepting L (recall, we have its advice function) up to its first call to its #P oracle; say the call is y. We know that the answer will be (see equation 4.3.1):

$$\{Rank_{L'}(y\#11\#1^{|y|^{k+1}}) - Rank_{L'}(y\#00\#1^{|y|^{k+1}})\} - 1.$$

But from our advice function, *we have small circuits computing the rank function of L'*, so our P/poly machine can easily do the above subtraction, and thus answer the first query made in the run of $\mathrm{P}^{\#\mathrm{P}}/\mathrm{poly}$.

Our P/poly machine continues to simulate the $\mathrm{P}^{\#\mathrm{P}}/\mathrm{poly}$ machine accepting L, answering later queries in the same way. Thus we have accepted an arbitrary language $L \in \mathrm{P}^{\#\mathrm{P}}/\mathrm{poly}$ via a P/poly machine. □

It is easy to see that

$$\mathrm{P} = \mathrm{P}^{\#\mathrm{P}} \quad \Longrightarrow \quad \mathrm{P}/\mathrm{poly} = \mathrm{P}^{\#\mathrm{P}}/\mathrm{poly}.$$

From Theorems 4.3 and 4.16, it follows that if uniform (P) sets can be uniformly ranked, then nonuniform sets (P/poly) can be nonuniformly ranked. Thus uniform and nonuniform ranking problems are structurally related. Yap [Yap83] has also related uniform and nonuniform membership and hierarchy questions.

Corollary 4.17 If P is P-rankable, then P/poly has small ranking circuits.

4.6 Enumerative Ranking

In this section we answer the question:

- Do all sets in P have enumerative rankers?

The number of possible ranks a given string x can have, expressed in terms of its length, is on the order of $2^{|x|}$. An enumerative ranker (recall Definition 4.2) reduces this huge set of potential ranks to a small set of plausible ranks. It follows that if all P sets have enumerative rankers, then the ranks of strings in P sets are of low relative Kolmogorov complexity. The *small name* of a rank, relative to the string it ranks, is its rank (index) in the small set of plausible ranks.

This enumerative approach to approximation is modeled after the work of Cai and Hemachandra [CH86b] on enumerative counting, and our results follow as corollaries of their main theorem: Suppose there exists an ϵ, $0<\epsilon<1$, and a polynomial time machine M such that for every boolean formula f, $M(f)$ outputs a set of m values, where $m \leq |f|^{1-\epsilon}$, of which one of the values is the number of satisfying assignments to f. Then $\mathrm{P} = \mathrm{P}^{\#\mathrm{P}}$.

Our key observation is that if we have a k-enumerative-ranker (Definition 4.2), we can k^2-enumerate #SAT. Why? With a k-enumerative-ranker, when we do the subtraction of ranks that yields our #SAT answer (as in the proof of Theorem 4.3), we now have k possibilities for each of the two participating ranks in the subtraction, so we have k^2 overall possibilities. The following theorem follows from this observation, the result of [CH86b], and an easy converse direction that is a corollary of Theorem 4.3. Theorem 4.18 below says that it is unlikely that we can enumeratively rank all sets in P. Indeed, enumerative ranking is so hard that it is no more likely that we can enumeratively rank them in polynomial time than that we can exactly rank them in polynomial time.

Theorem 4.18 $P = P^{\#P}$ if and only if each set $A \in P$ has a k-enumerative-ranker for some k.

Indeed, since (by the observation above) we need only square the size of the enumerative list for ranking to get the size of the enumerative list for #SAT, and since the [CH86b] result is valid up to $O(n^{1-\epsilon})$-enumerative-counting, we can even conclude that:

Theorem 4.19 $P = P^{\#P}$ if and only if for each set $A \in P$ there exists an $\epsilon_A > 0$ such that A has an $O(n^{1/2-\epsilon_A})$-enumerative-ranker.

4.7 Conclusions and Open Problems

This chapter has studied the complexity of the ranking problem and has characterized the complexity of many types of ranking. Now, with the same the certainty with which we believe counting is hard, and thus with at least the certainly with which we believe $P \neq NP$, we believe that uniform, strong, weak, and enumerative ranking cannot be done efficiently for all sets in P. Indeed, if any of these types of ranking can be done efficiently then deterministic polynomial time equals probabilistic polynomial time (since $P = P^{\#P}$ if and only if $P = PP$ [Gil77,Sim75]).

Many related open questions remain. Section 4.3, page 65, asks if there is a world in which P and NP have easy ranking problems but PSPACE has hard ranking problems. That is, is there a relativized world in which $P = P^{\#P} \neq PSPACE$? We also would like to know if there is a world in which P and NP have easy membership problems but hard ranking problems. That is, is there are relativized world in which $P = NP \neq P^{\#P}$? Resolving either of these questions would settle the major open problem about the relativized structure of the polynomial hierarchy that remains in the wake of Yao's results [Yao85]: Is there a relativized world in which the polynomial hierarchy *collapses* yet is not equal to PSPACE?

Any advance beyond the work of [CH86b] in our knowledge of the complexity of enumerative counting would strengthen the results of Section 4.6. Can the

$O(n^{1-\epsilon})$ bound in the theorem of [CH86b] cited on page 73 be improved to n^k for all fixed k?

Chapter 5

Robustness

5.1 Chapter Overview

A robust machine is a machine that maintains a computational property for *every* oracle. For example, a pair of NP machines that accept complementary languages not only in the real world but in every relativized world are said to be robustly complementary.

In this chapter we study robustly complementary, robustly categorical, robustly Σ^*-accepting, and robustly Σ^*-spanning machines. We prove that robust machines squander their powerful nondeterministic oracle access in all relativizations. For example, a robustly complementary machine will accept, for every oracle A, a language in $\mathrm{P}^{\mathrm{NP}\oplus A}$, where $\oplus$ represents disjoint union.

5.2 Introduction

Informally, a robust property of a machine is a property that a machine has with every oracle. For example, if two machines accept complementary languages for every oracle $((\forall A)[L(N_1^A)=\overline{L(N_2^A)}])$ we say that the machines are ***robustly complementary***.

Schöning [Sch84] considers deterministic machines M that accept some language robustly (i.e., there is a language L so that for all oracles A, $L(M^A)=L$).

He shows that the machines of this sort that for some oracle A' run in polynomial time accept exactly the NP$\cap$coNP languages.

In this chapter, we ask what price a machine pays to have robust properties. Throughout the chapter, all robust machines (with names like N, N_1, N_2, ..., N_i) are assumed to be *nondeterministic polynomial time Turing machines.* We discuss *robustly categorical* machines (machines that for no oracle and no input have more than one accepting path), *robustly Σ^*-accepting* machines (machines that for every oracle accept all inputs), *robustly complementary* machines (pairs of machines that accept complementary languages for every oracle), and *robustly Σ^*-spanning* machines (sets of machines whose languages union to Σ^* for every oracle). In each case we show, if P equals NP, that a machine having a property robustly is emasculated. That is, if P equals NP we conclude that, in some cases for all oracles A and in some cases for all sparse oracles A,

- robustly categorical machines accept only trivial (P^A) languages.
- robustly complementary machines accept only trivial (P^A) languages.
- robustly Σ^*-accepting machines have feasibly computable functions that determine *why* they accept.
- robustly Σ^*-spanning machines have feasibly computable selector functions. (Selman [Sel79,Sel82] defines a P-selector function over a set of NP machines to be a polynomial time function of x that, if at least one of the machines accepts on input x, outputs one of the machines that does accept.)

Another way of looking at our robustness results is as a study of the complexity of pulling NP from P. If P = NP, we are interested in knowing how oracles can pull NP from P. In general, even if P = NP, there will be oracles A that separate NP^A from P^A. Indeed, Hartmanis and Hemachandra completely characterize, in terms of Kolmogorov complexity, the sparse oracles that pull NP^S from P^S when P = NP [HH86b].

However, the NP^A machines that accept languages in $\mathrm{NP}^A - \mathrm{P}^A$ are wild machines—they make extensive use of their oracles. Our robustness theorems (e.g., Corollary 5.4) say that if P equals NP, no robust machine (say, no machine that has a machine that is robustly complementary to it) will accept a language in $\mathrm{NP}-\mathrm{P}$.

Yet another way of looking at these results is as "machine-based" lowness results. A set A is *extended low* [Sch83] if

$$\mathrm{NP}^{A} \subseteq \mathrm{P}^{\mathrm{NP}\oplus A}.$$

This says we can get by with a surprisingly weak form of access to A; though NP^{A} might access A exponentially often, $\mathrm{P}^{\mathrm{NP}\oplus A}$ touches A at most polynomially often and has a far weaker acceptance mechanism. Our robustness results (e.g., Theorem 5.1) say that any *language* accepted by a robust (e.g., robustly categorical) machine can be accepted with exactly such weak access to A.

For example, the following theorem shows that if nondeterministic machine N_i is robustly categorical, then for every oracle A the powerful nondeterministic access to the oracle N^{A} can make is squandered; a weak (polynomial time) machine with $\mathrm{NP}\oplus A$ as its oracle can accept the same language.

Theorem 5.1 $(\forall A)[N_i^A \text{ is categorical}] \Rightarrow (\forall A)[L(N_i^A) \in \mathrm{P}^{\mathrm{NP}\oplus A}]$.

We have mentioned general interpretations of our robustness theorems in terms of the difficulty of separating NP from P when $\mathrm{P}=\mathrm{NP}$, and in terms of lowness. Some of the results have special individual meaning, as they extend our knowledge about some intriguing facets of structural complexity theory.

One example of this is the study of *why* Σ^*-accepting NP machines accept. Borodin and Demers [BD76] show that $\mathrm{P} \neq \mathrm{NP} \cap \mathrm{coNP}$ implies there is an NP machine N so $L(N)=\Sigma^*$, yet there is no polynomial time machine that computes accepting paths of N. Simply put, they prove that if $\mathrm{P} \neq \mathrm{NP} \cap \mathrm{coNP}$ there is a machine that always accepts, but we cannot easily determine *why* it accepts.

Furthermore, even if $\mathrm{P}=\mathrm{NP}$, there will be many sparse oracles S and nondeterministic polynomial time machines N for which $L(N^S)=\Sigma^*$ yet we cannot easily find *why* $N^S(\cdot)$ accepts. Nonetheless, we show that if P equals NP and nondeterministic polynomial time Turing machine N accepts Σ^* for all sparse oracles S, then for all sparse oracles it will be *obvious* why N^S accepts. (This is true even for sparse oracles S for which $\mathrm{P}^S \neq \mathrm{NP}^S$.)

Corollary 5.8 If $\mathrm{P}=\mathrm{NP}$ and $L(N^T)=\Sigma^*$ for every sparse oracle T, then for every sparse set S there is a machine in P^S that on input x computes an accepting path of $N^S(x)$.

Simply put, we show that machines that maintain certain properties for all oracles accept relatively simple languages in every relativized world.

5.3 Robustness Theorems

This section gives a number of new robustness results. The proofs are outlined in Section 5.5. Simply put, machines pay a heavy price for maintaining robustness properties.

Below are a number of Theorem/Corollary pairs. The theorems emphasize a "lowness" approach: an NP^A machine satisfying a robustness property is repeatedly shown to be understandable via the far weaker access method of $\mathrm{P}^{\mathrm{NP}\oplus A}$. The corollaries emphasize that if $\mathrm{P}=\mathrm{NP}$, machines satisfying a robustness property cannot separate P^A from NP^A (i.e., do not accept languages in $\mathrm{NP}^A-\mathrm{P}^A$).

Theorem 5.1 (Robustly categorical machines accept simple languages) $(\forall A)[N_i^A \text{ is categorical}] \Rightarrow (\forall A)[L(N_i^A)\in \mathrm{P}^{\mathrm{NP}\oplus A}]$.

Corollary 5.2 If $\mathrm{P}=\mathrm{NP}$ and N_i is robustly categorical (i.e., $(\forall A)[N_i^A$ is categorical$]$), then for every oracle A,

$$L(N_i^A)\in \mathrm{P}^A.$$

Theorem 5.3 (Robustly complementary machines accept simple languages) $(\forall A)[L(N_i^A)=\overline{L(N_j^A)}] \Rightarrow (\forall A)[L(N_i^A)\in \mathrm{P}^{\mathrm{NP}\oplus A}]$.

Corollary 5.4 [1] If $\mathrm{P}=\mathrm{NP}$ and N_i and N_j are robustly complementary (i.e., $(\forall A)[L(N_i^A)=\overline{L(N_j^A)}]$), then for every oracle A

$$L(N_i^A)\in \mathrm{P}^A.$$

[1] Manuel Blum and Russell Impagliazzo have independently proved this result [Imp87].

We can restrict our attention to sparse[2] sets and get similar results (with easier proofs).

Theorem 5.5 (Machines robustly Σ^*-spanning on sparse oracles have simple selector functions)
$(\forall \text{ sparse } S)[L(N_{i_1}^S)\cup\cdots\cup L(N_{i_s}^S)=\Sigma^*] \Rightarrow (\forall \text{ sparse } S)(\exists f \text{ computable in } \mathrm{P}^{\mathrm{NP}\oplus S})$ $(\forall x)[x \in L(N_{i_{f(x)}})]$.

Corollary 5.6 If $\mathrm{P}=\mathrm{NP}$ and for every sparse oracle S, $L(N_{i_1}^S)\cup\cdots\cup L(N_{i_s}^S)=\Sigma^*$, then for every sparse oracle S there is a selector function f computable in P^S that for every input x selects one of the machines that indeed accepts. That is,

$$(\forall \text{ sparse } S)(\exists f \text{ computable in } \mathrm{P}^S)(\forall x)[x \in L(N_{i_{f(x)}}^S)].$$

Theorem 5.7 (Machines robustly Σ^*-accepting on sparse oracles accept for transparent reasons)
$(\forall \text{ sparse } S)[L(N_i^S)=\Sigma^*] \Rightarrow (\forall \text{ sparse } S)(\exists f \text{ computable in } \mathrm{P}^{\mathrm{NP}\oplus S})(\forall x)[f(x)$ prints an accepting path of $N_i^S(x)]$.

Corollary 5.8 If $\mathrm{P}=\mathrm{NP}$ and N_i robustly accepts Σ^* on sparse oracles (i.e., $(\forall \text{ sparse } T)[L(N_i^T)=\Sigma^*]$), then for every sparse oracle S, there is a function f computable in P^S so that on any input x

$$f(x) \textit{ prints an accepting path of } N_i^S(x).$$

Those who find the sight of "$\mathrm{P}=\mathrm{NP}$" unsettling will be pleased to know that we can trade off strength of structural assumptions for strength of robustness properties as shown below.

Theorem 5.9 (Machines robustly complementary and categorical on sparse oracles accept simple languages)
$(\forall \text{ sparse } S)[N_i^S$ and N_j^S are categorical and complementary$] \Rightarrow (\forall \text{ sparse } S)$ $[L(N_i^S) \in \mathrm{P}^{(\mathrm{UP}\cap\mathrm{coUP})\oplus S}]$.

[2]A set S is *sparse* if for some k, there are at every length n at most n^k+k strings in S.

Corollary 5.10 If $\mathrm{P}=\mathrm{UP}\cap\mathrm{coUP}$ and N_i^S and N_j^S are categorical and complementary for all sparse oracles S, then for all sparse oracles S,

$$L(N_i^S)\in \mathrm{P}^S.$$

A final point is that *all* of the above results hold *uniformly*. Taking Corollary 5.2 as an example, not only is each $L(N_i^A)$ in P^A, but there is a *single* polynomial time Turing machine that works for all A. That is, there is a polynomial time machine M so that for every A, $L(M^A)=L(N_i^A)$. The machine simply implements the procedure used in the proofs of Section 5.5.

5.4 A Note on Weakening the Hypotheses

In Theorems 5.1 and 5.3 and Corollaries 5.2 and 5.4, we can restrict our hypotheses to sparse oracles. This follows from the easy observation that a machine is, e.g., categorical for all oracles exactly when it is categorical for all sparse oracles.

Lemma 5.11

1. $(\forall A)[N^A \text{ is categorical}] \Leftrightarrow (\forall \text{ sparse } S)[N^S \text{ is categorical}]$.
2. $(\forall A)[L(N_i^A)=\overline{L(N_j^A)}] \Leftrightarrow (\forall \text{ sparse } S)[L(N_i^S)=\overline{L(N_j^S)}]$.

Proof Sketch for Lemma 5.11 The $\Rightarrow$ directions are direct. The other directions hold because if a machine is, e.g., noncategorical for (dense) oracle A', it fails to be categorical on some specific string x. Thus for any sparse oracle S' that agrees with A' on a prefix large enough to include all strings queried during the run of $N^{A'}(x)$, we know that $N^{S'}(x)$ will be noncategorical. (Note that the definition of sparseness allows oracles that are dense on a finite prefix.)

As an example, we can restate Theorems 5.1 and 5.3 as follows.

Theorem 5.12 (Robustly categorical machines accept simple languages) $(\forall \text{ sparse } S)[N_i^S \text{ is categorical}] \Rightarrow (\forall A)[L(N_i^A)\in \mathrm{P}^{\mathrm{NP}\oplus A}]$.

Theorem 5.13 (Robustly complementary machines accept simple languages)
$(\forall \text{ sparse } S)[L(N_i^S)=\overline{L(N_j^S)}] \Rightarrow (\forall A)[L(N_i^A)\in \mathrm{P}^{\mathrm{NP}\oplus A}]$.

Proofs of Theorems 5.12 and 5.13 These follow directly from Lemma 5.11 and Theorems 5.1 and 5.3.

5.5 Proof Sketches for Robustness Theorems

The proof techniques here are inspired by the method Baker, Gill and Solovay use to show that there is a relativized world A where $\mathrm{P}^A = \mathrm{NP}^A \cap \mathrm{coNP}^A \neq \mathrm{NP}^A$ [BGS75] and the method Rackoff uses to show that there is a relativized world A for which $\mathrm{P}^A = \mathrm{UP}^A \neq \mathrm{NP}^A$ [Rac82].

Proof of Theorem 5.1

Use P^{NP} to find if for some value of B' there is an accepting computation of $N_i^{B'}(x)$; if not, reject x. Use P^{NP} to get the path, say $path_0$. Query A about all elements in the path, let S_0 be all elements queried on the path, and let W_0 be the elements on which the path was wrong (disagreed with A). If the path was never wrong, we have a true accepting path of $N_i^A(x)$, so accept x.

Similarly, use P^{NP} to find if, for some value of B' consistent with our knowledge about the elements of S_0, there is an accepting computation of $N_i^{B'}(x)$; if not, reject x. Use $\mathrm{P}^{\mathrm{NP}}$to get the path, say $path_1$. Query all elements in the path, let S_1 be all elements queried on the path, and let W_1 be the elements on which the path was wrong (disagreed with A). If the path was never wrong, we have a true accepting path, so accept x.

Keep repeating this. The process finishes quickly. Why? Each $path_k$ must *conflict* with each of the paths $path_0, path_1, \ldots, path_{k-1}$, since N_i is robustly categorical. Note that for every j and l such that $j \neq l$, we have $W_j \cap W_l = \emptyset$. Thus $path_k$ must conflict with $path_0$ on some element that is both in W_0 of $path_0$ and $S_k - W_k$ of $path_k$. Similarly, it conflicts with $path_1$ on some element that is both in W_1 of $path_1$ and in $S_k - W_k$ of $path_k$, and so on. But since the W_j's are disjoint, we take up $k-1$ spaces of $S_k - W_k$ just to disagree with the previous paths. Thus the process goes on at most until we examine $|x|^i + i$ paths. At that point we either have eliminated all paths (so $N_i^A(x)$ rejects) or we have found a path consistent with our oracle (so $N_i^A(x)$ accepts). Thus we have accepted the language of N_i^A, for arbitrary fixed A, with a $\mathrm{P}^{\mathrm{NP}\oplus A}$ machine.

Proof of Theorem 5.3

This is like Theorem 5.1, but is a bit more involved. Recall we have machines N_i and N_j that are robustly complementary. W.l.o.g. they are respectively in NTIME$[n^i+i]$ and NTIME$[n^j+j]$. We consider the family $\mathcal{F}_i$ of paths, over all oracles A', on which $N_i^{A'}(x)$ accepts, and also consider the family $\mathcal{F}_j$ of paths, over all oracles A', on which $N_j^{A'}(x)$ accepts.

Now we use NP to get an accepting path from $\mathcal{F}_i$ and query in A all elements along the path. Then we use NP to get a path from $\mathcal{F}_j$ that is consistent with our knowledge of A on all elements in the first path. Continue this, crucially alternating between $\mathcal{F}_i$ and $\mathcal{F}_j$.

If we ever fail to find a path we know that family has no accepting paths and we are done (the machine of that family rejects). If we ever find a path that agrees with A we have a true accepting path and again are done (the machine the path belongs to accepts). Note that every pair of one path from $\mathcal{F}_i$ and one path from $\mathcal{F}_j$ must explicitly conflict over the membership of some element in A (or our machines would not be robustly complementary). But now the argument of the proof of Theorem 5.1 applies. Each path we take from $\mathcal{F}_j$ conflicts with each previous path from $\mathcal{F}_i$ on a different element, so our whole process terminates after at most $2\max(|x|^i+i, |x|^j+j)$ paths have been studied. □

Proof of Theorems 5.7 and 5.5 These are much easier to prove than the general theorems discussed above. We use an iterative adaptation of the methods of Baker, Gill, and Solvay [BGS75] and Rackoff [Rac82].

As an example, we prove Theorem 5.7.

Let N_i be the machine of the theorem. It robustly accepts Σ^* for sparse oracles. Our goal is to, in P^S, find an accepting path of $N_i^S(x)$.

Use NP to find an accepting path of $N_i^{\emptyset}$. Query S about all elements along the path. If none are in S, then we have a true accepting path of $N_i^S(x)$ and are done. If some are in S, then we've discovered some elements of S, call them S_0.

Now use NP to find an accepting path of $N_i^{S_0}$ (there must be one, as N_i robustly accepts Σ^*). Again, if no elements on the path are in $S - S_0$, we have a true accepting path. Otherwise, we have discovered some new elements of S. Let S_1 be the union of these elements and S_0. Keep repeating this.

How long can this go on? We find new elements of S at each step (or we have an accepting path and are done), but since S is sparse, it only has a polynomial number of elements that can be touched by the run of $N_i^S(x)$. So in a polynomial number of steps, we have found *all* the strings of S that $N_i^S(x)$ can touch, and the next use of NP will give us a true accepting path. □

The proof of Theorem 5.9 uses the same techniques as the proofs just stated.

5.6 Conclusions

By repeatedly reducing the sets of plausible accepting paths, we've shown that if P equals NP, robust machines are weak—with no oracle can they separate NP from P, have mysterious accepting paths, or have hard selector functions.

Thus the tragedy of a machine N that has a robustness property is that for every oracle A, N^A squanders its powerful access to its oracle. A mere polynomial time machine with oracle $\mathrm{NP} \oplus A$ can accept the language N^A accepts.

Chapter 6

Uniqueness

6.1 Chapter Overview

This chapter studies two uses of uniqueness. Section 6.2 shows that $P \neq UP \cap coUP$ if and only if there is a set S of formulas that obviously each have exactly one satisfying assignment but no polynomial time machine can find the (unique) satisfying assignments.

This is a UP analogue of a theorem of Borodin and Demers [BD76] about $P \neq NP \cap coNP$. However, only one direction is known for the $NP \cap coNP$ case.

Section 6.3 constructs a fast algorithm for SAT, under a complexity-theoretic assumption. The algorithm quickly finds satisfying assignments for satisfiable formulas that do not have many satisfying assignments.

6.2 A UP Analogue of the Borodin-Demers Theorem

Borodin and Demers [BD76] prove:

Theorem 6.1 (Borodin-Demers Theorem) If $P \neq NP \cap coNP$ then there is a set S so

1. $S \in P$ and $S \subseteq SAT$, and

2. no P machine can find solutions for all formulas in S. That is, for *no* polynomial time computable function g do we have $(\forall f)[f \in S \Rightarrow g(f)$ is a satisfying assignment of $f]$.

Thus, if $\mathrm{P} \neq \mathrm{NP} \cap \mathrm{coNP}$ there is an easily recognizable set S of satisfiable formulas whose satisfying assignments cannot be easily found. Loosely, we easily know that S formulas are satisfiable but we cannot easily tell *why* they are satisfiable. It is not known if the converse of this theorem holds, which would yield a complete characterization of $\mathrm{P} \neq \mathrm{NP} \cap \mathrm{coNP}$.

We prove a UP analogue of the Borodin-Demers Theorem, and show that the converse of the analogue holds. Thus we completely characterize $\mathrm{P} \neq \mathrm{UP} \cap \mathrm{coUP}$.

Theorem 6.2 $\mathrm{P} \neq \mathrm{UP} \cap \mathrm{coUP}$ if and only if there is a set S so

1. $S \in \mathrm{P}$ and $S \subseteq \mathrm{SAT}$,
2. $f \in S \Rightarrow f$ has exactly one solution, and
3. no P machine can find solutions for all formulas in S. That is,

$$g(f) = \begin{cases} 0 & f \notin S \\ \text{the unique satisfying assignment of } f & f \in S \end{cases}$$

is not a polynomial time computable function.

This theorem shows that if the (co)unique acceptance model yields power beyond P, then sets with bizarre properties exist. However, we need not consider these results as evidence that $\mathrm{P} = \mathrm{UP} \cap \mathrm{coUP}$. Rather, we should view these results as reflections of the amazing power of logical formulas to describe computations—a power that spawned the theory of effective computability.

Proof of Theorem 6.2

($\Longrightarrow$) Let $L_0 \in (\mathrm{UP} \cap \mathrm{coUP}) - \mathrm{P}$. Let N_0 and N_1 be categorical machines accepting, respectively, L_0 and $\overline{L_0}$.

Construct a machine N that on input x nondeterministically simulates $N_0(x)$ and $N_1(x)$. Now $L(N) = L(N_0) \cup L(N_1) = L_0 \cup \overline{L_0} = \Sigma^*$. Since N_0 and N_1 are categorical, N has exactly one accepting path on each input. Thus, letting $F_{N,x}$ be the

Cook's theorem formula for N's computation on x, $F_{N,x}$ has exactly one satisfying assignment (since Cook's reduction is parsimonious).

Let $S = \cup_{x \in \Sigma^*} F_{N,x}$. From the structure of Cook's reduction (as $F_{N,x}$ clearly displays N and x) S is in P. By the previous paragraph, $f \in S$ implies f has exactly one solution. Thus conditions 1 and 2 of Theorem 2 are met by S.

From the satisfying assignment to $F_{N,x}$ we can quickly determine whether $x \in L_0$ or $x \in \overline{L_0}$, by checking which path of the initial branching led to acceptance. Thus if some polynomial time machine on input $f \in S$ outputs the (unique) satisfying assignment of f, then $L_0 \in \mathrm{P}$. This contradicts our assumption that $L_0 \notin \mathrm{P}$ and proves condition 3.

($\Longleftarrow$) Let $S' = \{<f, a_1, a_2, ..., a_k> \mid f \in S$ and each a_i assigns some variable in f and $f(a_1, a_2, ..., a_k)$ is uniquely satisfiable$\}$. $f(a_1, ..., a_k)$ specifies the formula resulting from making the assignments $a_1, ..., a_k$ in f. For example, if $f = x_1x_2x_3$ and $a_1 =$ "x_2 is true," then $f(a_1) = x_1x_3$. $\overline{a_1}$ here would mean "x_2 is false" and $f(\overline{a_1})$=False.

If S' were in P, then we could use tree search to find the satisfying assignment for any formula in S, contradicting condition 3. So $S' \notin \mathrm{P}$. It is obvious that $S' \in \mathrm{UP}$.

To see that $S' \in \mathrm{coUP}$, simply note that $\overline{S'} = \{<f, a_1, a_2, ..., a_k> \mid f \notin S$ or $[f \in S$ and $f^* = f(\overline{a_1}) \vee f(a_1, \overline{a_2}) \vee ... \vee f(a_1, a_2, ..., \overline{a_k})$ is uniquely satisfiable$]\}$. f^* has at most one solution; it just picks up all assignments contradicting "$a_1, ..., a_k$." Thus $\overline{S'} \in \mathrm{UP}$, so $S' \in \mathrm{coUP}$. So $S' \in (\mathrm{UP} \cap \mathrm{coUP}) - \mathrm{P}$. □

Of course, if $\mathrm{P} = \mathrm{UP} \cap \mathrm{coUP}$ then this theorem is of little interest. However, it is easy to diagonalize so that $\mathrm{P}^A \neq \mathrm{UP}^A \cap \mathrm{coUP}^A$, thus witnessing the possibility that $\mathrm{P} \neq \mathrm{UP} \cap \mathrm{coUP}$.

Fact 6.3 There is a recursive A so $\mathrm{P}^A \neq \mathrm{UP}^A \cap \mathrm{coUP}^A$.

6.3 Conditionally Fast Algorithms for Finding Satisfying Assignments: On Distinguishing Zero from One

6.3.1 Introduction

In their paper, "NP is as Easy as Exact Solutions," Valiant and Vazirani show how to probabilistically reduce a (satisfiable) formula to one that with high probability has exactly one solution [VV85]. It follows that if the (unlikely) Conjecture $(**)$ holds then $\mathrm{NP} = \mathrm{R}$.

Conjecture (**) There exist polynomial time algorithms that correctly accept SAT for inputs with at most one solution (but possibly err on other inputs).

The conjecture $(**)$ can be written $(\exists\, Q)[\mathrm{USAT}_Q \in \mathrm{P}]$, where $\mathrm{USAT}_Q = \{f \mid (||f|| > 1 \wedge Q(f)) \vee ||f|| = 1\}$. Valiant and Vazirani [VV85,Vaz87] also show that $(**)$ implies $\mathrm{P} = \mathrm{UP}$. Note that $(**)$ does not appear to imply $\mathrm{P} = \mathrm{NP}$.

We ask how likely it is that $(**)$ holds, and thus that one and zero can be distinguished in polynomial time. We also ask if one and zero can be distinguished in other time classes (UP, $\mathrm{NP} \cap \mathrm{coNP}$, FewNP). We could also ask if few and zero can be distinguished, and show that this gives algorithms that are similar to but slightly faster than those of the zero-vs-one case.

The result of [VV85] that $(**) \Rightarrow \mathrm{NP} = \mathrm{R}$ gives evidence that $(**)$ is false, though we should note that there are relativized worlds where $\mathrm{P}^A \neq \mathrm{NP}^A$ yet $\mathrm{NP}^A = \mathrm{R}^A$ [Rac82]. The result of Valiant and Vazirani that $(**) \Rightarrow \mathrm{P} = \mathrm{UP}$ also gives evidence that $(**)$ is false. $\mathrm{P} \neq \mathrm{UP}$ if and only if one-way functions exist (Grollmann and Selman [GS84]). Since we suspect that one-way functions do exist, this suggests that $(**)$ is false. However, even if $\mathrm{P} \neq \mathrm{NP}$, it is possible that one-way functions do not exist (Rackoff [Rac82]).

We present strong evidence that $(**)$ is false by showing that $(**)$ implies a surprising structure to the complexity of SAT. If $(**)$, then given a satisfiable formula f we can find a satisfying assignment to f in time $\mathrm{O}(|f|^k \cdot ||f|| \cdot \binom{\#var(f)}{\min(\log ||f||, \#var(f)/2)})$ where $||f||$ is the number of satisfying assignments to f, $|f|$ is the size of f, $\#var(f)$

is the number of distinct variable names in f, and k is a constant.

Thus (**) says that formulas with few solutions have subexponential time witness-finding algorithms,[1] and formulas with $o(2^{\sqrt{n}})$ solutions have witness finding algorithms faster than any now known. Intuition and relativized evidence[2] say that NP witness finding requires exponential time even for formulas with few solutions. This suggests that (**) is false.

Even though, given the above evidence, we suspect that distinguishing zero from one in polynomial time is unlikely, we might hope that one can distinguish zero from one within more powerful classes. Indeed, it is clear that we can (trivially) distinguish zero from one in NP: $(\exists\, Q)[\mathrm{USAT}_Q \in \mathrm{NP}]$. What about smaller classes? For example, can we tell zero from one within unique polynomial time (Section 2.7)—i.e., $(\exists\, Q)[\mathrm{USAT}_Q \in \mathrm{UP}]$?

Valiant and Vazirani's proof that that $(**) \Rightarrow \mathrm{NP} = \mathrm{R}$, does *not* show that $(\exists\, Q)$ $[\mathrm{USAT}_Q \in \mathrm{UP}] \Rightarrow \mathrm{UP} = \mathrm{NP}$, as their reductions are randomized. Adopting the approach of Valiant and Vazirani's observation that $(**) \Rightarrow \mathrm{P} = \mathrm{UP}$, we see that $(\exists\, Q)[\mathrm{USAT}_Q \in \mathrm{UP}] \Rightarrow \mathrm{UP} = \mathrm{UP}$. Unfortunately, UP *does* equal UP, so this does not tarnish the hypothesis.

We show that if $(\exists\, Q)[\mathrm{USAT}_Q \in \mathrm{UP}]$ then we can find witnesses for satisfiable formulas in:

$$\mathrm{TIME}^{\mathrm{UP}}[|f|^k \cdot ||f|| \cdot \binom{\#var(f)}{\min(\log ||f||, \#var(f)/2)}].$$

This is unlikely and thus suggests that the hypothesis is false, and that UP lacks the power to distinguish zero from one.

Similar results hold for distinguishing zero from one in FewNP, $\mathrm{NP} \cap \mathrm{coNP}$, etc., and an analogous set of results, with sharper time bounds, hold for $\mathrm{LinearSAT}_Q \in \mathrm{P}$.[3]

---

[1]Though these algorithms quickly find one of few solutions, if a given formula has *no* solutions the algorithms may take exponentially long to detect this.

[2]**Fact** For every k there is an oracle A and a machine N so N^A is categorical (and thus has at most one witness, see Section 2.7, yet no deterministic algorithm to find N^A's witnesses runs in time 2^{n^k}.

Proof Sketch Let $L_A = \{0^n | (\exists y)[|y| = n^{k+1} \wedge y \in A]\} \in \mathrm{NP}^A$. Putting at most one string of each length in A (thus assuring that $L_A \in \mathrm{UP}^A$ and thus $N(x)$ has at most one witness), diagonalize in turn against each $\mathrm{TIME}[2^{n^k}]$ machine. □

[3]$\mathrm{LinearSAT}_Q = \{f | (||f|| > n \wedge Q(f)) \vee n \geq ||f|| > 0\}$. That is, $\mathrm{LinearSAT}_Q$ agrees with SAT on

Thus it seems likely that distinguishing zero from one lies far beyond the power of P and that the complexity of unique solutions is great.

6.3.2 Theorems, Algorithms, Proofs

We show that, if (**) holds, then there exist fast algorithms for finding solutions to formulas with few solutions.

Theorem 6.4 If $(\exists\, Q)[\mathrm{USAT}_Q \in \mathrm{P}]$ then there exists a machine M so if f is a satisfiable formula, then M(f) prints a satisfying assignment of f in time $O(|f|^k \cdot ||f|| \cdot \binom{\#var(f)}{\min(\log||f||, \#var(f)/2)})$.

Some corollaries follow. Though less interesting than the theorem, they tell us about the effect of (**) on classes below NP.

Corollary 6.5 If $(\exists\, Q)[\mathrm{USAT}_Q \in \mathrm{P}]$ then

1. P = UP [Vaz87,VV85], and
2. $(\forall k)[\mathrm{P} = \mathrm{NP}_{\leq k}]$.[4]
3. $\mathrm{FewNP} \in \cup_c \mathrm{TIME}[n^{c \log n}]$.[5]

Corollary 6.6 If $(\exists\, Q)[\mathrm{USAT}_Q \in \mathrm{UP}]$ then there exists a machine M so if f is a satisfiable formula, then M(f) prints a satisfying assignment of f in $\mathrm{TIME}^{\mathrm{UP}}[|f|^k \cdot ||f|| \cdot \binom{\#var(f)}{\min(\log||f||, \#var(f)/2)}]$.

Corollary 6.7 If $(\exists\, Q)[\mathrm{USAT}_Q \in \mathrm{FewNP}]$ then there exists a machine M so if f is a satisfiable formula, then M(f) prints a satisfying assignment of f in $\mathrm{TIME}^{\mathrm{FewNP}}[|f|^k \cdot ||f|| \cdot \binom{\#var(f)}{\min(\log||f||, \#var(f)/2)}]$.

---

formulas of size n that have at most n solutions, but it may disagree with SAT on formulas with more than n solutions.

[4] $\mathrm{NP}_{\leq k} = \{L \mid \exists$ nondeterministic polynomial time Turing machine N so N accepts L and for no input x does $N(x)$ have more than k accepting paths. This part of the corollary could also be proved from part 1 of the corollary combined with the result of Osamu Watanabe that $\mathrm{P} = \mathrm{UP} \Rightarrow (\forall k)[\mathrm{P} = \mathrm{NP}_{\leq k}]$ [Wat87].

[5] FewNP is defined in Section 2.7.

Corollary 6.8 If $(\exists\, Q)[\mathrm{USAT}_Q \in \mathrm{NP} \cap \mathrm{coNP}]$ then there exists a machine M so if f is a satisfiable formula, then M(f) prints a satisfying assignment of f in $\mathrm{TIME}^{\mathrm{NP}\cap\mathrm{coNP}}[|f|^k \cdot ||f|| \cdot \binom{\#var(f)}{\min(\log ||f||, \#var(f)/2)}]$.

The following lemma simplifies the proof of Theorem 6.4.

Lemma 6.9 If $(\exists Q)[\mathrm{USAT}_Q \in \mathrm{P}]$ then there exists a polynomial time Turing machine $M^\star$ so:

1. $||f|| = 1 \Rightarrow M^\star(f)$ prints the unique satisfying assignment of f, and
2. $||f|| \neq 1 \Rightarrow M^\star(f)$ either prints a satisfying assignment to f or prints "$||f|| \neq 1$."

Proof of Lemma 6.9

Let M be a polynomial time Turing machine that for some Q accepts USAT_Q. Let f^* be our input formula. Compute $M(f^*)$. If M claims that f^* is unsatisfiable (that is, M rejects f^*), print "$||f|| \neq 1$"; f^* is unsatisfiable or has more than one solution. Otherwise, set some variable of f^* to TRUE and FALSE. Call the resulting formulas f^*_{T} and f^*_{F}. If M accepts both f^*_{T} and f^*_{F} then print "$||f|| \neq 1$" as f^* must have $||f|| > 1$, as its subformulas have distinct solutions. If M rejects both f^*_{T} and f^*_{F} then forget f^*, it must have $||f|| > 1$, as $||f^*|| = 1$ or $||f^*|| = 0$ both conflict with the behavior shown by $f^* \in L(M)$, $f^*_{\mathrm{T}} \notin L(M)$, and $f^*_{\mathrm{F}} \notin L(M)$. Finally, if $f^*_{\mathrm{T}} \in L(M)$ and $f^*_{\mathrm{F}} \notin L(M)$, it is plausible that $||f^*|| = ||f^*_{\mathrm{T}}|| = 1$, so repeat the process just described in this paragraph for f^* on f^*_{T}. Similarly, if $f^*_{\mathrm{F}} \in L(M)$ and $f^*_{\mathrm{T}} \notin L(M)$, it is plausible that $||f^*|| = ||f^*_{\mathrm{F}}|| = 1$, so repeat the process just described in this paragraph for f^* on f^*_{F}.

If this process ever gets down to a leaf (fully substituted formula), take the assignment we have constructed amd see if it is a satisfying assignment to f^*. If so, we are done; we have, and print, a satisfying assignment to f^*. If not, we have been fooled—the current f^* must have many solutions; print "$||f|| \neq 1$." □

Proof Sketch for Theorem 6.4

In a nutshell, the structure of the proof is as follows. We seek to find a partially substituted version of f that has exactly one solution. This will quickly allow up to find the solution of that formula. Unfortunately, we don't know how to quickly obtain a partially substituted version of f that has exactly one solution. We give a scheme that lets us create a collection of formulas so that we know that at least one of the formulas in the collection will have exactly one solution (when $||f||$ is below a certain bound); this suffices.

Given f, we choose subsets of f's variables. For each subset we try all possible truth assignments (hoping to create a formula with exactly one solution), and with the resulting partially substituted formulas use machine $M^\star$ of Lemma 6.9 to try getting a satisfying assignment.

Given a satisfiable formula f, we want to find a satisfying assignment of f. As an aid, assumption (**) gives us, by Lemma 6.9, a machine $M^\star$ that correctly finds assignments for inputs with at most one satisfying assignment.

Running $M^\star$ on input f will give us little information; f might have more than one solution, thus $M^\star$ might fail. Our goal is to obtain a formula f' that has exactly one satisfying assignment, which is also a satisfying assignment of f. Using $M^\star$ on f' we can then quickly obtain that assignment.

Consider A, the set of satisfying assignments of f. Every two distinct elements of A differ on at least one variable. Thus, if $|A| > 1$, there is some variable x_k so that $A_{1,\mathrm{T}} = \{a | a$ is a satisfying assignment of f and a has $x_k =$ True$\}$ and $A_{1,\mathrm{F}} = \{a | a$ is a satisfying assignment of f and a has $x_k =$ False$\}$ are both nonempty. Consider the smaller of these two sets, which will be of size $\leq |A|/2$. If its size is greater than one, then it too has some variable that splits it into nonempty parts, and we can repeat this process. Thus, if we were magically guided in choosing the variables to split, in at most $\log ||f||$ splits we would split to a formula that has exactly one solution (i.e., f, with the split variables assigned as the splits dictate). Once we have f', a formula with exactly one solution (which itself indicates a solution of f) we can quickly find the solution using $M^\star$.

Unfortunately, we have assumed magic guidance in the choice of split variables. Lacking sorcerous power, we must make do with a more brutish approach: we try all possible sets of splitting variables.

However, we don't know $||f||$, so we don't even know how many splitting variables are needed. We solve this by trying, in turn, the following procedure with *split* set, in turn, to 2, 4, 8, 16, ... The variable *split* is our guess of an upper bound on $||f||$.

Now, for the choice of *split* we are currently working with, guess every possible set of *split* variables of f. There are at most $\binom{\#var(f)}{split}$ such sets. For each set, set the split variable to True and False in every possible combination, yielding 2^{split} partially substituted versions of f from each of the $\binom{\#var(f)}{split}$ splitting sets. For each of these $2^{split} \cdot \binom{\#var(f)}{split}$ formulas, attempt to self-reduce it, using the machine M^* of Lemma 6.9.

Note that if $split \geq \lceil \log ||f|| \rceil$, by our earlier argument one of the formulas will have exactly one solution, and thus M^* finds it.

Why does this work? As soon as $split > \log ||f||$, we know that some set of splitting assignments will yield at least one f^* with $||f^*|| = 1$, and that $M^*(f^*)$ yields a satisfying assignment. On the other hand, we can never be fooled into accepting an unsatisfying assignment as M^* only prints valid satisfying assignments and in any case we can always check any final assignment to insure that it *is* a satisfying assignment of f.

When $split = m$, the cost of the above procedure is, for some k (related to the degree of the polynomial run time of machine M^*), and with $|f| = n$,

$$O(n^{k-1} 2^m \binom{\#var(f)}{m}),$$

i.e., $\binom{\#var(f)}{m}$ splitting sets times 2^m substituted formulas per set times polynomial work tree-pruning per formula. Since we set $split = 2, 4, 8, \ldots$ and finish when $split \geq ||f||$, our total cost is (note that $\lceil \log ||f|| \rceil = O(n)$):

$$\begin{aligned}
\text{total cost} &= O(\sum_{0 \leq i \leq \lceil \log ||f|| \rceil} n^{k-1} 2^i \binom{\#var(f)}{i}) \\
&= O(\lceil \log ||f|| \rceil n^{k-1} 2^{\lceil \log ||f|| \rceil} \binom{\#var(f)}{\min(\lceil \log ||f|| \rceil, \#var(f)/2)}) \\
&= O(n^k ||f|| \binom{\#var(f)}{\min(\lceil \log ||f|| \rceil, \#var(f)/2)})
\end{aligned}$$

Recall that this analysis applies only when $||f|| > 0$; if f is unsatisfiable it may take us exponential time to detect its unsatisfiability. □

6.4 Conclusions and Open Problems

Many open problems remain.

1. Can we prove the converse of the Borodin-Demers Theorem (Theorem 6.1)?

2. Can we find further connections between the number of solutions a formula has and the complexity of finding some solution of the formula?

   We've shown that if there is a Q such that USAT_Q is in P, then formulas with *few* satisfying assignments have easily obtained satisfying assignments. Conversely, Ajtai and Widgerson [AW85, Section 5] show that formulas in CNF form with *many* satisfying assignments have easily obtained satisfying assignments—finding satisfying assignments for formulas in 3-CNF that are satisfied by, say, one quarter of their assignments can be done in polynomial time.

3. How likely is it that $\mathrm{P} = \mathrm{UP}$ for a random oracle? That is, is $\mathrm{Prob}_A(\mathrm{P}^A = \mathrm{UP}^A) = 1$? Blum and Impagliazzo [Imp87] have shown that if $\mathrm{P} = \mathrm{NP}$ then for all generic oracles $\mathrm{P}^A = \mathrm{UP}^A$, but a probability one result has not yet been found.

Appendix A

Complexity Class Summary Sheets

P – Polynomial Time

Power

Feasible computation.

Definition

$\mathrm{P} = \bigcup_k \mathrm{TIME}[n^k]$.

Background

P was described as embodying the power of feasible computation by Cobham [Cob64] and Edmonds [Edm65]. The field of design and analysis of algorithms attempts to place as many problems as possible in P.

Complete Languages

P has well-known complete languages under $\leq_{many-one}^{logspace}$ reductions, e.g., the emptiness for context-free grammars [HU79].

Sample Problem

In a fixed, reasonable proof system, asking if x is a proof of T is a polynomial time question. In particular, in polynomial time we can check if assignment x satisfies boolean formula F.

Figure A.1: P

NP – Nondeterministic Polynomial Time

Power Guessing. Nondeterminism.

Definition $\mathrm{NP} = \bigcup_k \mathrm{NTIME}[n^k]$.

Background

In the early 1970s, Cook [Coo71], Karp [Kar72], and Levin [Lev73] initiated the study of NP and its complete problems. Many NP-complete problems are now known, and the study of NP's structure is the unifying theme of structural complexity theory.

Complete Problems

NP has hundreds of $\leq_m^p$ (polynomial time many-one) complete problems [GJ79]. The most studied NP-complete problem is satisfiability. SAT$=\{f \mid$ boolean formula f is satisfiable$\}$ was shown to be Turing-complete for NP by Cook. Karp showed that SAT and many other problems were $\leq_m^p$-complete for NP.

Theorems

- If $(\exists$ sparse $S)[\mathrm{NP} \subseteq \mathrm{P}^S]$ then the polynomial hierarchy, PH, equals $\mathrm{NP}^{\mathrm{NP}}$. [KL80]
- NP has sparse complete sets if and only if $\mathrm{P} = \mathrm{NP}$. [Mah80]
- $\mathrm{NP} - \mathrm{P}$ contains sparse sets if and only if $\mathrm{E} \neq \mathrm{NE}$. [HIS85]
- All paddable sets are p-isomorphic to SAT. [BH77,MY85]
- If $\mathrm{P} = \mathrm{NP}$ and S is sparse then $[\mathrm{P}^S = \mathrm{NP}^S \Leftrightarrow (\exists k)[S \subseteq K^S[k \log n, n^k]]]$, where $K[]$ represents time bounded Kolmogorov complexity. [HH86b]
- $\mathrm{P} \neq \mathrm{NP} \Rightarrow \mathrm{NP} - \mathrm{P}$ contains sets that are not NP-complete. [Lad75]
- $(\exists A)[\mathrm{P}^A = \mathrm{NP}^A]$. $(\exists B)[\mathrm{P}^B \neq \mathrm{NP}^B]$. [BGS75]

Figure A.2: NP

PH, PSPACE, P, NP, coNP, $\mathbf{P^{NP}}$, $\mathbf{NP^{NP}}$, ... – The Polynomial Hierarchy and Polynomial Space

Power Alternating polynomial bounded existential and universal quantifiers.

Definition

$$\begin{aligned}
\Sigma_0^p &= \Pi_0^p = \mathrm{P} \\
\Delta_{i+1}^p &= \mathrm{P}^{\Sigma_i^p} \qquad i \geq 0 \\
\Sigma_{i+1}^p &= \mathrm{NP}^{\Sigma_i^p} \qquad i \geq 0 \\
\Pi_{i+1}^p &= \mathrm{co}\Sigma_{i+1}^p = \{L \mid \overline{L} \in \Sigma_{i+1}^p\} \qquad i \geq 0 \\
\mathrm{PH} &= \bigcup_i \Sigma_i^p \\
\mathrm{PSPACE} &= \bigcup_k \mathrm{SPACE}[n^k].
\end{aligned}$$

Background The polynomial hierarchy was defined in (Stockmeyer [Sto77]).

Complete Languages Canonical complete languages exist for each level of the hierarchy [Wra77] and for PSPACE [Sto77].

Theorems
- $(\exists A)[\mathrm{P}^A = \mathrm{PH}^A]$. [BGS75]
- $(\exists A)[\mathrm{P}^A \neq \mathrm{NP}^A \neq \mathrm{NP}^{\mathrm{NP}^A} \neq \cdots \neq \mathrm{PSPACE}^A]$. [Yao85]
- $\mathrm{PH} = \mathrm{PSPACE} \Leftrightarrow (\forall \text{ sparse } S)[\mathrm{PH}^S = \mathrm{PSPACE}^S]$. [BBL*84]
- $\Sigma_k^p = \Pi_k^p \Rightarrow \Sigma_k^p = \mathrm{PH}$ (Downward Separation). [Sto77]
- $\mathrm{Prob}_A(\mathrm{PH}^A \neq \mathrm{PSPACE}^A) = 1$. [Cai86]
- PSPACE = NPSPACE = Probabilistic-PSPACE. [Sav70,Sim77]

Open Problems
- Does the polynomial hierarchy collapse?
- $\mathrm{Prob}_A(\mathrm{P}^A \neq \mathrm{NP}^A \neq \mathrm{NP}^{\mathrm{NP}^A} \neq \cdots) = 1$?
- For which j can we construct oracles so $(\Sigma_j^p)^A \neq (\Sigma_{j+1}^p)^A = \mathrm{PH}^A$?
- $(\exists k, A)[(\Sigma_k^p)^A = (\Pi_k^p)^A \neq \mathrm{PSPACE}^A]$?

Figure A.3: The Polynomial Hierarchy and PSPACE

E, NE, P^{NE}, NP^{NE}, ... – The Strong Exponential Hierarchy

Power Exponential computation hierarchy.

Definitions

$$\begin{aligned}
\mathrm{E} &= \Sigma_0^{\mathrm{SEH}} = \bigcup_c \mathrm{TIME}[2^{cn}] \\
\mathrm{NE} &= \Sigma_1^{\mathrm{SEH}} = \bigcup_c \mathrm{NTIME}[2^{cn}] \\
\Delta_i^{\mathrm{SEH}} &= \mathrm{P}^{\Sigma_{i-1}^{\mathrm{SEH}}} \quad i \geq 2 \\
\Sigma_i^{\mathrm{SEH}} &= \mathrm{NP}^{\Sigma_{i-1}^{\mathrm{SEH}}} \quad i \geq 2 \\
\mathrm{SEH} &= \mathrm{E} \cup \mathrm{NE} \cup \mathrm{NP}^{\mathrm{NE}} \cup \mathrm{NP}^{\mathrm{NP}^{\mathrm{NE}}} \cup \cdots \\
\mathrm{EXPSPACE} &= \bigcup_k \mathrm{SPACE}[2^{n^k}]
\end{aligned}$$

Background The complexity of sparse sets in the polynomial hierarchy is closely related to the structure of exponential time classes [Boo74,HH74, HY84,HIS85,CH86a,CHHS86b].

Complete Languages All these classes have straightforward canonical complete languages that capture the actions of generic machines (see the techniques of [Har78]).

Theorems

- $\mathrm{E} = \mathrm{NE}$ if and only if there are no tally sets in $\mathrm{NP} - \mathrm{P}$. [Boo74,HH74]
- $\mathrm{E} = \mathrm{NE}$ if and only if there are no sparse sets in $\mathrm{NP} - \mathrm{P}$. [HIS85]
- $\mathrm{E} = \mathrm{NE}$ if and only if all capturable sets in the boolean hierarchy are in P. [CH86a]
- $\mathrm{NE} = \mathrm{coNE}$ if and only if every sparse set in NP is NP-printable. [HY84]
- $\mathrm{P}^{\mathrm{NE}} = \mathrm{E} \cup \mathrm{NE} \cup \mathrm{NP}^{\mathrm{NE}} \cup \mathrm{NP}^{\mathrm{NP}^{\mathrm{NE}}} \cup \cdots$. (Chapter 3)
- $\mathrm{E} = \mathrm{NE} \Rightarrow \mathrm{EXP} = \mathrm{SEH}$ (Downward Separation). (Chapter 3)

Figure A.4: E, NE, and the Strong Exponential Hierarchy

P/poly – Nonuniform Polynomial Time

Power

Small circuits. Table lookup.

Definition

$\mathrm{P/poly} = \{L \mid (\exists \text{ sparse } S)[L \in \mathrm{P}^S]\}$

Theorems

- $\mathrm{NP} \subseteq \mathrm{P/poly} \Rightarrow \mathrm{PH} = \mathrm{NP}^{\mathrm{NP}}$. [KL80]
- $(\exists S)[\mathrm{NP} \subseteq \mathrm{P}^S \wedge S \text{ sparse} \wedge S \in \mathrm{NP}] \Rightarrow \mathrm{PH} = \mathrm{P}^{\mathrm{NP}[\log]}$. [Kad87]
- There is a relativized world A and a sparse set S so $\mathrm{PSPACE}^A \subseteq \mathrm{P}^{A \oplus S}$ yet the boolean hierarchy relative to A is infinite. [CH86a]
- If P has small ranking circuits then $\mathrm{P}^{\#\mathrm{P}} \subseteq \mathrm{PH} = \mathrm{NP}^{\mathrm{NP}}$. (Chapter 4)
- If P has small ranking circuits then P has small ranking circuits that can be printed by a Δ_3^p machine. (Chapter 4)

Figure A.5: P/Poly

UP, US – Unique Polynomial Time

Power

Categorical acceptance. Uniqueness.

Definition

$$\begin{aligned} \mathrm{UP} &= \{L \mid \text{ there is a nondeterministic polynomial time Turing machine } N \\ &\quad \text{so } L = L(N), \text{ and for all } x, \text{ the computation of } N(x) \text{ has at most} \\ &\quad \text{one accepting path}\}. \\ \mathrm{US} &= \{L \mid \text{ there is a polynomial predicate } P \text{ and integer } k \text{ so for all } x, \\ &\quad x \in L \Leftrightarrow |\{y \mid |y| \leq |x|^k \wedge P(x,y)\}| = 1\}. \end{aligned}$$

Background

UP is defined in (Valiant [Val76]). US is studied in (Blass and Gurevich [BG82]). UP is related to cryptography [GS84] and to central conjectures in structural complexity theory [JY85,HH87].

Complete Problems

USAT$=\{f \mid f$ has exactly one satisfying assignment$\}$ is complete for US.

UP may not have complete languages. There are relativized worlds where it does not have complete languages and relativized worlds where $\mathrm{P}^A \neq \mathrm{UP}^A \neq \mathrm{NP}^A$ yet UP^A does have complete languages [HH86a].

Figure A.6: UP—Part I

UP, US – Unique Polynomial Time

Theorems

- $P \neq UP \Leftrightarrow$ one-way functions exist. [GS84]
- $P \neq UP \cap coUP \Leftrightarrow$ one-way functions whose range is in P exist. [GS84]
- $UP \subseteq NP \cap US$
- If UP has complete languages then it has a complete language of the form $L = SAT \cap A$, $A \in P$. [HH86a]
- There is an oracle so $USAT^A$ is not $(D^P)^A$-complete. [BG82]
- $(\exists A)[P^A = UP^A \neq NP^A]$. $(\exists B)[P^B \neq UP^B = NP^B]$. [Rac82]
- $(\exists A)[UP^A$ has no complete languages$]$. [HH86a]
- $(\exists A)[P^A \neq UP^A \neq NP^A$ and UP^A has complete languages $]$. [HH86a]
- $(\forall A)[N_i^A$ is categorical $] \Rightarrow (\forall A)[L(N_i^A) \in P^{NP \oplus A}]$. (Chapter 5) and [HH87]
- There is a reasonable (i.e., $P^A \neq NP^A$) oracle A for which $P^A = UP^A$ (that is, there are no one-way functions) yet there are sets that are $\leq_m^{p,A}$-complete for NP^A and are non-p^A-isomorphic. [HH87]
- $P \neq UP \cap coUP$ if and only if there is a set S so (1) $S \in P$ and $S \subseteq SAT$, and (2) $f \in S \Rightarrow f$ has exactly one solution, and (3) no P machine can find solutions for all formulas in S—that is,

$$g(f) = \begin{cases} 0 & f \notin S \\ \text{the unique satisfying assignment of } f & f \in S \end{cases}$$

is not a polynomial time computable function. (Chapter 6)

Open Problems $\text{Prob}_A(P^A = UP^A) = 1$? Does UP have complete languages [HH86a]? Do one-way functions exist [GS84]?

Figure A.7: UP—Part II

#P – Sharp P (Counting Functions)

Power

Counting.

Definition

$$\#\mathrm{P} = \{f \mid (\exists \text{ nondeterministic polynomial time Turing machine } N)(\forall x)\\ [f(x) = \text{ number of accepting paths of } N(x)]\}.$$

Background

#P was first studied by Valiant [Val79a], who showed that counting versions not only of NP-complete problems but also of P problems can be #P-complete.

Complete Problems

#SAT is a representative #P function: $\mathrm{P}^{\#\mathrm{P}[1]} = \mathrm{P}^{\#\mathrm{SAT}[1]}$.

Theorems

- $(\exists A)[\mathrm{P}^{\#\mathrm{P}^A} - ((\Sigma_2^p)^A \cup (\Pi_2^p)^A) \neq \emptyset]$. [Ang80]
- If #SAT has a $\mathrm{O}(n^{1-\epsilon})$ enumerative approximator then $\mathrm{P} = \mathrm{P}^{\#\mathrm{P}}$. [CH86a]
- $\mathrm{P} = \mathrm{P}^{\#\mathrm{P}}$ if and only if every P set has a polynomial time computable ranking function. [GS85], [Rud87], and (Chapter 4)

Open Problems

- $\mathrm{P}^{\#\mathrm{P}} \subseteq \mathrm{PH}$?
- $\mathrm{PH} \subseteq \mathrm{P}^{\#\mathrm{P}}$?

Figure A.8: #P

Appendix B

Structural Inclusions

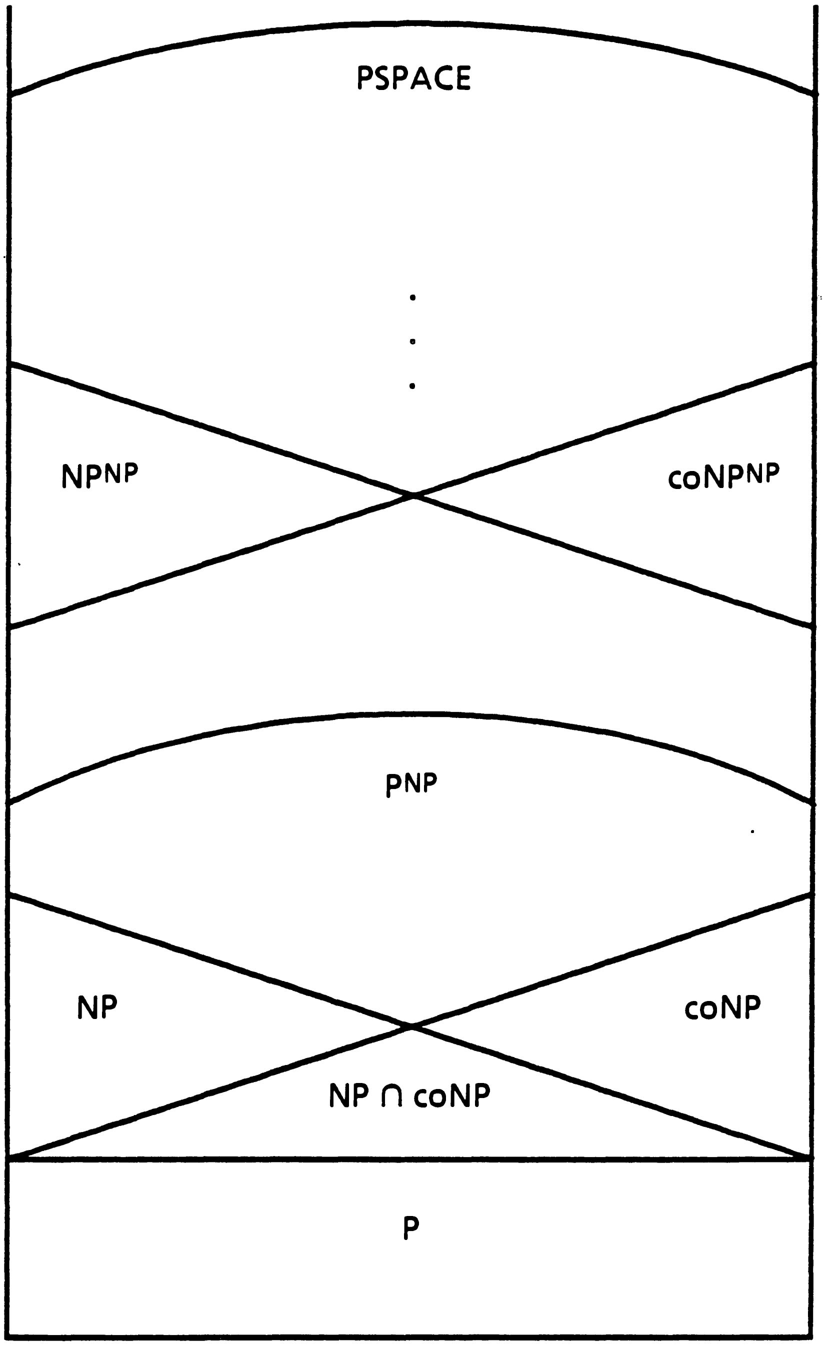


Figure B.1: The Structure of The Polynomial Hierarchy

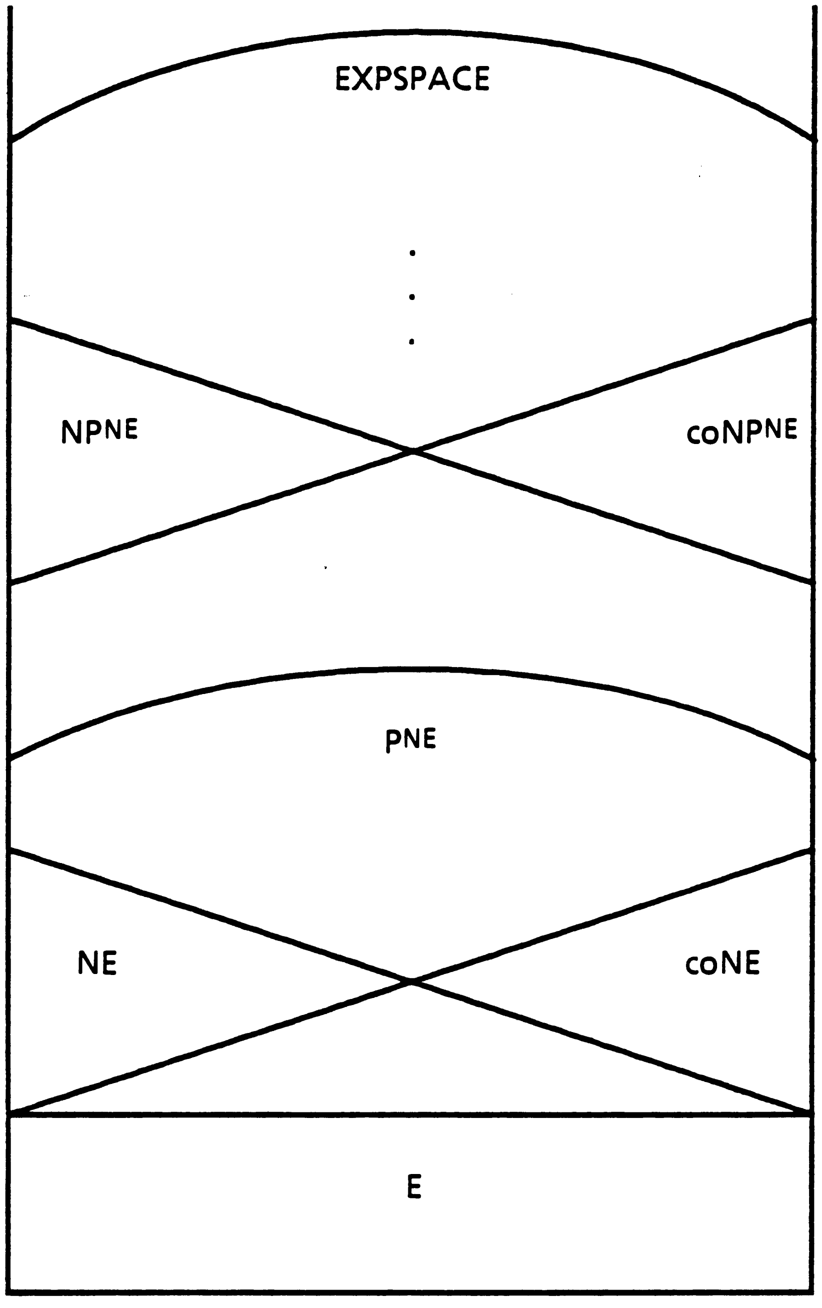


Figure B.2: The Structure of the Strong Exponential Hierarchy

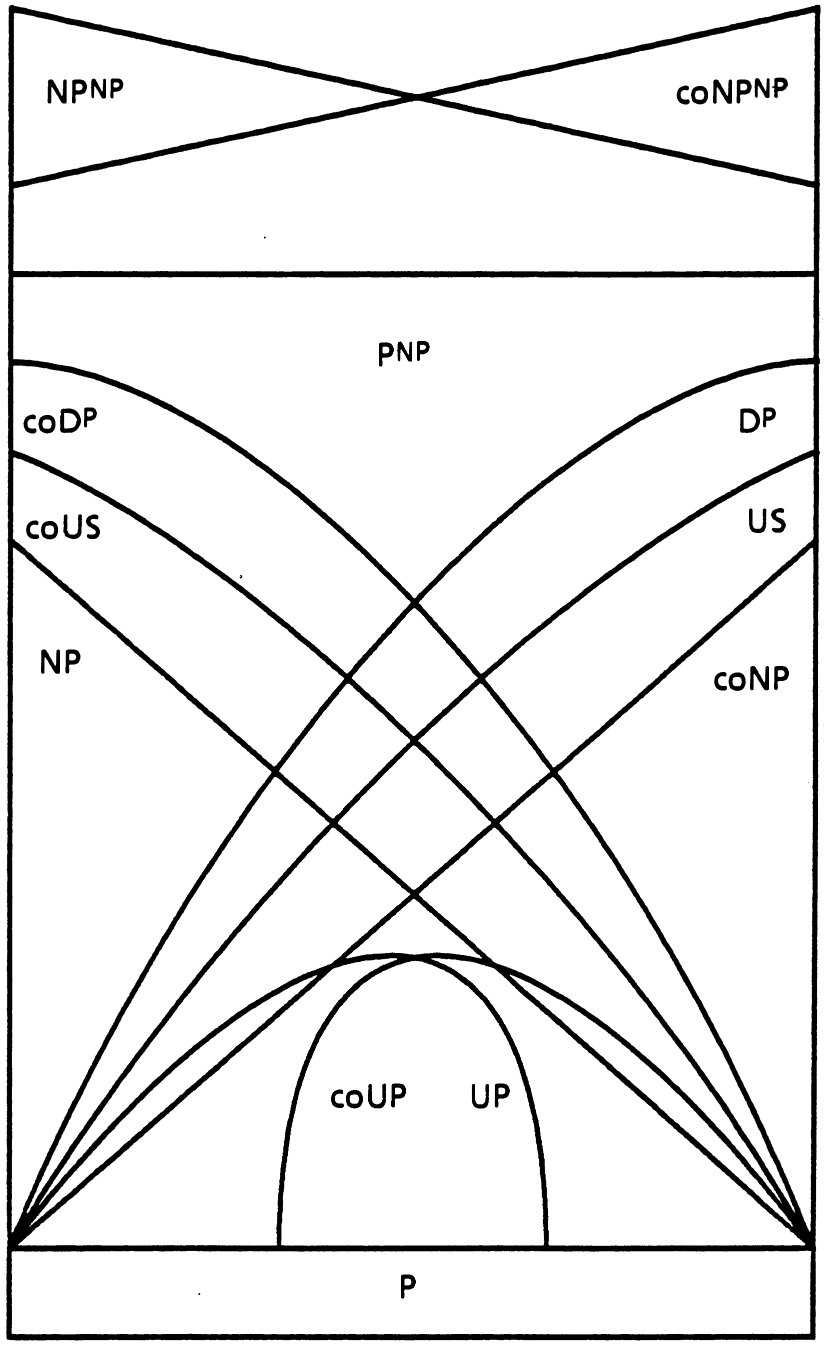


Figure B.3: The Structure of UP and US within the Polynomial Hierarchy

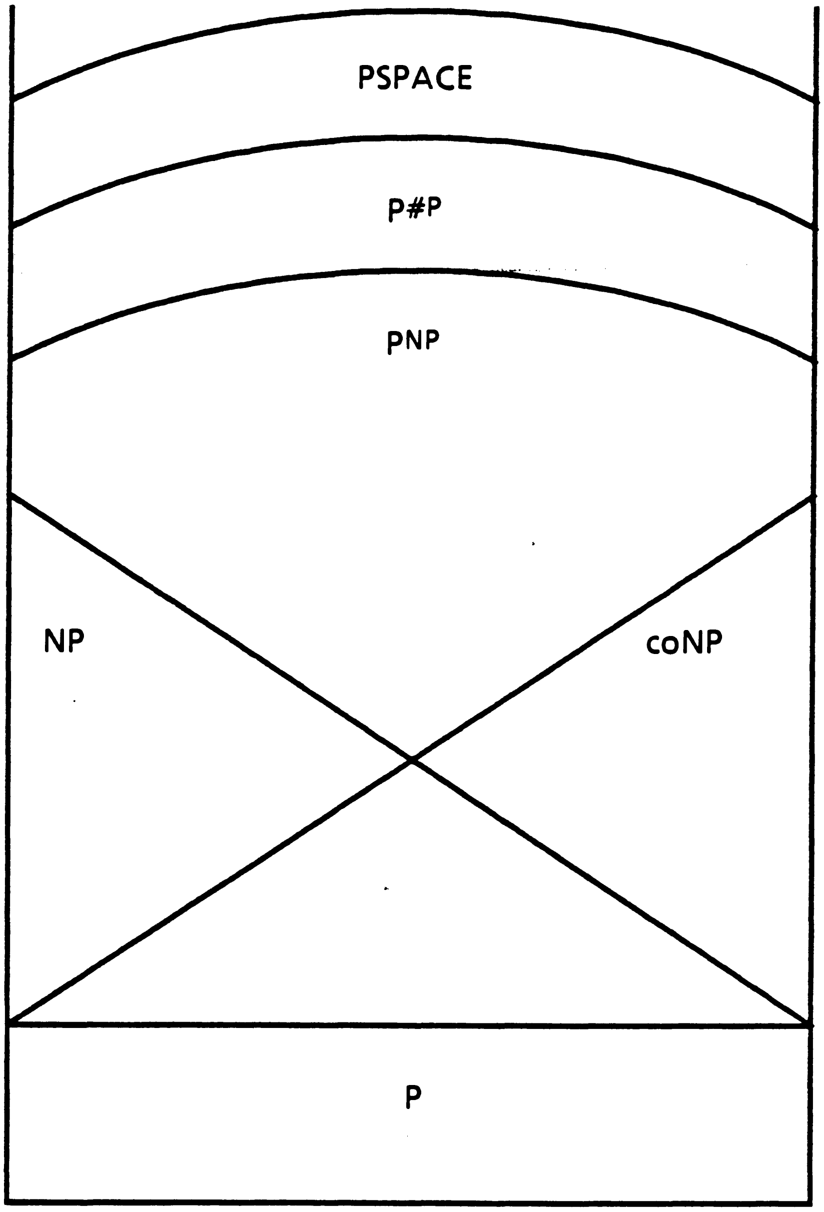


Figure B.4: The Structure of #P and the Polynomial Hierarchy

Bibliography

[Adl78] L. Adleman. Two theorems on random polynomial time. In *Proceedings 19th IEEE Symposium on Foundations of Computer Science*, pages 75–83, 1978.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AIK81] A. Adachi, S. Iwata, and T. Kasai. Low level complexity for combinatorial games. In *13th ACM Symposium on Theory of Computing*, pages 228–237, 1981.

[All85] E. Allender. Invertible functions. 1985. Ph.D. thesis, Georgia Institute of Technology.

[All86] E. Allender. The complexity of sparse sets in P. In *Proceedings 1st Structure in Complexity Theory Conference*, pages 1–11, Springer-Verlag *Lecture Notes in Computer Science #223*, June 1986.

[Ang80] D. Angluin. On counting problems and the polynomial-time hierarchy. *Theoretical Computer Science*, 12:161–173, 1980.

[APL80] L. Adleman, C. Pomerance, and Lenstra. On distinguishing prime numbers from composite numbers. In *Proceedings 21st IEEE Symposium on Foundations of Computer Science*, pages 387–406, 1980.

[AW85] M. Ajtai and A. Wigderson. Deterministic simulation of probabilistic constant depth circuits (preliminary version). In *Proceedings 26th*

IEEE Symposium on Foundations of Computer Science, pages 11–19, oct 1985.

[BB86] J. Balcázar and R. Book. On generalized Kolmogorov complexity. In *Proceedings of the 3rd Annual Symposium on Theoretical Aspects of Computer Science*, pages 334–340, 1986.

[BBL*84] J. Balcázar, R. Book, T. Long, U. Schöning, and A. Selman. Sparse oracles and uniform complexity classes. In *Proceedings 25th IEEE Symposium on Foundations of Computer Science*, pages 308–313, 1984.

[BD76] A. Borodin and A. Demers. *Some Comments on Functional Self-Reducibility and the NP Hierarchy.* Technical Report TR 76-284, Cornell Department of Computer Science, Ithaca, NY, July 1976.

[BG82] A. Blass and Y. Gurevich. On the unique satisfiability problem. *Information and Control*, 55:80–88, 1982.

[BGS75] T. Baker, J. Gill, and R. Solovay. Relativizations of the P=?NP question. *SIAM Journal on Computing*, 4(4):431–442, 1975.

[BH77] L. Berman and J. Hartmanis. On isomorphisms and density of NP and other complete sets. *SIAM Journal on Computing*, 6(2):305–322, 1977.

[BHH*82] R. Brayton, G. Hachtel, L. Hemachandra, A. Newton, and A. Sangiovanni-Vincentelli. A comparison of logic minimization strategies using ESPRESSO: an APL program package for partitioned logic minimalization. In *IEEE International Symposium on Circuits and Systems (ISCAS) 1982,* Volume 1, pages 42–48, 1982.

[BLS84] R. Book, T. Long, and A. Selman. Quantitative relativizations of complexity classes. *SIAM Journal on Computing*, 13(3):461–487, 1984.

[Boo74] R. Book. Tally languages and complexity classes. *Information and Control*, 26:186–193, 1974.

[Boo81] R. V. Book. Bounded query machines: on NP and PSPACE. *Theoretical Computer Science*, 15:27–39, 1981.

[Boo86] R. Book, editor. *Studies in Complexity Theory. Research Notes in Theoretical Computer Science*, John Wiley and Sons, 1986.

[Cai86] J. Cai. With probability one, a random oracle separates PSPACE from the polynomial-time hierarchy. In *18th ACM Symposium on Theory of Computing*, pages 21–29, 1986.

[CH86a] J. Cai and L. Hemachandra. The boolean hierarchy: hardware over NP. In *Proceedings 1st Structure in Complexity Theory Conference*, pages 105–124, Springer-Verlag *Lecture Notes in Computer Science #223*, June 1986.

[CH86b] J. Cai and L. Hemachandra. *Exact Counting is as Easy as Approximate Counting.* Technical Report TR86-761, Cornell Department of Computer Science, Ithaca, NY, June 1986.

[CHHS86a] J. Cai, J. Hartmanis, L. Hemachandra, and V. Sewelson. The boolean hierarchy I: structural properties. 1986. Submitted.

[CHHS86b] J. Cai, J. Hartmanis, L. Hemachandra, and V. Sewelson. The boolean hierarchy II: applications. 1986. Submitted.

[Chu36] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[Chu41] A. Church. *The Calculi of Lambda-Conversion. Annals of Mathematics Studies #6*, Princeton University Press, 1941.

[CKS81] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of ACM*, 26(1), 1981.

[Cob64] A. Cobham. The intrinsic computational difficulty of functions. In *Proc. 1964 International Congress for Logic Methodology and Philosophy of Science*, pages 24–30, North Holland, 1964.

[Coo71] S. Cook. The complexity of theorem-proving procedures. In *3rd ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[CT86] P. Clote and G. Takeuti. Exponential time and bounded arithmetic. In *Structure in Complexity Theory*, pages 125–143, Springer Verlag *Lecture Notes in Computer Science #223*, June 1986.

[Dav58] M. Davis. *Computability and Unsolvability*. Dover, 1958.

[Edm65] J. Edmonds. Paths, trees and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[EHSW87] A. El Gamel, L. Hemachandra, I. Shperling, and V. Wei. Using simulated annealing to design good codes. *IEEE Transactions on Information Theory*, IT-33(1):116–123, January 1987.

[FGJ*78] A. Fraenkel, M. Garey, D. Johnson, T. Schaefer, and Y. Yesha. The complexity of checkers on an NxN board—preliminary report. In *Proceedings 19th IEEE Symposium on Foundations of Computer Science*, pages 55–64, 1978.

[FHL80] M. Furst, J. Hopcroft, and E Luks. Polynomial-time algorithms for permutation groups. In *Proceedings 21st IEEE Symposium on Foundations of Computer Science*, pages 36–41, 1980.

[FR74] M. Fischer and M. Rabin. Super-exponential complexity of Presburger arithmetic. In R. Karp, editor, *Complexity of Computation*, Proceedings of SIAM-AMS Symposium in Applied Mathematics, 1974.

[Gil77] J. Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, 6(4):675–695, December 1977.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[GK86] S. Goldwasser and J. Kilian. Almost all primes can be quickly certified. In *18th ACM Symposium on Theory of Computing*, pages 316–329, 1986.

[God31] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[GS84] J. Grollmann and A. Selman. Complexity measures for public-key cryptosystems. In *Proceedings 25th IEEE Symposium on Foundations of Computer Science*, pages 495–503, 1984.

[GS85] A. Goldberg and M. Sipser. Compression and ranking. In *17th ACM Symposium on Theory of Computing*, pages 440–448, 1985.

[Har78] J. Hartmanis. *Feasible Computations and Provable Complexity Properties. CBMS-NSF Regional Conference Series in Applied Mathematics #30*, SIAM, 1978.

[Har85] J. Hartmanis. Solvable problems with conflicting relativizations. *Bulletin of the European Association for Theoretical Computer Science*, 27:40–49, October 1985.

[HE] L. Hemachandra and Euclid. GCDs and LCMs. In preparation.

[Hem86a] L. Hemachandra. *Can P and NP Manufacture Randomness?* Technical Report TR86-795, Cornell Computer Science Department, Ithaca, NY, December 1986.

[Hem86b] L. Hemachandra. *THE SKY IS FALLING: The Strong Exponential Hierarchy Collapses.* Technical Report TR86-777, Department of Computer Science, Cornell University, Ithaca, NY, August 1986.

[Hem87a] L. Hemachandra. On ranking. June 1987. To appear in *Proceedings 2nd Structure in Complexity Theory Conference*, IEEE Computer Society Press.

[Hem87b] L. Hemachandra. The strong exponential hierarchy collapses. To appear in *19th ACM Symposium on Theory of Computing.* May 1987.

[HH74] J. Hartmanis and H. Hunt. The LBA problem and its importance in the theory of computing. *SIAM-AMS Proceedings*, 7:1–26, 1974.

[HH86a] J. Hartmanis and L. Hemachandra. Complexity classes without machines: on complete languages for UP. In *Automata, Languages, and Programming (ICALP 1986)*, pages 123–135, Springer-Verlag *Lecture Notes in Computer Science #226*, July 1986.

[HH86b] J. Hartmanis and L. Hemachandra. On sparse oracles separating feasible complexity classes. In *STACS 1986: 3rd Annual Symposium on Theoretical Aspects of Computer Science*, pages 321–333, Springer-Verlag *Lecture Notes in Computer Science #210*, January 1986.

[HH87] J. Hartmanis and L. Hemachandra. One-way functions, robustness, and the non-isomorphism of NP-complete sets. June 1987. To appear in *Proceedings 2nd Structure in Complexity Theory Conference*, IEEE Computer Society Press.

[HI85] J. Hartmanis and N. Immerman. On complete problems for NP$\cap$coNP. In *Automata, Languages, and Programming (ICALP 1985)*, pages 250–259, Springer-Verlag *Lecture Notes in Computer Science #194*, 1985.

[HIS85] J. Hartmanis, N. Immerman, and V. Sewelson. Sparse sets in NP-P: EXPTIME versus NEXPTIME. *Information and Control*, 65(2/3):159–181, May/June 1985.

[Hof82] C. Hoffman. *Group-Theoretic Algorithms and Graph Isomorphism. Lecture Notes in Computer Science #136*, Springer-Verlag, 1982.

[HS65] J. Hartmanis and R. Stearns. On the computational complexity of algorithms. *Trans. AMS*, 117:285–306, 1965.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[HY84] J. Hartmanis and Y. Yesha. Computation times of NP sets of different densities. *Theoretical Computer Science*, 34:17–32, 1984.

[Imp87] Russell Impagliazzo. April 1987. Personal communication.

[JVV86] M. Jerrum, L. Valiant, and V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science*, 43(2,3):169–188, 1986.

[JY85] D. Joseph and P. Young. Some remarks on witness functions for non-polynomial and non-complete sets in NP. *Theoretical Computer Science*, 39:225–237, 1985.

[Kad87] J. Kadin. $P^{NP[\log]}$ and sparse Turing complete sets for NP. June 1987. To appear in *Proceedings 2nd Structure in Complexity Theory Conference*, IEEE Computer Society Press.

[KAI79] T. Kasai, A. Adachi, and S. Iwata. Classes of pebble games and complete problems. *SIAM Journal on Computing*, 8(4):574–587, 1979.

[Kar72] R. Karp. Reducibilities among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, 1972.

[Kel85] S. Kellogg. *Chicken Little.* W. Morrow, 1985.

[KL80] R. Karp and R. Lipton. Some connections between nonuniform and uniform complexity classes. In *12th ACM Sym. on Theory of Computing*, pages 302–309, 1980.

[KMR86] S. Kurtz, S. Mahaney, and J. Royer. Collapsing degrees. In *Proceedings 27th IEEE Symposium on Foundations of Computer Science*, pages 380–389, 1986.

[Ko85] K. Ko. Continuous optimization problems and a polynomial hierarchy of real functions. *Journal of Complexity*, 1:210–231, 1985.

[Lad75] R. Ladner. On the structure of polynomial time reducibility. *Journal of the ACM*, 22:155–171, 1975.

[Lev73] L. Levin. Universal sorting problems. *Problems of Information Transmission*, 9:265–266, 1973.

[LLS75] R. Ladner, N. Lynch, and A. Selman. A comparison of polynomial time reducibilities. *Theoretical Computer Science*, 1(2):103–124, 1975.

[Lon82] T. Long. A note on sparse oracles for NP. *Journal of Computer and System Sciences*, 24:224–232, 1982.

[Lon85] T. Long. On restricting the size of oracles compared with restricting access to oracles. *SIAM Journal on Computing*, 14(3):585–597, 1985.

[LS78] D. Lichtenstein and M. Sipser. GO is PSPACE hard. In *Proceedings 19th IEEE Symposium on Foundations of Computer Science*, pages 48–54, 1978.

[Mah80] S. Mahaney. Sparse complete sets for NP: solution of a conjecture of Berman and Hartmanis. In *Proceedings 21st IEEE Symposium on Foundations of Computer Science*, pages 54–60, 1980.

[Mah82] S. Mahaney. Sparse complete sets for NP: solution of a conjecture of Berman and Hartmanis. *Journal of Computer and System Sciences*, 25(2):130–143, 1982.

[MI82] S. Mahaney and N. Immerman. Oracles for which NP has polynomial size circuits. September 1982. Draft.

[MS72] A. Meyer and L. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proceedings of the 13th IEEE Symposium on Switching and Aut. Th.*, pages 125–129, 1972.

[MY85] S. Mahaney and P. Young. Reductions among polynomial isomorphism types. *Theoretical Computer Science*, 39:207–224, 1985.

[Pos46] E. Post. A variant of a recursively unsolvable problem. *Bulletin of the AMS*, 52:264–268, 1946.

[Rab76] M. Rabin. Probabilistic algorithms. In *Algorithms and Complexity*, Academic Press, 1976.

[Rac82] C. Rackoff. Relativized questions involving probabilistic algorithms. *Journal of the ACM*, 29(1):261–268, 1982.

[Rog67] H. Rogers, Jr. *The Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967.

[Rud87] Steven Rudich. Personal communication, March 1987.

[Sav70] W. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[Sav72] J. Savage. Computational work and time on finite machines. *Journal of the ACM*, 19:660–674, 1972.

[Sch83] U. Schöning. A low and a high hierarchy in NP. *Journal of Comp. and Sys. Sci.*, 27:14–28, 1983.

[Sch84] U. Schöning. Robust algorithms: a different approach to oracles. In *Automata, Languages, and Programming (ICALP 1984)*, pages 448–453, Springer-Verlag *Lecture Notes in Computer Science*, 1984.

[Sch86] U. Schöning. *Complexity and Structure*. Springer Verlag *Lecture Notes in Computer Science #211*, 1986.

[Scu62] Vincent Scully. *The Earth, the Temple, and the Gods*. Yale University Press, 1962.

[Sel79] A. Selman. P-selective sets, tally languages, and the behavior of polynomial time reducibilities on NP. *Mathematical Systems Theory*, 13:55–65, 1979.

[Sel82] A. Selman. Reductions on NP and P-selective sets. *Theoretical Computer Science*, 19:287–304, 1982.

[Sel86] A. Selman, editor. *Structure in Complexity Theory.* Springer Verlag *Lecture Notes in Computer Science #223*, June 1986.

[Sim75] J. Simon. On some central problems in computational complexity. January 1975. Ph.D. thesis, Cornell University, Ithaca, N.Y.

[Sim77] J. Simon. On the difference between one and many. In *Automata, Languages, and Programming (ICALP 1977)*, pages 480–491, Springer-Verlag *Lecture Notes in Computer Science #52*, 1977.

[SMB83] A. Selman, X. Mei-Rui, and R. Book. Positive relativizations of complexity classes. *SIAM Journal on Computing*, 12:565–579, 1983.

[SS83] J. Schwartz and M. Sharir. On the piano movers' problem: III. coordinating the motion of several independent bodies: the special case of circular bodies moving amidst polygonal barriers. *Int. Journal of Robotics Research*, 2(3):46–75, 1983.

[Ste79] J. Stein, editor. *The Random House Dictionary of the English Language: Unabridged Edition.* Random House, 1979.

[Sto77] L. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1977.

[Sto85] L. Stockmeyer. On approximation algorithms for #P. *SIAM Journal on Computing*, 14(4):849–861, November 1985.

[Tar83] R. Tarjan. *Data Structures and Network Algorithms. CBMS-NSF Regional Conference Series in Applied Mathematics #44*, SIAM, 1983.

[Tor86] L. Torenvliet. Structural concepts in relativized hierarchies. 1986. Ph.D. thesis, Universiteit van Amsterdam.

[Tur36] A. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society, ser. 2*, 42, 1936. Correction, *ibid*, vol. 43, pp. 554–546, 1937.

[Val76] L. Valiant. The relative complexity of checking and evaluating. *Information Processing Letters*, 5:20–23, 1976.

[Val79a] L. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[Val79b] L. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[Vaz87] V. Vazirani. March 1987. Personal Communication.

[VV85] L. Valiant and V. Vazirani. NP is as easy as detecting unique solutions. In *17th ACM Symposium on Theory of Computing*, pages 458–463, 1985.

[Wat87] O. Watanabe. May 1987. Personal Communication.

[WH81] D. Wallace and L. Hemachandra. Some properties of a probabilistic model for global wiring. In *Proceedings of the ACM IEEE 18^{th} Design Automation Conference*, pages 660–667, IEEE Computer Society, 1981.

[Wra77] C. Wrathall. Complete sets and the polynomial-time hierarchy. *Theoretical Computer Science*, 3:23–33, 1977.

[Yao85] A. Yao. Separating the polynomial-time hierarchy by oracles. In *Proceedings 26th IEEE Symposium on Foundations of Computer Science*, pages 1–10, 1985.

[Yap83] C. Yap. Some consequences of non-uniform conditions on uniform classes. *Theoretical Computer Science*, 26:287–300, 1983.

[Yes83] Y. Yesha. On certain polynomial-time truth-table reducibilities of complete sets to sparse sets. *SIAM Journal on Computing*, 12(3):411–425, 1983.