Contents

1 Introduction 3

2 P, NP and all that 5

3 Space complexity 11

4 Diagonalization 18

5 Polynomial Hierarchy 23

6 Circuits 26

Chapter 1

Introduction

The discovery of a formal model for describing algorithms —followed in short order by the invention of an implementation of this model in the form of the electronic computer— is one of the chief scientic discoveries of the 20th century. Interestingly, more than one formal model was conceived for describing algorithms, but in the end all were found to be equivalent (i.e., each could *simulate* the others). Today, many decades later, we take this equivalence for granted. After all, a new programming language (C, Java or any other) may ease the task of programming, but doesn't result in fundamentally different programs. In fact, a simple translator called the *compiler* can translate programs written in one computer language into another.

Once the equivalence of different computational models was realized, researchers could study computation in the abstract and focus on understanding its power. Research from the 1930s through the 1950s focused on understanding which algorithmic problems are *computable*. Many interesting algorithmic tasks were found to be *uncomputable* or *undecidable*: no computer can solve them without going into infinite loops (i.e., never halting) on certain inputs. This beautiful theory is usually taught in undergraduate courses and though it still has many interesting open problems, will not concern us here.

Rather, we will be interested in issues of *computational efficiency*, which, after the introduction of computers into scientific fields in the 1950s have increasingly occupied researchers. Consider the problem of multiplying two n-bit numbers, a basic task in scientific computing. Since the early 1960s it is known that the trivial algorithm —familiar from elementary school— is less efficient than a new, less intuitive algorithm based upon the Fast Fourier Transform (FFT). Many other such examples of counterintuitive yet efficient algorithms are known, and newer ones are discovered every year as part of a broad research program in algorithms design. Since the 1950s, interest also grew in a large class of optimization problems whose solutions seemed to require brute-force searches in a large space of possible solution. The effort to understand these problems resulted in the theory of *NP-completeness* in the 1970s.

The field of *computational complexity* concerns itself with these and related investigations. Broadly, it studies the following type of question: how efficiently can we solve a specific computational problem on a given computational model? Of course, the model ultimately of interest is the Turing machine (TM), or equivalently, any modern programming language such as C or Java. However, while trying to understand complexity issues arising in the study of the Turing machine, we often gain interesting insight by considering modifications of the basic Turing machine —nondeterministic, alternating and probabilistic TMs, circuits, quantum TMs etc.— as well as totally different computational models—communication games, decision trees, algebraic computation trees etc. Many beautiful results of complexity theory concern such models and their interrelationships.

Here we briefly recall the definition of the Turing machine; for a detailed discussion see an

undergraduate text such as Sipser's.

A multitape TM consists of

to be completed. Will emphasize the local nature of computation. Also defines nondeterminism.

We will assume that the TM uses the alphabet $\{0,1\}$. A *language* is a set of strings over $\{0,1\}$. The TM is said to *decide* the language if it accepts every string in the language, and rejects every other string.

Chapter 2

P, NP and all that

Now we define the classes P and NP. The question whether they are the same (the P versus NP problem) is the most important question in complexity theory and the motivation for much of what we will study.

We will usually use asymptotic notation in measuring the resources used by a TM. Let DTIME($t(n)$) consist of every language that can be decided by a deterministic multitape TM whose running time is $O(t(n))$ on inputs of size n. Let NTIME($t(n)$) be the class of languages that can be decided by a nondeterministic multitape TM (NDTM for short) in time $O(t(n))$.

DEFINITION 1 $P = \cup_{c \geq 0} DTIME(n^c)$.
$NP = \cup_{c \geq 0} NTIME(n^c)$

We believe that the class P is invariant to the choice of a computational model, since all "reasonable" computational models we can think of happen to be polynomially equivalent[1]. Namely, t steps on one model can be simulated in $O(t^c)$ steps on the other, where c is a fixed constant depending upon the two models. Thus in a very real sense, the class P exactly captures the notion of languages with "feasible" decision procedures. Of course, one may argue whether DTIME(n^{100}) represents "feasible" computation in the real world. However, in practice, whenever we show that a problem is in P, we usually find an n^3 or n^5 time algorithm, not an n^{100} algorithm.

NP contains decision problems in which a "YES" answer has a short certificate whose size is polynomial in the input length, and which can be verified deterministically in polynomial time. Formally, we have the following definition, which is easily seen to be equivalent to Definition 1.

DEFINITION 2 (ALTERNATIVE DEFINITION OF NP) *Language L is in NP if there is a language $L_0 \in P$ and constants $c, d > 0$ such that*

$$\forall x \in \{0,1\}^* \quad x \in L \Leftrightarrow \quad \exists y \in \{0,1\}^*, |y| \leq |x|^c + d \quad \textit{and} \quad (x, y) \in L_0.$$

EXAMPLE 1 The following languages are in NP:
CLIQUE = $\{< G, k >:$ graph G has a clique of size $\geq k\}$, where a clique is a subset of vertices in which there is an edge between every pair. (The "certificate" for membership in this language is the clique.)
3-COLOR = $\{G:$ graph G has a coloring with 3 colors$\}$, where a coloring is an assignment of Red, Blue, Green (or $1, 2, 3$) to each vertex such that no adjacent vertices get the same color.

These problems are not known to be in P.

[1] Recent results suggest that a computational model based upon quantum mechanics may not be polynomially equivalent to the Turing machine, though we do not yet know if this model is "reasonable" (i.e., can be built). We will discuss the quantum model later in the course.

The language $\{G : G \text{ is a connected graph}\}$ is in P, since a simple breadth first search can detect the connectivity of the graph.

A *polynomial-time reduction* from language A to language B is a polynomial-time computable function f mapping strings to strings, such that $x \in A$ if and only if $f(x) \in B$. If such a reduction exists we say A is *polynomial-time reducible to* B, denoted $A \leq_p B$. It is easily checked that polynomial-time reductions compose, namely, if $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$. (This transitivity property is usually the minimum one expects from any reasonable notion of reducibility. We will later see many other notions of reducibility.)

A language is *NP-hard* if there is a polynomial-time reduction from every NP language to it. An NP-hard language is *NP-complete* if it is NP-hard and in NP. The following statements follow from the definition.

THEOREM 1

1. *If $A \leq_p B$ and $B \in P$ then $A \in P$.*
2. *If language A is NP-complete, then $A \in P$ iff $P = NP$.*

Thus to study the P versus NP question it suffices to study whether any NP-complete problem can be decided in polynomial time. In this sense NP-complete problems capture the essence of NP. (Throughout the book we will encounter other complexity classes and will try to define a notion of complete problems for them in an effort to capture the essence of those classes.)

Do NP-hard problems exist? There are some trivial examples, for instance, the well-known *halting problem*, which is undecidable (i.e., there is no Turing machine that halts with the correct answer on every input)

$$A_{TM} = \{< M, w >: \text{ Turing machine } M \text{ halts when run on input } w\} \tag{2.1}$$

is NP-hard. (See Exercises.) Similarly, there are trivial examples of NP-complete languages, such as the following one:

$$K_{TM,poly} = \{< M, w, 1^n >: \text{NDTM } M \text{ accepts } w \text{ in time } n\}. \tag{2.2}$$

However, such NP-complete problems are not very useful because their definition is intimately tied to the notion of the Turing machine, and hence the fact they are NP-complete does not provide much new insight.

Around 1971, Cook and Levin in independent papers gave examples of combinatorial NP-complete problems whose definition seems to have nothing to do with Turing machines. (These papers actually also introduced the notion of NP-completeness.) A subsequent paper of Karp showed how to reduce 3SAT to a host of other combinatorial problems, and thus initiated the study of NP-completeness. To date, thousands of problems in a variety of disciplines have been found to be NP-complete.

Some of the simplest examples of NP-complete problems come from propositional logic. A 3CNF formula in propositional logic is a formula in some variables boolean variables $x_1, x_2, x_3, \ldots$ of the form

$$\bigwedge_i (y_{i_1} \vee y_{i_2} \vee y_{i_3}),$$

where each y_{i_j} is either a variable or its negation. (Following usual convention, we equate the boolean values True and False with the bits 1 and 0 respectively.)

3SAT is the set of boolean 3CNF formulae that have a satisfying assignment (that is, an assignment to the variables that makes it evaluate to true).

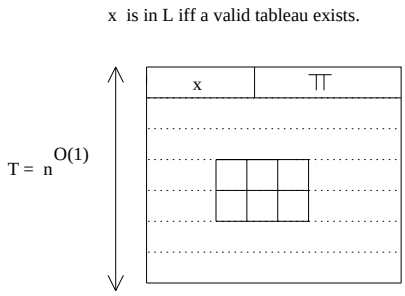

Figure 2.1: Tableau as used in Cook-Levin reduction

Both Cook and Levin showed that 3SAT is NP-complete. We briefly recall this classical reduction. Let L be an NP language and x be an input. The reduction uses the idea of a tableau, which is a step-by-step transcript whose ith line contains the state of the tape at step i of the computation. Clearly, $x \in L$ iff there exists a tableau of the computation of M that contains x in the first line and in which the last line shows that M accepts (Figure 1).

The main observation is that a Turing machine computation is very local. In going from one step to the next, nothing much changes on the tape except contents of the cell underneath the head. Thus the tableau represents a correct computation iff all 2×3 *windows* look "correct," i.e. thay satisfy some local consistency conditions. (If the head was part of the window, then the contents of that cell should have been modified according to the program of the machine, and if not then the cell contents should be unchanged.) The Cook-Levin reduction encodes these consistency checks with $3CNF$ clauses so that a valid tableau exists iff the set of all these $3CNF$ clauses is satisfiable.

2.1 coNP

The class coNP seems relevant to the study of the P versus NP question.

DEFINITION 3 $coNP = \left\{L : \overline{L} \in P\right\}$.

It is easy to check that the following definition is equivalent.

DEFINITION 4 (ALTERNATIVE DEFINITION OF CONP) *Language L is in coNP if there is a language $L_0 \in P$ and constants $c, d > 0$ such that*

$$\forall x \in \{0,1\}^* \quad x \in L \Leftrightarrow \quad \forall y \in \{0,1\}^*, |y| \leq |x|^c + d \quad \textit{and} \quad (x,y) \in L_0.$$

The key fact to note is the use of "$\forall$" in this definition where Definition 2 used $\exists$.

We can define coNP-completeness in analogy to NP-completeness: a language is coNP-complete if it is in coNP and every coNP language is polynomial-time reducible to it.

EXAMPLE 2 The following language is coNP-complete:

$$\text{TAUTOLOGY} = \{\varphi : \varphi \text{ is a boolean formula that is satisfied by every assignment}\}.$$

It is in coNP (the language L_0 is the set of pairs $(<\varphi>, s)$ where φ is a boolean formula and s is a string that is a satisfying assignment.

For every $L \in$ coNP we have to show that $L \leq_p$ TAUTOLOGY. But this is easy: just modify the Cook-Levin reduction from $\overline{L}$ (which is in NP) to SAT. For every input $x \in \{0,1\}^*$ that reduction produces a formula ϕ_x that is satisfiable iff $x \in \overline{L}$. Now consider the formula $\neg\phi_x$. It is in TAUTOLOGY iff $x \in L$, and this completes the description of the reduction.

2.2 EXPTIME and NEXPTIME

The following two classes are exponential time analogues of P and NP.

DEFINITION 5 $EXPTIME = \cup_{c \geq 0} DTIME(2^{n^c})$.
$NEXPTIME = \cup_{c \geq 0} NTIME(2^{n^c})$.

Note that P $\subseteq$ NP $\subseteq$ EXPTIME $\subseteq$ NEXPTIME. Is there any point to studying classes involving exponential running times? The following simple result —providing merely a glimpse of the rich web of relations we will be establishing between disparate complexity questions— may be a partial answer. It also introduces a technique called *padding*, whereby we "pad" every string in a language by a a string of (useless) symbols. This seemingly simple idea underlies many elementary results in complexity theory.

THEOREM 2
If EXPTIME $\neq$ NEXPTIME then P $\neq$ NP.

PROOF: We prove the contrapositive: assuming P = NP we show EXPTIME = NEXPTIME. Suppose $L \in$ NTIME(2^{n^c}). Then the following language

$$L_{\text{pad}} = \left\{ < x, 1^{2^{|x|^c}} > : \ x \in L \right\} \tag{2.3}$$

is in NP (in fact in NTIME(n)). Hence if P = NP then L_{pad} is in P. But if L_{pad} is in P then L is in EXPTIME: to determine whether an input x is in L, we just pad the input and decide whether it is in L_{pad} using the polynomial-time machine for L_{pad}. □

We note that most of the complexity classes studied later are, like P and NP, also contained in EXPTIME or NEXPTIME.

2.3 Decision versus search

We have chosen to define the notion of NP using Yes/No problems ("Is the given formula satisfiable?") as opposed to *search* problems ("Find a satisfying assignment to this formula if one exists"). However, it is easy to see that if we can solve the decision problem in time $T(n)$ then we can solve the search problem in time $n^c T(n)$ for some small constant c. This fact is true for all NP-complete languages but let us illustrate it using 3SAT.

Suppose A is an algorithm that decides 3SAT in time $T(n)$. Given a 3CNF formula φ with n variables, we can use A to first check that φ is satisfiable. If so, we substitute $x_1 = 0$ and $x_1 = 1$ in φ (this simplifies and shortens the formula a little and leaves a formula with $n-1$ variables) and check which of the two is satisfiable (it is possible that both are). Say it is the first. Then we fix $x_1 = 0$ from now on and continue with the simplified formula. Continuing this way we end up fixing all n variables while ensuring that each intermediate formula is satisfiable. Thus the final assignment to the variables satisfies φ. Clearly, the running time is $n^2 T(n)$ or so.

Formally, we say that 3SAT is *downward self-reducible*: the solution of instances with n variables reduces to the solution of instances with $n-1$ variables.

2.4 List of some problems in P and NP

Will contain a list of problems in each class, including linear programming, primality etc.

NP optimization problems

Why 3SAT?

The reader may wonder why complexity theorists are so fond of 3SAT. One answer is that it is useful for proving the NP-completeness of other problems: it has very minimal combinatorial structure and thus easy to use in reductions. Another answer has to do with history: propositional logic has had a central role in mathematical logic —in fact it was exclusively the language of classical logic (e.g. in ancient Greece). This historical resonance is one reason why Cook and Levin were interested in 3SAT. A third answer has to do with practical importance: it is a simple example of *constraint satisfaction problems*, which are ubiquitous in many fields including artificial intelligence.

What if P = NP?

If P = NP —specifically, if an NP-complete problem like 3SAT had a very efficient algorithm running in say $O(n^2)$ time— then the world would be mostly a Utopia. Mathematicians could be replaced by efficient theorem-discovering programs (a fact first pointed out by Kurt Gödel in 1955). This follows from the fact that the following problem is in NP:

$$\text{THEOREMS} = \{(\varphi, 1^n) : \varphi \text{ is a mathematical statement with a formal proof of length} \leq n \text{ in axiomatic set theory}\}.$$

(The "axiomatic set theory" could be replaced by any other mathematical or scientific area with an axiomatic description. The exact choice of axioms for doesn't matter so long as they have a succinct description, which currently accepted axiomatic systems do.)

AI software would be perfect since we could easily do exhaustive searches in a large tree of possibilities. Inventors and engineers would be greatly aided by software packages that can design the perfect part or gizmo for the job at hand. VLSI designers will be able to whip up optimum circuits, with minimum power requirements.

Somewhat intriguingly, this Utopia would have no need for randomness. As we will later see, if P = NP then randomized algorithms would buy essentially no efficiency gains over deterministic algorithms. (Armchair philosophers should ponder this.)

This Utopia would also come at one price: there would be no privacy in the digital domain. Any encryption scheme would have a trivial decoding algorithm. There would be no digital cash, no PGP, no RSA (as we will later see). We would just have to learn to get along better without these, folks.

Exercises

§1 Show that the languages defined in (2.1) and (3.1) are NP-hard and NP-complete respectively.

§2 If P = NP then there is a polynomial time decision algorithm for 3SAT. Show that in fact if P = NP then there is also a polynomial time algorithm that, given any 3CNF formula, produces a satisfying assignment if one exists.

§3 We have defined a relation $\leq_p$ among languages. We noted that it is *Reflexive* (that is, $A \leq_p A$ for all languages A) and *transitive* (that is, if $A \leq_p B$ and $B \leq_p C$ then $A \leq_p C$. Show that it is not commutative, namely, $A \leq_p B$ need not imply $B \leq_p A$.

§4 Mathematics can be axiomatized using for example the *Zermelo Frankel* system, which has a finite description. Show that the following language is NP-complete.

$$\{< \varphi, 1^n >: \text{ math statement } \varphi \text{ has a proof of size at most } n \text{ in the ZF system}\}.$$

(Hints: Why is this language in NP? Is boolean satisfiability a mathmatical statement?) Conclude that if P = NP then mathematicians can be replaced by polynomial-time Turing machines.

§5 Can you give a definition of NEXPTIME analogous to the definition of NP in Definition 2? Why or why not?

§6 We say that a language is NEXPTIME-complete if it is in NEXPTIME and every language in NEXPTIME is polynomial-time reducible to it. Describe a NEXPTIME-complete language.

§7 Show that if $L \in \text{NTIME}(t(n))$ then we can give a reduction from L to 3SAT that transforms instances of size n into 3CNF formulae of size $O(t(n) \log t(n))$. (Hint: The representation of the tableau used in our description of the Cook-Levin reduction is very inefficient, since all the action happens only near the machine's head.)

§8 *Need to have some NP-completeness questions here*

Chapter 3

Space complexity

This chapter studies space-bounded computation, which has to be performed within a restricted number of tape cells. (The intention is to study the memory requirements of computational tasks.) A space-bounded machine has two tapes. The input is provided on a read-only tape, which, as the name suggests, can only be read and not overwritten. The machine also has a read-write *work tape*. We say that the machine runs in $S(n)$ space if it can perform its computations using $O(S(n))$ cells of the work tape whenever the input has n bits. (See Figure 3.1.) We denote by $\mathrm{SPACE}(S(n))$ the class of languages that can be decided in $O(S(n))$ space. We will restrict attention to $S(n) \geq \log n$, so the machine has enough space to maintain a pointer into the input tape. (Machines that use less than $\log n$ space have been studied but will not concern us.) Note that $\mathrm{DTIME}(t(n)) \subseteq \mathrm{SPACE}(t(n))$ since a TM can access only one tape cell per step. Also, $\mathrm{SPACE}(s(n)) \subseteq \mathrm{DTIME}(2^{O(s(n))})$, since a machine with a work tape of size $s(n)$ only has $O(n \cdot 2^{O(s(n))}) = 2^{O(s(n))}$ different configurations (reason: once we fix the input on the read-only tape, the machine's configuration is completely described by the position of its head on the input tape, the machine's inner state, and the contents of the work tape) and the machine cannot enter the same configuration twice because that would mean it is in an infinite loop. (Recall that the machine is required to halt on every input.) It is also easy to construct space-bounded machines that do run in $2^{S(n)}$ time: for instance the machine that uses its work tape to maintain a counter which it increments from 1 to $2^{S(n)-1}$ and then halts. However, it is an open problem whether there is a language in $\mathrm{DTIME}(2^{O(s(n))})$ for some $S(n) > \log n$ that is not in $\mathrm{SPACE}(S(n))$.

We can similarly define nondeterministic space-bounded machines, and the class $\mathrm{NSPACE}(s(n))$. The following definitions are similar to the definitions of P and NP.

DEFINITION 6 $PSPACE = \cup_{c>0} SPACE(n^c)$.
$NPSPACE = \cup_{c>0} NSPACE(n^c)$.

Note that $\mathrm{NP} \subseteq \mathrm{PSPACE}$, since polynomial space —in fact, linear space— is enough to decide 3SAT (just use the linear space to cycle through all 2^n assignments in order, where n is the number of variables, and note that once we have checked an assignment we can erase it from the work tape and reuse the work tape to check the next assignment).

Space-bounded computation continues to be interesting for $S(n) < n$, of course, and the following classes are the most interesting in this context.

DEFINITION 7 $L = SPACE(\log n)$.
$NL = NSPACE(\log n)$.

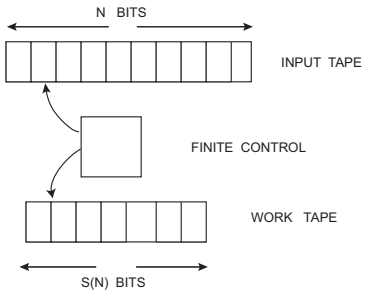

Figure 3.1: A TM running in space $O(S(n))$

EXAMPLE 3 The following languages are in L:

$$\text{EVEN} = \{x : x \text{ has an even number of 1s}\}.$$

$$\text{MULT} = \{(< n >, < m >, < nm >) : n \in \mathbf{N}\}.$$

In fact, it seems pretty difficult to identify any nontrivial computations that belong in L. Even so, we cannot currently rule out that 3SAT $\in$ L (in other words —see the exercises— it is open whether NP = L).

3.1 PSPACE completeness

As already indicated, we do not know if P $\neq$ PSPACE, though we strongly believe that the answer is YES. Note in particular that P = PSPACE implies P = NP.

Recall that complete problems can help capture the essence of a complexity class. Now we define complete problems for PSPACE.

DEFINITION 8 *A language A is* PSPACE-hard *if for every $L \in$ PSPACE, $L \leq_p A$. If in addition $A \in$ PSPACE then A is* PSPACE-complete.

Clearly, a PSPACE-complete language is in P iff P = PSPACE, so such a language may be viewed as the "most difficult" problem of PSPACE.

As in the case of NP, it is trivial to describe a PSPACE-complete problem. The following is one (Exercise):

$$K_{TM,pspace} = \{< M, w, 1^n >: \text{NDTM } M \text{ accepts } w \text{ in space } n\}. \tag{3.1}$$

Now we describe a more nontrivial complete problem. A *quantified boolean formula* has the form $Q_1 x Q_2 x_2 \cdots Q_n x_n \varphi(x_1, x_2, \ldots, x_n)$ where each Q_i is one of the two quantifiers $\forall$ or $\exists$ and φ is a boolean formula.

Let TQBF be the set of quantified boolean formulae that are true.

EXAMPLE 4 The formula $\forall x \exists y\ (x \wedge y) \vee (\overline{x} \wedge \overline{y})$ is in TQBF but $\forall x \forall y\ (x \wedge y) \vee (\overline{x} \wedge \overline{y})$ is not in TQBF. Note that here $\forall$ and $\exists$ quantify over the set $\{0, 1\}$.

Clearly, TQBF $\in$ PSPACE. The algorithm needs to explore the tree of possibilites obtained by all possible substitutions to the variables. A moment's thought shows that linear space suffices, since at any time the algorithm only needs to keep one value for each of the n variables on tape.

THEOREM 3
TQBF is PSPACE-complete.

PROOF: Let L be any PSPACE language and M be a machine that decides it in n^c space. We show how to construct a quantified boolean formula of size roughly $O(n^{2c})$ that is true iff M accepts x. The construction is reminiscent of the proof of the Cook-Levin theorem, except it has to take into account the fact M's computation could run in $2^{O(n^c)}$ time, so the number of variables is too few to represent a tableau directly.

For notational convenience, we will use vectors of variables. A *variable vector* C has $O(n^c)$ variables whereby substituting boolean values gives a potential configuration of M. (Many substitutions —in fact, most— may yield nonsensical configurations.) An *assignment vector* D is an assignment of boolean values to a variable vector.

Restatement of Cook-Levin construction: For any variable vectors C_1, C_2 we can write in $O(n^{c+1})$ time an unquantified boolean formula $\varphi(C_1, C_2)$ such that for any assignment vectors D_1, D_2, $\varphi(D_1, D_2) = 1$ iff D_1, D_2 correspond to valid configurations of M and D_2 can be obtained from D_1 in one step.

The main idea in the current proof is as follows. We show how to write for any $T > 0$ a quantified formula $\psi_T(C_1, C_2)$ whose free —i.e. unquantified— variables are C_1, C_2 (the formula may have many other variables but they are all quantified) such that for any assignment vectors D_1, D_2, $\psi_T(D_1, D_2)$ is true iff D_1, D_2 correspond to valid configurations of M and D_2 can be obtained from D_1 in $\lceil T \rceil$ steps.

Clearly, if $T = 1$, then $\psi_1(C_1, C_2) = \varphi(C_1, C_2)$ fits the bill. For general T we would be tempted to write

$$\psi_T(C_1, C_2) = \exists C_3 \psi_{T/2}(C_1, C_3) \wedge \psi_{T/2}(C_3, C_2). \tag{3.2}$$

The resulting formula does the job, except in one respect: its length is $O(Tn^c)$ whereas we were hoping the length will be $O(n^c \log T)$.

The correct expression is as follows

$$\psi_T(C_1, C_2) = \exists C_3 \forall (C_4, C_5) \qquad ((C_4, C_5) = (C_1, C_3) \vee (C_4, C_5) = (C_3, C_2)) \Rightarrow \psi_{T/2}(C_4, C_5). \tag{3.3}$$

The reduction consists of the following. First, compute an upperbound $T = 2^{dn^c}$ for an appropriate constant d (derived from the number of distinct configurations of M) on the running time of M on input x. Then construct the quantified formula $\psi_{2^{n^c}}(C_1, C_2)$. Finally, substitute values for the free variables C_1, C_2 to obtain $\psi_{2^{n^c}}(D_{start}, D_{accept})$, where D_{start} is the assignment vector corresponding to the start configuration (assume the machine is started with both heads at the left end of the tape) and D_{accept} is the assignment vector corresponding to the accept configuration, which we assume is unique (just modify the machine's description so that it erases fills the work tape with 0's and moves both heads to the tapes' left ends before entering the accept state).

The resulting formula is not strictly a quantified boolean formula, on two counts. First, our definition of quantified boolean formulae requires all variables to be quantified before the boolean formula begins, whereas here (after unwrapping the recursive definition) we find quantifiers in the midst of the boolean formula. But it is an easy exercise to move these quantifiers to the beginning. Second, for ease of exposition we used the nonstandard boolean connectives $\Rightarrow$ and $=$ in the formula, but they can be easily expressed using the standard boolean connectives $\vee, \wedge, \neg$ by noticing that $p \Rightarrow q$ is equivalent to $\overline{p} \vee q$ and $p = q$ is equivalent to $(p \wedge q) \vee (\neg p \wedge \neg q)$.

The reduction applies these transformations on $\psi_{2^{n^c}}(D_{start}, D_{accept})$ to obtain a quantified boolean formula, and outputs it. □

The essence of PSPACE: optimum strategies for game-playing. Recall that the central feature of NP-complete problems is that a yes answer has a short certificate. In fact, this was essentially the definition of NP. The analogous unifying concept for PSPACE-complete problems seems to be that of a winning strategy for a 2-player game with perfect information. A good example of such a game is Chess: two players alternately make moves, and the moves are made on a board visible to both. Thus moves have no hidden sideeffects; hence the term "perfect information." Finding whether or not the first player has a winning strategy involves searching the tree of all possible moves. Note that the first player has a winning strategy iff there is a 1st move for player 1 such that for every possible 1st move of player 2 there is a 2nd move of player 1 such that.... (and so on) such that at the end player 1 wins. The interplay of existential and universal quantifiers in the previous line suggests the following example of a game.

EXAMPLE 5 (THE QBF GAME) The "board" for the QBF game is a boolean formula φ whose free variables are $x_1, x_2, \ldots, x_{2n}$. The two players alternately make moves, which involve picking values for $x_1, x_2, \ldots$, in order. Thus player 1 will pick values for the odd-numbered variables $x_1, x_3, x_5, \ldots$ (in that order) and player 2 will pick values for the even-numbered variables $x_2, x_4, x_6, \ldots,$. We say player 1 wins iff at the end φ becomes true.

Clearly, player 1 has a winning strategy iff

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \cdots \forall x_{2n} \varphi(x_1, x_2, \ldots, x_{2n}),$$

namely, iff this quantified boolean formula is true.

Thus deciding whether player 1 has a winning strategy in the QBF game is PSPACE-complete.

Many other games —including an $n \times n$ version of chess— are PSPACE-complete (see Sipser or Garey and Johnson).

3.2 NL completeness

What kind of reduction should we use to define completeness for NL? We cannot use the polynomial-time reductions since NL $\subseteq$ P. Specifically, a polynomial-time reduction could just solve the NL problem as part of the reduction. To give an example, every language in NL is polynomial-time reducible to the trivial language $\{0,1\}$ (reduction: "decide whether or not the input is in the NL language, and then map to 1 or 0 accordingly"). Clearly, such trivial languages should not be complete under any useful definition of reduction.

Instead, we use *logspace* reductions. A *logspace transducer* is a machine that has three tapes: a read-only tape containing an input, a read-write work tape whose length is $O(\log n)$ when the input has n bits, and a write-only *output tape*. For any input, the machine computes in poly(n) time (remember, a logspace machine cannot take any more than poly(n) time if it has to avoid looping forever) and during its computation it writes a string on its output tape. Thus such a transducer computes a function $f : \{0,1\}^* \to \{0,1\}^*$.

DEFINITION 9 *Language A is* logspace reducible *to language B, denotes $A \leq_l B$, if there is a function $f : \{0,1\}^* \to \{0,1\}^*$ computable by a logspace transducer such that for every string $x \in \{0,1\}^*$, $x \in A$ iff $f(x) \in B$.*

Logspace reducibility satisfies usual properties one expects (see Exercises). (a) If $A \leq_l B$ and $B \leq_l C$ then $A \leq_l C$ (b) If $A \leq_l B$ and $B \in$ L then $A \in$ L.

DEFINITION 10 *Language A is NL-*complete *if it is in NL and every NL language is logspace reducible to it.*

THEOREM 4
PATH is NL-complete.

PROOF: We have already seen that PATH is in NL. Let L be any language in NL and M be a machine that decides it in space $O(\log n)$. We describe a logspace transducer that reduces L to PATH.

For any input x of size n, the transducer outputs a graph $G_{M,x}$ which is the configuration graph of M on x. This is a directed graph whose nodes are all possible $2^{O(\log n)}$ configurations of the machine on input x, and there is an edge from configuration S_i to configuration S_j iff the machine can go from S_i to S_j in one step. In this graph there is a path from the start configuration to the accepting configuration (assume there is a unique accepting configuration, by modifying the machine so that it cleans up its work tape before entering the accept state) iff M accepts x.

The transducer outputs the set of configurations one by one, and for each configuration it outputs all edges going out of that configuration (the number of such edges is some constant depending upon M). Clearly, this is possible in $O(\log n)$ space. □

3.3 Two suprising algorithms

Now we describe two surprising algorithms for space-bounded computation. Let PATH be the language

$$\text{PATH} = \{< G, s, t >: \ G \text{ is a directed graph in which there is a path from } s \text{ to } t\} \tag{3.4}$$

PATH has a trivial polynomial time algorithm that uses depth-first search; this algorithm uses linear space and almost linear time. Can we decide PATH in sublinear space? This seems difficult, since intuition says that a search algorithm must store information about every node it has visited. Note however that PATH $\in$ NL, since a nondeterministic machine can take a "nondeterministic walk" starting at s, always maintaining the index of the vertex it is at, and using nondeterminism to select a neighbor of this vertex to go to next. The machine accepts iff the walk ends at t in at most n steps, where n is the number of nodes. If the nondeterministic walk has run for n steps already and t has not been encountered, the machine rejects. The work tape only needs to hold $O(\log n)$ bits of information at any step, namely, the number of steps that the walk has run for, and the identity of the current vertex.

The next theorem says that this computation can be made deterministic with quadratic blowup in space. (Note that the computation no longer runs in poly(n) time, however, and could take $2^{O(\log^2 n)} = n^{O(\log n)}$ time.)

THEOREM 5 (SAVITCH)
PATH $\in$ *SPACE*$(\log^2 n)$.

PROOF: Let $< G, s, t >$ be the input. We describe a recursive procedure REACH?(u, v, l) that returns "YES" if there is a path from u to v of length at most l and "NO" otherwise. The procedure uses the observation that if there is a path from u to v of length at most l, there is a "middle" node z along this path such that there is a path of length $\lceil l/2 \rceil$ from u to z and $\lfloor l/2 \rfloor$ from z to v.

The procedure is as follows. *If $l = 1$, return YES iff (u, v) is an edge. Otherwise for each node z, run* REACH?*$(u, z, \lceil l/2 \rceil)$ and* REACH?*$(u, z, \lfloor l/2 \rfloor)$ and return YES if both return YES.*

The main observation is that the second invocation of the REACH procedure can run in the same space used by the first one. Also, keeping track of the current value of z takes only

$O(\log n)$ space. Thus if $S(l)$ denotes the space requirement, we have

$$S(l) \leq O(\log n) + S(\lceil \frac{l}{2} \rceil). \tag{3.5}$$

This yields $S(l) = O(\log n \log l)$.

The final answer is REACH?(s, t, n), so the space required is $S(n) = O(\log^2 n)$. □

COROLLARY 6
For every $S(n) > \log n$, $NSPACE(S(n)) \subseteq SPACE(S(n)^2)$.

PROOF: Similar to the above proof, where the "graph" in question is the configuration graph of the nondeterministic machine. Let M be a nondeterministic machine running in space $S(n)$ and x be an input of length n. Then the configuration graph is a directed graph whose nodes are all possible $2^{O(S(n))}$ configurations of the machine on input x, and there is an edge from configuration S_i to configuration S_j iff the machine can go from S_i to S_j in one step. In this graph there is a path from the start configuration to the accepting configuration (assume there is a unique accepting configuration, by modifying the machine so that it cleans up its work tape before entering the accept state) iff M accepts x.

Note also that the algorithm in the proof of Theorem 5 requires access to the input graph only to verify for pairs of nodes (u, v) whether $u \to v$ is an edge. In the current situation u, v are configurations and the machine can check for itself whether there is an edge from one to the other in the configuration graph. □

The next result concerns $\overline{\text{PATH}}$, the complement of PATH. A decision procedure for this language must accept when there is no path from s to t in the graph. The following result is the second surprising algorithm of this section.

THEOREM 7 (IMMERMAN-SZLEPCSENYI)
$\overline{PATH} \in NL$.

PROOF: For notational ease we assume the nodes are numbered from 1 to n and $s = 1, t = n$. We have to describe a nondeterministic computation running in $O(\log n)$ space that accepts iff there is no path from s to t. Let us first consider an easier problem. Suppose somebody gives the machine a number c, which is exactly the number of nodes reachable from s. (This count is ≥ 1, since s is reachable from s.) Then a little thought suggests that the task becomes easier. The machine enumerates nodes one by one from 1 to $n - 1$ and tries to guess —using a nondeterministic walk —a path of length at most n from s to that node. It keeps a count of how many nodes it succeeded for. It accepts at the end iff it succeeds for c nodes, since that means that t is not one of the c nodes reachable from s. Note on the other hand that if every branch of this nondeterministic computation fails (i.e., does not accept) then there do not exist c nodes different from t that are reachable from s, which means that then t must be reachable from s.

Now we can describe the nondeterministic computation for $\overline{PATH}$. We use an *inductive counting* technique to calculate c, where step i determines c_i, the number of nodes reachable from s in i steps. Thus c_n is the same as c in the above description. (When we say that a nondeterministic computation "computes" a number, we require branches to either HALT without an answer, or to output the correct answer. Also, at least one branch has to output the correct answer.)

Let C_i be the set of nodes reachable from s using a path of length at most i. Thus $c_i = |C_i|$.

NONDETERMINISTIC PROCEDURE TO COMPUTE $c_{i+1} - c_i$ GIVEN c_i.
Maintain a counter called `count` that is initialized to 0.
Do the following for each node j from 1 to n.

1. (*Verify that $j \notin C_i$:*)
   Enumerate all nodes from 1 to n except j and try to guess a nondeterministic walk —whose length is also nondeterministically picked from 1 to i— from s to that node of length at most i. Keep track of the number of nodes for which this succeeds, and if this number equals c_i, then we have managed to verify that $j \notin C_i$. If so, continue, else HALT and reject. □

2. (*Verify that $j \in C_{i+1} \setminus C_i$*)
   Take a nondeterministic walk of length $i+1$ from s. If the walk ends at j, then clearly $j \in C_{i+1} \setminus C_i$ and so increment `count` by 1. If the walk does not end at j, then HALT.

Clearly, if the machine has not yet halted, then `count` contains $c_{i+1} - c_i$ at the end. Also, it is clear from the description that only $O(\log n)$ cells are needed for storage.

Using the notion of the configuration graph as in Corollary 6 we can modify the proof of Theorem 7 to prove the following.

COROLLARY 8
For every $s(n) > \log n$ $NSPACE(s(n)) = coNSPACE(s(n))$.

Picture of Space-Bounded Complexity Classes.
The following is our understanding of space-bounded complexity.

$$\text{DTIME}(s(n)) \subseteq \text{SPACE}(s(n)) \subseteq \text{NSPACE}(s(n)) = \text{coNSPACE}(s(n)) \subseteq \text{DTIME}(2^{O(s(n))}).$$

None of the inclusions are known to be strict though we believe they are.

Exercises

§1 In analogy with the characterization of NP in terms of certificates, show that we can define NSPACE($s(n)$) as the set of languages for which certificates can be checked deterministically using $O(S(n))$ space, where the certificate is provided on a read-only tape of size $2^{O(S(n))}$ and the verifier can scan it only once from left to right.

§2 Show that (a) If $A \leq_l B$ and $B \leq_l C$ then $A \leq_l C$ (b) If $A \leq_l B$ and $B \in$ L then $A \in$ L.

§3 Show that the following language is NL-complete:

$$\{< G >: G \text{ is a strongly connected digraph}\}.$$

§4 Suppose we define NP-completeness using logspace reductions instead of polynomial-time reductions. Show (using the proof of the Cook-Levin Theorem) that SAT, 3SAT continue to be NP-complete under this new definition. Conclude that SAT $\in$ L iff NP = ł.

Chapter 4

Diagonalization

To separate two complexity classes we need to exhibit a machine in one class that is different (namely, gives a different answer on some input) from *every* machine in the other class. This lecture describes *diagonalization*, essentially the only general technique known for constructing such a machine. We also indicate why this technique has been unable thus far to resolve P =? NP and other interesting questions.

4.1 Time Hierarchy Theorem

The Time Hierarchy Theorem shows that allowing Turing Machines more computation time strictly increases the class of languages that they can decide. We say that a function $f : \{0,1\}^* \to \mathbf{N}$ is a *time-constructible* function if there is a Turing machine that, given an input with n bits, writes down $f(n)$ on its tape in $f(n)$ time. Usual functions like $n \log n$ or n^2 satisfy this property, and we will restrict attention to running times that are time-constructible.

THEOREM 9
If f, g are time-constructible functions satisfying $f(n) \log f(n) = o(g(n))$, then

$$DTIME(f(n)) \subsetneq DTIME(g(n)) \tag{4.1}$$

To showcase the essential idea of the proof of Theorem 9, we prove the simpler statement $\mathrm{DTIME}(n) \subsetneq \mathrm{DTIME}(n^2)$. We use diagonalization. Suppose $M_1, M_2, M_3, \ldots$ is a numbering of all Turing Machines, where the description of M_i can be produced from i in time $O(i)$. (Such numberings exist. For example, one can order TMs according to the number of states in their transition diagrams, and use lexicographic ordering among then all TMs that have the same number of states. Note that we allow the numbering to include machines that may not halt on all inputs.)

Consider the following Turing Machine, D: "*On input x, if $x = 0^j 1^k$ for some j, k then construct M_k and simulate it on x for $|x|^{1.5}$ steps. If M_k halts and accepts during this simulation, reject. If M_k halts and rejects, accept. If M_k does not halt in this time, accept.*"

This machine runs in time at most $2n^2$. Specifically, it has to maintain a timer that keeps tracks of the number of steps in the simulation of M_k. Maintaining this counter introduces an overhead (left to the reader) so the running time of the modified machine will be $O(n^{1.5} \log n) = o(n^2)$.

Now we show that language accepted by D is not in $\mathrm{DTIME}(n)$. Suppose M_k is a machine that runs in linear time, specifically, in time at most $cn + d$. Then for every integer j satisfying $j + k > \max\{(c+1)^2, d\}$, the input $y = 0^j 1^k$ is such that D's computation on this input

involves simulating M_k on the same input and flipping the answer. Since $|y|^{1.5} > c|y| + d$, the diagonalizer has enough time to complete its simulation of M_k and determine its answer.

The proof of Theorem 9 is similar, and involves the observation that the diagonalizing Turing machine D wants to simulate a machine that runs in time $f(n)$, and also maintain a counter to keep track of the running time. The overhead of maintaining the counter increases the running time of D by a factor $O(\log f(n))$.

4.2 Space Hierarchy Theorem

The space hierarchy theorem is completely analogous to the time hierarchy theorem. One restricts attention to *space-constructible* functions, which are functions $f : \mathbf{N} \to \mathbf{N}$ for which there is a machine that, given any n-bit input, constructs $f(n)$ in space $O(f(n))$. The proof of the next theorem is completely analogous to that of Theorem 9.

THEOREM 10
If f, g are space-constructible functions satisfying $f(n) = o(g(n))$, then

$$SPACE(f(n)) \subsetneq SPACE(g(n)) \tag{4.2}$$

4.3 Nondeterministic Time Hierarchy Theorem

The analogous hierarchy theorem for nondeterministic computation is even tighter than for deterministic computation: it drops the extra log term.

TBE later: why no extra log term?

THEOREM 11
If f, g are running times satisfying $f(n) = o(g(n))$, then

$$NTIME(f(n)) \subsetneq NTIME(g(n)) \tag{4.3}$$

Again, we just showcase the main idea of the proof by proving $\text{NTIME}(n) \subsetneq \text{NTIME}(n^3)$. The technique from the previous section does not directly apply. A nondeterminisitic machine that runs in $O(n)$ time may have $2^{O(n)}$ branches in its computation. It is unclear how to determine in $O(n^2)$ time whether or not it accepts and then flip this answer. Instead we use a technique called *lazy* diagonalization, which is only guaranteed to flip the answer on some input in a fairly large range.

For a pair of integers i, j, define $f(i, j)$ to be

$$f(i, j) = 2^{2^{2^{2^i} 3^j}}. \tag{4.4}$$

Clearly, f is one-to-one. We say $(i, j) \precsim (k, l)$ if $f(i, j) < f(k, l)$. Then $\precsim$ is a linear ordering. Thus for any integer n, there is a largest pair i, j and smallest pair k, l such that $f(i, j) < n \leq f(k, l)$. We use the shorthands $L_n = f(i, j), H_n = f(k, l)$. Note that L_n, H_n are easily computed given n, certainly in $O(n^2)$ time. Furthermore, $H_n > 2^{(L_n)^2}$, so the interval $[L_n, H_n)$ is quite large. The diagonalizing machine tries to flip the answer in such large intervals.

Let $M_1, M_2, M_3, \ldots$ be an enumeration of all NDTMs. We construct the following NDTM D_1.

"*On input x, if $x \notin 1^*$, accept. If $x = 1^n$, then compute L_n, H_n. Let $H_n = (i, j)$. If $n = 1^{H_n}$, accept 1^n iff M_i rejects 1^{L_n+1} in $|L_n|^{1.5}$ time. (Note that this requires going through all possible* $\exp(|L_n|^{1.5})$ *nondeterministic branches of M_i on input 1^{L_n+1}, but that is fine since the input size H_n exceeds even $2^{(L_n)^2}$.) Otherwise simulate M_i on input 1^{n+1} using nondeterminism in $n^{1.5}$ time and output its answer.*"

Clearly, D_1 runs in $O(n^2)$ time. We show that D_1 accepts a different language from any NDTM M_i that runs in $O(n)$ time. For contradiction's sake assume M_i runs in $cn + d$ time and accepts the same language as D_1. From the construction of M_i, we know that for all input sizes n such that $f(i, j) = L_n$ for some j and $L_n < n \leq H_n$, D_1 accepts 1^n (and hence so does M_i) iff M_i accepts 1^{n+1}. Now if M_i and D_1 accept the same language, we conclude that M_i accepts 1^n iff M_i accepts 1^{n+1}. This chain of implications leads to the conclusion that M_i accepts 1^{L_n+1} iff it accepts 1^{H_n}. But this leads to a a contradiction, since when j is large enough (specifically, larger than $c + d$), machine D_1 accepts 1^{H_n} iff M_i rejects 1^{L_n+1}. Hence M_i and D_1 cannot accept the same language.

4.4 Can diagonalization resolve P vs NP?

Quantifying the limits of "diagonalization" is not easy. Certainly, the diagonalization in Section 4.3 seems more clever than the one in Section 4.1 or the one that proves the undecidability of the halting problem.

For concreteness, let us say that "diagonalization" is any technique that relies upon the following properties of Turing machines

1. The existence of an effective enumeration of all Turing machines $M_1, M_2, \ldots$, and
2. The ability of one TM to simulate any another without much overhead in running time or space. This is a stronger version of the classical theorem about the existence of a universal Turing machine: we are using the fact that the universal Turing machine, when given the index i and some input x, can simulate t steps of machine M_i on this input almost as efficiently as M_i would.

Any argument that only uses these facts is treating machines as *blackboxes*: the machine's internal workings do not matter. To formalize this further we need the following notion.

DEFINITION 11 (ORACLE TURING MACHINES) *If B is a language, then a machine M* with access to oracle *B, denoted M^B, is a machine with a special* query tape*. It can write down any string on this tape, and learn from the oracle in a single step whether or not the string is in the language B.*

We denote by P^B the class of languages accepted by deterministic polynomial time machines that have access to oracle B. We similarly define C^B for any complexity class C.

EXAMPLE 6 $\overline{\text{SAT}} \in P^{\text{SAT}}$. To decide whether a formula $\varphi \in \overline{\text{SAT}}$, the machine asks the oracle if $\varphi \in$ SAT, and then gives the opposite answer as its output.

Note that the two properties needed for diagonalization arguments continue to hold when all machines are provided with the same oracle. There is an effective enumeration of Turing machines since an oracle Turing machine's description (minus the oracle) is of finite size. Furthermore, the Universal machine can still simulate any other machine —whenever the machine being simulated queries the oracle, so does the Universal machine.

We conclude that if we could resolve P vs NP —in whichever direction— using a proof technique involving only simulation then the proof would also work in the presence of any oracle. Hence the following theorem implies that such a proof cannot exist.

THEOREM 12 (BAKER, GILL, SOLOVAY)
There exist oracles B, C such that $P^C = NP^C$ and $P^B \neq NP^B$.

PROOF: For C we may take any PSPACE-complete problem, say TQBF, since $P^{\text{TQBF}} = NP^{\text{TQBF}} =$ PSPACE. (See exercises.) Now we construct B. For any language A, let A_u be the unary language

$$A_u = \{1^n : \text{some string of length } n \text{ is in } A\}.$$

For every oracle A, the language A_u is clearly in NP^A. Below we construct an oracle B such that $B_u \notin P^B$. Hence $B_u \in NP^B \setminus P^B$, and we conclude $P^B \neq NP^B$.

Construction of B: Let $M_1, M_2, M_3, \ldots$ be all polynomial-time Oracle Turing Machines. (This enumeration need not be effective, since we are merely showing the *existence* of the desired oracle.) We construct B in stages, where stage i ensures that M_i^B does not decide B_u. We construct B by initially letting it be empty, and gradually deciding which strings are in it or outside it. Each stage determines the status of a finite number of strings.

Stage i: So far, we have declared for a finite number of strings whether or not they are in B. Choose n large enough so that it exceeds the length of any such string, and 2^n exceeds the running time of M_i on inputs of length n. Now run M_i on input 1^n. Whenever it queries the oracle about strings whose status has been determined, we answer consistently. When it queries strings whose status is undetermined, we declare that the string is not in B. We continue until M_i halts. Now we make sure its answer on 1^n is incorrect. If M_i accepts, we declare that all strings of length n are not in B, thus ensuring $1^n \notin B_u$. If M_i rejects, we pick a string of length n that it has not queried (such a string exists because 2^n exceeds the running time of M_i) and declare that it is in B, thus ensuring $1^n \in B_u$. In either case, the answer of M_i is incorrect.

Thus our construction of B ensures that for every i, machine M_i does not decide B_u. □

Let us now answer our original question: Can diagonalization or any simulation method resolve P vs NP? Answer: Possibly, but it has to use some fact about TMs that does not hold in presence of oracles. Such facts are termed *nonrelativizing* and we will later encounter examples of such facts. Whether or not they have any bearing upon P vs NP (or other interesting complexity theoretic questions) is open.

Exercises

§1 Show that the following language is undecidable:

$$\left\{< M >: \ M \text{ is a machine that runs in } 100n^2 + 200 \text{ time}\right\}.$$

§2 Show that maintaining a time counter can be done with logarithmic overhead. (Hint??)

§3 Show that $\text{P}^{\text{TQBF}} = \text{NP}^{\text{TQBF}}$.

§4 Show that $\text{SPACE}(n) \neq \text{NP}$. (Note that we do not know if either class is contained in the other.)

§5 Say that a class C_1 is *superior to* a class C_2 if there is a machine M_1 in class C_1 such that for every machine M_2 in class C_2 and every large enough n, there is an input of size between n and n^2 on which M_1 and M_2 answer differently.

(a) Is $\text{DTIME}(n^{1.1})$ superior to $\text{DTIME}(n)$?

(b) Is $\text{NTIME}(n^{1.1})$ superior to $\text{NTIME}(n)$?

§6 Show that there exists a function that is not time-constructible.

§7 (Harder) Suppose we pick a random language C, by deciding for each string independently and with probability $1/2$ whether or not it is in C. Show that with high probability $\mathrm{P}^C \neq \mathrm{NP}^C$.

Chapter 5

Polynomial Hierarchy

The class NP was defined to capture some natural computational problems, for instance

$$\text{CLIQUE} = \{< G, k >: \text{ graph } G \text{ has a clique of size} \geq k\}. \tag{5.1}$$

However, it seems to fail to capture many other natural problems. Consider the problem obtained by just modifying the above problem a little, namely, determinining the largest clique in a graph (phrased as a decision problem):

$$\text{EXACT-CLIQUE} = \{< G, k >: \text{ the largest clique in graph } G \text{ has size exactly } k\}. \tag{5.2}$$

There seems to be no short certificate for membership: $< G, k > \in$ EXACT-CLIQUE iff *there exists* a clique of size k in G and *every other* clique has size at most k.

Similary, consider the language MIN-CNF; it is the decision version of circuit minimization, a topic of interest in electrical engineering. We say that two boolean formulae are *equivalent* if they have the same set of satisfying assignments.

$$\text{MIN-CNF} = \{< \varphi >: \varphi \text{ is not equivalent to any smaller formula}\}. \tag{5.3}$$

$$= \{< \varphi >: \forall \psi, |\psi| < |\varphi|, \exists \text{ assignment } s \text{ such that } \varphi(s) \neq \psi(s)\}. \tag{5.4}$$

Again, there is no obvious notion of a certificate of membership. Note that the problem is in PSPACE, but does not appear to be PSPACE-complete either.

Clearly, one is faced with a new kind of complexity phenomenon and it is instructive to see how we define the right complexity class, namely, the polynomial hierarchy, to "capture" it. This class is a generalization of P, NP, and coNP. Recall the definition of NP, coNP.

DEFINITION 12 *Language L is in NP if there is a language $L_0 \in P$ and constants $c, d > 0$ such that*

$$\forall x \in \{0,1\}^* \quad x \in L \Leftrightarrow \quad \exists y \in \{0,1\}^*, |y| \leq d|x|^c \quad \textit{and} \quad (x, y) \in L_0.$$

Language L in in coNP iff $\overline{L} \in$ NP. In other words, there is a language $L_0 \in P$ and constants $c, d > 0$ such that

$$\forall x \in \{0,1\}^* \quad x \in L \Leftrightarrow \quad \forall y \in \{0,1\}^*, |y| \leq d|x|^c \quad \textit{and} \quad (x, y) \in L_0.$$

DEFINITION 13 (POLYNOMIAL HIERARCHY) *The polynomial hierarchy is defined as $\cup_{i \geq 0} \Sigma_i^p$ or equivalently $\cup_{i \geq 0} \Pi_i^p$, where*

1. $\Sigma_0^p = \Pi_0^p = P$.
2. $\Sigma_1^p = NP$, $\Pi_1^p = coNP$.

3. Σ_i^p *consists of any language for which there is a language* $L_0 \in \Pi_{i-1}^p$ *and* $c, d > 0$ *such that*

$$\forall x \in \{0,1\}^* \quad x \in L \Leftrightarrow \quad \exists y \in \{0,1\}^*, |y| \leq d|x|^c \quad \textit{and} \quad (x,y) \in L_0.$$

4. Π_i^p *consists of any language for which there is a language* $L_0 \in \Sigma_{i-1}^p$ *such that*

$$\forall x \in \{0,1\}^* \quad x \in L \Leftrightarrow \quad \forall y \in \{0,1\}^*, |y| \leq d|x|^c \quad \textit{and} \quad (x,y) \in L_0.$$

EXAMPLE 7 To understand this definition, let us unwrap it for $i = 2$. Language L is in Σ_2^p if there is a language $L_0 \in$ P and constants $c, d > 0$ such that a string x is in L iff

$$\exists y_1 \ \leq d|x|^c \ \forall y_2 \leq d|x|^c \ \ (x, y_1, y_2) \in L_0.$$

Similarly Language L is in Π_2^p if there is a language $L_0 \in$ P and constants $c, d > 0$ such that a string x is in L iff

$$\forall y_1 \ \leq d|x|^c \ \exists y_2 \leq d|x|^c \ \ (x, y_1, y_2) \in L_0.$$

Clearly, $L \in \Pi_2^p$ iff $\overline{L} \in \Sigma_i^p$.

Similarly we can unwrap the definition for general i and directly define Σ_i^p using i quantifiers, the first being $\exists$ and the rest alternating between $\exists$ and $\forall$. The class Π_i^p involves involves i quantifiers, alternating between $\exists$ and $\forall$ and beginning with $\forall$.

What are some natural problems in these classes?

EXAMPLE 8 We show that EXACT-CLIQUE $\in \Sigma_2^p \cap \Pi_2^p$. (A finer placement of EXACT-CLIQUE is done in the exercises.)

To see that EXACT-CLIQUE $\in \Sigma_2^p$, note that $< G, k >$ $\in$ EXACT-CLIQUE iff

$\exists$ S that is a clique of size k in G and $\forall S'$ set S' is not a clique of size $\geq k+1$.

Furthermore, EXACT-CLIQUE $\in \Pi_2^p$ since one could also say

$\forall S'$ set S' is not a clique of size $\geq k+1$ and $\exists$ S that is a clique of size k in G.

The reader can similarly check that MIN-CNF in Π_2^p.

The class Σ_i^p has a complete problem involving quantified boolean with limited number of alternations. Specifically, it is

$$\Sigma_i\text{-SAT} = \exists \vec{x}_1 \forall \vec{x}_2 \exists \cdots Q \vec{x}_i \ \varphi(\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_i) = 1, \tag{5.5}$$

where φ is a boolean formula, each $\vec{x}_i$ is a vector of boolean variables, and Q is $\exists$ or $\forall$ depending on whether i is odd or even.

The next simple theorem is the only theorem in this chapter.

THEOREM 13

1. *If* P = NP *then* PH = P.
2. *If* $\Sigma_i^p = \Pi_i^p$ *then* PH = Σ_i^p.

PROOF: We do the first part; the second part is similar and also easy.

Assuming P = NP we prove by induction on i that $\Sigma_i^p, \Pi_i^p \subseteq$ P. Clearly this is true for $i = 0$. Assume it is true for $i \leq m$ and $L \in \Sigma_{m+1}^p$. Then by Definition 13 there is a language $L_0 \in \Pi_m^p$ and $c, d > 0$ such that

$$\forall x \in \{0,1\}^* \quad x \in L \Leftrightarrow \quad \exists y \in \{0,1\}^*, |y| \leq d|x|^c \quad \text{and} \quad (x,y) \in L_0.$$

By the inductive hypothesis, $L_0 \in$ P. By examining how L is defined in terms of L_0, we conclude that then $L \in$ NP. Hence L must be in P, and the induction is completed. □

We strongly believe that not only is P ≠ NP but also that all levels of PH are distinct. This latter conjecture will be useful in the rest of the course; we will reduce other conjectures to it (in other words, show that the other conjectures follow from this conjecture).

5.1 Alternating Turing machines

There is an alternating characterization of the polynomial hierarchy using *alternating Turing machines*. (Aside: There is a third characterization of PH that uses oracle turing machines, and is explored in the exercises.)

Alternating TMs are like nondeterministic TMs except the states are labelled with either $\exists$ or $\forall$. A nondeterministic TM is a special case in which states are labelled with only $\exists$. The acceptance criterion for such a machine is defined in the obvious way by looking at the tree of all possible computation branches, and propagating the YES/NO decisions at the leaves to the root using the obvious semantics for $\exists$ and $\forall$. Then Σ_i^p is the class of languages accepted by polynomial time alternating TMs in which each computation branches features at most $(i-1)$ alternations between $\exists$ and $\forall$, and the machine starts in an $\exists$ state. The class Π_i^p is similarly defined except the machine starts in a $\forall$ state.

It is easily checked that this definition is equivalent to our earlier definitions.

Unlimited number of alternations? What if we consider polynomial-time alternating Turing machines with no *a priori* bound on the number of quantifiers? The class of languages accepted by them is called AP, and the exercises ask you to prove that AP = PSPACE.

Similarly, one can consider alternating Turing machines that run in polynomial space. The class of languages accepted by such machines is called APSPACE, and the exercises ask you to prove that APSPACE = EXPTIME. One can similarly consider alternating logspace machines; the set of languages accepted by them is exactly P.

Problems

§1 Show that the language in (5.5) is complete for Σ_i^p under polynomial time reductions. (Hint use the NP-completeness of SAT.)

§2 Show that PH has a complete language iff it collapses to some finite level Σ_i^p.

§3 Prove that AP = PSPACE.

§4 Show that APSPACE = EXPTIME. (Hint: The nontrivial direction EXPTIME $\subseteq$ APSPACE uses ideas similar to those in the Cook-Levin theorem, namely, that computation is very local.)

§5 Show that $\Sigma_2^p = \text{NP}^{\text{SAT}}$. Generalize your proof to give a characterization of PH in terms of oracle Turing machines.

§6 The class DP is defined as languages L for which there are two languages $L_1 \in$ NP, $L_2 \in$ coNP such that $L = L_1 \cap L_2$. (Remark: Do not confuse DP with NP $\cap$ coNP, which may seem superficially similar.) Show that

(a) EXACT-CLIQUE $\in$ DP.

(b) Every language in DP is polynomial-time reducible to EXACT-CLIQUE.