COS 522: Computational Complexity
Princeton University, Spring 2001
Lecturer: Sanjeev Arora

These are scribe notes from the above course. Eight grads and two undergrads took the course. They were supposed to have an undergrad level preparation in Theory of Computation, preferably using Mike Sipser's excellent text.

I am grateful to these students, who eagerly attended these lectures, solved the homework problems (gamely trying their hands at the occasional open problem), and wrote up scribe notes. I have edited the scribe notes but many errors undoubtedly remain.

These lecture notes omit many topics from complexity theory, most notably the PCP Theorem and computational pseudorandomness as developed in the 1990s (extractors, hardness vs randomness, etc). Luca Trevisan and I plan to write a graduate textbook on Computational Complexity that will cover these and other topics as well.

Course homepage: `http://www.cs.princeton.edu/courses/archive/spring01/cs522/`

Homepage for Luca's course: `http://www.cs.berkeley.edu/ luca/cs278/`

Contents

1 Introduction **5**

2 Space Complexity **9**

3 Diagonalization **13**

4 Polynomial Hierarchy **17**

5 Circuits **20**

6 Randomized Computation **25**

7 Counting classes **29**

8 Toda's Theorem **35**

9 One-way functions and hard-core bit theorem **40**

10 One-way permutations and Goldreich-Levin bit theorem **44**

11 Interactive Proofs **50**

12 PCP Theorem and the hardness of approximation **57**

13 Decision Tree Complexity **58**

14 Communication complexity **65**

15 Circuit complexity **69**

16 Circuit Lower Bounds Using Multiparty Communication Complexity **75**

17 Algebraic Computation Models **80**

18 Natural Proofs **88**

19 Quantum Computation **93**

20 Homeworks **106**

Chapter 1

Introduction

SCRIBE: *Arora*

Computational complexity theory 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.

But let us refresh our memories about complexity as defined using a multitape TM, and explored extensively in undergrad texts (e.g., Sipser's excellent "Introduction to the Theory of Computation.") 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. We use asymptotic notation in measuring the resources used by a TM. Let $\mathbf{DTIME}(t(n))$ 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 $\mathbf{NTIME}(t(n))$ be the class of languages that can be decided by a nondeterministic multitape TM (NDTM for short) in time $O(t(n))$.

The classes $\mathbf{P}$ and $\mathbf{NP}$, and the question whether they are the same is basic to the study of complexity.

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

We believe that the class $\mathbf{P}$ 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 $\mathbf{P}$

[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.

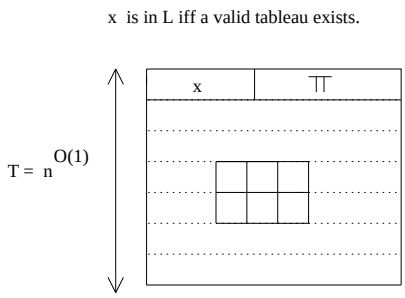


Figure 1.1: Tableau as used in Cook-Levin reduction

exactly captures the notion of languages with "feasible" decision procedures. Of course, one may argue whether $\mathbf{DTIME}(n^{100})$ represents "feasible" computation in the real world. However, in practice, whenever we show that a problem is in $\mathbf{P}$, we usually can find an n^3 or n^5 time algorithm for it.

$\mathbf{NP}$ 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 $\mathbf{NP}$) *Language L is in $\mathbf{NP}$ if there is a language $L_0 \in \mathbf{P}$ 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 CLIQUE, 3-COLORING are $\mathbf{NP}$ problems and not known to be in $\mathbf{P}$. The language of connected graphs is in $\mathbf{P}$.

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 $f(x) \in B$. A language is $\mathbf{NP}$-*hard* if there is a polynomial-time reduction from every $\mathbf{NP}$ language to it. An $\mathbf{NP}$-hard language is $\mathbf{NP}$-*complete* if it is $\mathbf{NP}$-hard and in $\mathbf{NP}$.

Cook and Levin independently showed that the language 3SAT is $\mathbf{NP}$-complete. We briefly recall this classical reduction. Let L be an $\mathbf{NP}$ 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 that contains a 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 the tableau represents a correct computation iff all 2×3 *windows* look "correct," i.e. thay satisfy some local consistency conditions. 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.

Remark 1 The fact that $\mathbf{NP}$-hard problems *exist* is trivial: the halting problem is an $\mathbf{NP}$-hard problem. Actually, the fact that $\mathbf{NP}$-complete languages exist is also trivial. For instance, the following language is $\mathbf{NP}$-complete:

$$\{< M, w, 1^n >: \text{NDTM } M \text{ accepts } w \text{ in time } n\}.$$

Cook and Levin's seminal contribution was to describe explicit, combinatorial problems that are **NP**-complete. 3SAT has very simple structure and is easy to reduce to other problems.

1.1 EXPTIME and NEXPTIME

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

DEFINITION 3 $\mathbf{EXPTIME} = \cup_{c \geq 0} \mathbf{DTIME}(2^{n^c})$.
$\mathbf{NEXPTIME} = \cup_{c \geq 0} \mathbf{NTIME}(2^{n^c})$.

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.

THEOREM 1
If $\mathbf{EXPTIME} \neq \mathbf{NEXPTIME}$ *then* $\mathbf{P} \neq \mathbf{NP}$.

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

$$L_{\text{pad}} = \left\{ < x, 1^{2^{|x|^c}} > : \ x \in L \right\} \tag{1.1}$$

is in **NP** (in fact in $\mathbf{NTIME}(n)$). (Aside: this technique of adding a string of symbols to each string in the language is called *padding*.) Hence if $\mathbf{P} = \mathbf{NP}$ 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}. □

What if P = NP?

If $\mathbf{P} = \mathbf{NP}$ —specifically, if an **NP**-complete problem like 3SAT had say an $O(n^2)$ algorithm— 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). 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. Designers of financial software will be able to write the perfect stock market prediction program.

Somewhat intriguingly, this Utopia would have no need for randomness. 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. We would just have to learn to get along better without these, folks.

We will encounter all these consequences of $\mathbf{P} = \mathbf{NP}$ later in the course.

Exercises

§1 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.

§2 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.

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

Chapter 2

Space Complexity

SCRIBE: *Arora*

Today we discuss space-bounded computation. A space-bounded machine has a read-only input tape, and a read-write work tape. We say that it runs in $S(n)$ space if the work tape has $O(S(n))$ cells when the input has n bits. (See Figure 2.1.) We denote by $\mathbf{SPACE}(S(n))$ 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. Note that $\mathbf{DTIME}(t(n)) \subseteq \mathbf{SPACE}(t(n))$ since a TM running in time $t(n)$ can only use $t(n)$ cells in the work tape. Also, $\mathbf{SPACE}(s(n)) \subseteq \mathbf{DTIME}(2^{O(t(n))})$, 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, and it cannot enter the same configuration twice since that would mean it is in an infinite loop. (Recall that the machine is required to halt on every input.)

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

DEFINITION 4 $\mathbf{PSPACE} = \cup_{c>0}\mathbf{SPACE}(n^c)$.
$\mathbf{NPSPACE} = \cup_{c>0}\mathbf{NSPACE}(n^c)$.

Note that $\mathbf{NP} \subseteq \mathbf{PSPACE}$, since polynomial space is enough decide 3SAT (just cycle through all 2^n assignments, where n is the number of variables).

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

EXAMPLE 2 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.

For a proof of the following theorem, see Sipser Chapter 8.

THEOREM 2
TQBF is complete for **PSPACE** *under polynomial-time reductions.*

2.1 Two suprising algorithms

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

$$\text{PATH} = \{< G, s, t >:\ G \text{ is a directed graph in which there is a path from } s \text{ to } t\} \quad (2.1)$$

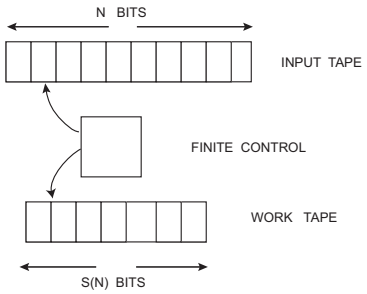


Figure 2.1: A TM running in space $O(S(n))$

PATH has a trivial polynomial time algorithm that uses depth-first search. It is unclear how to implement decide PATH in sublinear space, though. Note 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 an outgoing edge out of this vertex for the next vertex to go to. The machine accepts iff the walk ends at t in at most n steps, where n is the number of nodes. Thus it needs to keep track of the current vertex, which it can do using $O(\log n)$ bits.

The next theorem says that this computation can be made deterministic with quadratic blowup in space.

THEOREM 3 (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 all recursive calls in this description can reuse the same space. 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

$$S(l) \leq O(\log n) + S(\lceil \frac{l}{2} \rceil). \tag{2.2}$$

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

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

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 quite surprising.

THEOREM 4 (IMMERMAN-SZLEPCSENYI)
$\overline{PATH} \in$ **NL**.

PROOF: How can a nondeterministic computation decide that 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. Then the machine's task becomes easier. It keeps aside t and for every other nodes, it sequentially tries to guess —using a nondeterministic walk —a path from s to that node. It accepts at the end iff it succeeds for c nodes, since that means that t is not one of the c nodes connected to s. Note 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, whereby step i determines c_i, the number of nodes reachable from s in i steps. (Thus c_n is the same as c.) Clearly, $c_0 = 1$. To compute c_{i+1} from c_i we use a modification of the idea in the previous paragraph. For each node u we perform a nondeterministic computation which succeeds iff u has distance exactly $i+1$ from s. Our basic "nondeterministic walk" is used over and over. The following nondeterministic procedure computes $c_{i+1} - c_i$ given c_i. (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.)

Maintain a counter, initialized to 0. *Do the following for nodes* 1 *through* n. *When processing node* j, *start a nondeterminstic computation that tries to enumerate exactly* c_i *nodes different from* j *whose distance from* s *is at most* i. *If this enumeration fails, HALT immediately without an answer. If the enumeration succeeds and one of the enumerated nodes has an edge to node* j *then increment the counter and continue on to node* $j + 1$. *If none of the enumerated nodes has an edge to* j, *do not increment the counter but continue the computation.*

When all nodes have been processed, output the counter.

□

2.2 Picture of Space-Bounded Complexity Classes

The following are simple corollaries of the two algorithms, since the PATH problem is complete for **NL**.

THEOREM 5 (SAVITCH)
$\mathbf{NSPACE}(s(n)) \subseteq \mathbf{SPACE}(s(n)^2)$.

(Thus in particular, $\mathbf{PSPACE} = \mathbf{NPSPACE}$, in contrast to the conjecture $\mathbf{P} \neq \mathbf{NP}$.)

THEOREM 6 (IMMERMAN-SZLEPCSENYI)
$\mathbf{NSPACE}(s(n)) = \mathbf{coNSPACE}(s(n))$.

(This is in contrast to the conjecture that $\mathbf{NP} \neq \mathbf{coNP}$.)

Thus the following is our understanding of space-bounded complexity.

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

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.

Chapter 3

Diagonalization

SCRIBE: *Arora*

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 $\mathbf{P} =? \mathbf{NP}$ and other interesting questions.

3.1 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.

THEOREM 7
If f, g are running times satisfying $f(n) \log f(n) = o(g(n))$, then

$$\mathbf{DTIME}(f(n)) \subsetneq \mathbf{DTIME}(g(n)) \tag{3.1}$$

To showcase the essential idea of the proof of Theorem 7, we prove the simpler statement $\mathbf{DTIME}(n) \subsetneq \mathbf{DTIME}(n^2)$. 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 $O(\log i)$. (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 machines that do 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, reject. If M_k halts and rejects, accept. In every other situation (for example if M_k does not halt), 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 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 $\mathbf{DTIME}(n)$. 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 7 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 $O(\log f(n))$.

3.2 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 8
If f, g are running times satisfying $f(n) = o(g(n))$, then

$$\mathbf{NTIME}(f(n)) \subsetneq \mathbf{NTIME}(g(n)) \tag{3.2}$$

Again, we just showcase the main idea of the proof by proving $\mathbf{NTIME}(n) \subsetneq \mathbf{NTIME}(n^2)$. 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

$$f(i,j) = 2^{2^{2^{2^i 3^j}}}. \tag{3.3}$$

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. 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}.) 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 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.

3.3 Can diagonalization resolve P vs NP?

Quantifying the limits of "diagonalization" is not easy. Certainly, the diagonalization in Section 3.2 seems more clever than the one in Section 3.1 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 ability of one TM to simulate another. In order to identify the limitations of such techniques, we observe that they treat TMs as blackboxes: the machine's internal workings do not matter. In particular, the simulations also work if all Turing Machines are provided with the same oracle. (Whenever the TM being simulated queries the oracle, so does the simulating TM.) If we could resolve P vs NP —in whichever direction— using such techniques then the proof would also work in the presence of any oracle. However, we now exhibit oracles B, C such that $P^C = NP^C$ and $P^B \neq NP^B$, which implies that such a proof cannot exist.

For C we may take any **PSPACE**-complete problem, say TQBF, since $P^{\mathrm{TQBF}} = NP^{\mathrm{TQBF}} = \mathbf{PSPACE}$. 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 no M_i decides 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.

Oracle Turing Machines

We give a quick introduction to oracle Turing machines, which were used above. 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 $\mathbf{P}^B$ is the class of languages accepted by deterministic polynomial time machines that have access to oracle B.

EXAMPLE 3 $\overline{\text{SAT}} \in P^{\text{SAT}}$. To decide whether a formula $\varphi \in \overline{\text{SAT}}$, the machine asks the oracle if $\varphi \in \text{SAT}$, and then gives the opposite answer as its output.

Exercises

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

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

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

§4 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 $\mathbf{DTIME}(n^{1.1})$ superior to $\mathbf{DTIME}(n)$?

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

§5 Show that the following language is undecidable:

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

Chapter 4

Polynomial Hierarchy

SCRIBE: *Arora*

The polynomial hierarchy is a hierarchy of complexity classes that generalizes **P**, **NP**, and **coNP**. Recall the definition of **NP**, **coNP**.

DEFINITION 5 *Language L is in* **NP** *if there is a language $L_0 \in \mathbf{P}$ 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 $\overline{L} \in \mathbf{NP}$. In other words, there is a language $L_0 \in \mathbf{P}$ 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 6 (POLYNOMIAL HIERARCHY) *The polynomial hierarchy is defined as $\cup_{i \geq 0} \mathbf{\Sigma}_i^p$ or equivalently $\cup_{i \geq 0} \mathbf{\Pi}_i^p$, where*

1. $\mathbf{\Sigma}_0^p = \mathbf{\Pi}_0^p = \mathbf{P}$.
2. $\mathbf{\Sigma}_1^p = \mathbf{NP}$, $\mathbf{\Pi}_1^p = \mathbf{coNP}$.
3. $\mathbf{\Sigma}_i^p$ *consists of any language for which there is a language $L_0 \in \mathbf{\Pi}_{i-1}^p$ such that*

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

4. $\mathbf{\Pi}_i^p$ *consists of any language for which there is a language $L_0 \in \mathbf{\Sigma}_{i-1}^p$ such that*

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

EXAMPLE 4 To understand this definition, let us unwrap it for $i = 2$. Language L is in $\mathbf{\Sigma}_2^p$ if there is a language $L_0 \in \mathbf{P}$ and constants $c, d > 0$ such that a string x is in L iff

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

Similarly Language L is in $\mathbf{\Pi}_2^p$ if there is a language $L_0 \in \mathbf{P}$ and constants $c, d > 0$ such that a string x is in L iff

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

Clearly, $L \in \mathbf{\Pi}_2^p$ iff $\overline{L} \in \mathbf{\Sigma}_i^p$.

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

What are some natural problems in these classes? Consider the language EXACT-TSP, defined as

$$\{< G, C >:\ G \text{ is a weighted graph and its shortest salesman tour has length } C\}\,. \quad (4.1)$$

$$= \left\{< G, C >:\ \exists \text{ salesman tour } \pi \text{ s.t. } \text{cost}(\pi) = C \text{ and } \forall \text{ tour } \pi' \ \text{cost}(\pi') \geq C\right\} \quad (4.2)$$

Then EXACT-TSP $\in \mathbf{\Sigma}_2^p$.

Now we describe a language MIN-CNF in $\mathbf{\Pi}_2^p$; this language is of interest in electrical engineering, specifically, circuit minimization. We say that two boolean formulae are *equivalent* if they have the same set of satisfying assignments.

$$\text{MIN-CNF} = \{< \varphi >:\ \varphi \text{ is not equivalent to any smaller formula}\}\,. \quad (4.3)$$

$$= \{< \varphi >:\ \forall \psi, |\psi| < |\varphi|\,, \exists \text{ assignment } s \text{ such that } \varphi(s) \neq \psi(s)\}\,. \quad (4.4)$$

The class Σ_i^p has a complete problem involving quantified boolean formulae[1] with limited number of alternations. Specifically, it is

$$\Sigma_i\text{-SAT} = \exists \vec{x}_1 \forall \vec{x}_2 \exists \cdots Q \vec{x}_i \ \varphi(\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_i) = 1, \quad (4.5)$$

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 lecture.

THEOREM 9
If $\mathbf{P} = \mathbf{NP}$ *then* $\mathbf{PH} = \mathbf{P}$.

PROOF: Easy. □

We strongly believe that not only is $\mathbf{P} \neq \mathbf{NP}$ but also that all levels of $\mathbf{PH}$ are distinct. This latter conjecture will be useful in the rest of the course; we will reduce other conjectures to it (that is to say, prove other conjectures assuming this is true).

[1]The resemblance to TQBF is not coincidental. In the definition of $\mathbf{\Sigma}_i^p$ if we allow the number of alternations i to be polynomial in the input, then we get a class called AP, which is exactly PSPACE, and hence has TQBF as its complete problem. Verify this!

4.1 Alternating Turing machines

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 $\mathbf{\Sigma}_i^p$ 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 $\mathbf{\Pi}_i^p$ is similarly defined except the machine starts in a $\forall$ state.

It is easily checked that this definition is equivalent to our earlier definitions. We note that the levels of the polynomial hierarchy can also be defined using oracle turing machines. This is explored in the exercises.

Problems

§1 Show that the language in (4.5) is complete for $\mathbf{\Sigma}_i^p$ under polynomial time reductions. (Hint use the **NP**-completeness of SAT.)

§2 Prove Theorem 9.

§3 Prove that $\mathbf{AP} = \mathbf{PSPACE}$, as claimed in the footnote.

§4 Suppose we define logspace computation.

§5 Show that $\mathbf{\Sigma}_2^p = \mathbf{NP}^{\mathrm{SAT}}$.

Chapter 5

Circuits

SCRIBE: *Arora*

The boolean circuit model mimics a silicon chip. A circuit with n inputs is a directed acyclic graph with n leaves (called the *input wires*) and each internal node is labelled with $\vee, \wedge, \neg$. The $\vee$ and $\wedge$ nodes have fanin (i.e., number of incoming edges) of 2 and the $\neg$ nodes have fanin 1. There is a designated *output node*. We think of n bits being fed into the input wires. Each internal node/gate applies the boolean operation it is labeled with on its incoming wires to produce a single bit. Thus bits filter through the circuit —remember that it is acylic, so feedback effects are absent— until a single output bit emerges at the output node. Thus the circuit implements a function from $\{0,1\}^n \to \{0,1\}$. The *size* of the circuit is the number of nodes in it.

A *circuit family* $(W_n)_{n\geq 1}$ is a sequence of circuits where W_n is a circuit with n inputs. We say that the family *decides* a language L if for every input x,

$$x \in L \Leftrightarrow W_{|x|}(x) = 1. \tag{5.1}$$

Note that every language is decidable by a circuit family of size $O(n2^n)$, since the circuit for input length n could contain 2^n "hardwired" bits indicating which inputs are in the language. Given an input, the circuit looks up the answer from this table. (The reader may wish to work out an implementation of this circuit.)

DEFINITION 7 *The class* $\mathbf{P}/poly$ *consists of every language that is decidable by a circuit family of size* $O(n^c)$ *for some* $c > 0$.

THEOREM 10
$\mathbf{P} \subseteq \mathbf{P}/poly$.

PROOF: We show that every language that is decidable in $t(n)$ time has circuits of size $O(t(n)^2)$. (A more careful argument can actually yield circuits of size $O(t(n)\log t(n))$. The main idea is similar to that in the Cook-Levin theorem, where we noticed that computation is very local, since the machine affects its tape only in the cell currently occupied by its head.

On any input of size n, the tableau of the machine's computation is a matrix of size $t(n) \times t(n)$. Assume wlog that the machine enters the leftmost cell before accepting. Construct a

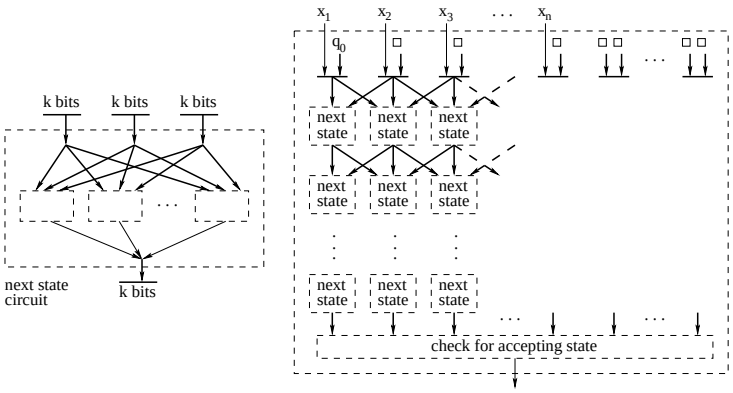


Figure 5.1: Circuit to simulate a Turing machine computation by constructing the tableau.

circuit in which use a wire for each bit of the tableau. The bits of the machine's input enter at the input wires of the circuit. Construct a "next step" module, which is a circuit that, given the contents of three adjacent cells in a row of the tableau, computes the contents of the cell below the middle cell. Since the machine has a fixed transition diagram and alphabet, each tableau cell is represented by $k = O(1)$ bits, so the the next-step circuit has size $O(1)$ (could be exponential in k). Replicate this next-step module everywhere as needed so that all the bits in the tableau are computed correctly. (See Figure 5.1.) The output of the overall circuit is computed by examining the cell in the lower left corner of tableau; it is 1 iff the machine is in an accept state. □

Can we give a good upperbound on the computational power represented by $\mathbf{P}/\text{poly}$? The next example shows this is difficult since $\mathbf{P}/\text{poly}$ contains an undecidable language.

Example 5 Every unary language has linear size circuits since the circuit for an input size n only needs to have a single "hardwired" bit indicating whether or not 1^n is in the language. The following unary language is undecidable:

$$\left\{1^{<M,w>} : \text{ TM } M \text{ accepts } w\right\}, \tag{5.2}$$

where $< M, w >$ is an encoding that encodes strings with integers.

5.1 Karp-Lipton Theorem

Is SAT in $\mathbf{P}/\text{poly}$? If so, one could imagine a government-financed project to construct a small circuit that solves SAT on, say, $10,000$ variables. Upon discovering such a circuit we could incorporate it in a small chip, to be used in all kinds of commercial products. In particular, such a chip would jeopardise all current cryptographic schemes.

The next theorem seems to dash such hopes, at least if you believe that the polynomial hierarchy does not collapse.

Theorem 11 (Karp-Lipton, with improvements by Sipser)
If $\mathbf{NP} \subseteq \mathbf{P}/poly$ *then* $\mathbf{PH} = \mathbf{\Sigma}_2^p$.

Note that if $\mathbf{NP} \subseteq \mathbf{P}/\text{poly}$ then SAT has a polynomial-size circuit family. Namely, for some $k > 0$ there is a sequence of circuits $W_1, W_2, W_3, \ldots$, where W_n is a circuit of size n^k that decides satisfiability of all boolean formulae whose encoding size is $\leq n$. Of course, $\mathbf{NP} \subseteq \mathbf{P}/\text{poly}$ merely implies the *existence* of such a family; there may be no easy way to construct the circuits.

LEMMA 12 (SELF-REDUCIBILITY OF SAT)
There is a polynomial-time computable function h such that if $\{W_n\}_{n\geq 1}$ is a circuit family that solves SAT, then for all boolean formulae φ:

$$\varphi \in SAT \qquad \textit{iff} \qquad h(\varphi, W_{|\varphi|}) \textit{ is a satisfying assignment for } \varphi. \tag{5.3}$$

PROOF: The main idea in the computation of h is to use the provided circuit to generate a satisfying assignment. Ask the circuit if φ is satisfiable. If so, ask it if $\varphi(x_1 = T)$ (i.e., φ with the first variable assigned True) is satisfiable. The circuit's answer allows us to reduce the size of the formula. If the circuit says no, we can conclude that $\varphi(x_1 = F)$ is true, and have thus reduced the number of variables by 1. If the circuit says yes, we can substitute $x_1 = T$ and again reduced the number of variables by 1. Continuing this way, we can generate a satisfying assignment. This proves the Lemma. (Aside: The formal name for the above property of SAT is *downward self-reducibility.* All **NP**-complete languages have this property.) □

Now we prove the Theorem.

PROOF:(Theorem 11) We show $\mathbf{NP} \subseteq \mathbf{P}/\text{poly}$ implies $\mathbf{\Pi}_2^p \subseteq \mathbf{\Sigma}_2^p$.

Let $L \in \mathbf{\Pi}_2^p$. Then there is a language $L_1 \in \mathbf{NP}$ and $c > 0$ such that

$$L = \{x \in \{0,1\}^* : \forall\ y, |y| \leq |x|^c, (x, y) \in L_1\}. \tag{5.4}$$

Since $L_1 \in \mathbf{NP}$, there is a polynomial-time reduction, say g, from L_1 to SAT. Thus

$$\forall\ z \in \{0,1\}^* : z \in L_1 \qquad \text{iff} \qquad g(z) \in \text{SAT}.$$

Suppose further that $d > 0$ is such that $|g(z)| \leq |z|^d$.

Now we can rewrite (5.4) as

$$L = \{x \in \{0,1\}^* : \forall\ y, |y| \leq |x|_1^c, g(x, y) \in \text{SAT}\}.$$

Note that $|g(x, y)| \leq (|x| + |y|)^d$. Let us simplify this as $|x|^{cd}$. Thus if $W_{n\geq 1}$ is a n^k-sized circuit family for SAT, then by (5.3) we have

$$L = \left\{x : \forall\ y, |y| \leq |x|_1^c, h(g(x, y), W_{|x|^{cd}}) \text{ is a satisfying assignment for } g(x, y)\right\}.$$

Now we are almost done. Even though there may be no way for us to construct the circuit for SAT, we can just try to "guess" it. Namely, an input x is in L iff

$$\begin{gathered}\exists W, \text{ a circuit with } |x|^{cd} \text{ inputs and size } |x|^{cdk} \text{ such that} \\ \forall\ y, |y| \leq |x|_1^c, h(g(x, y), W) \text{ is a satisfying assignment for } g(x, y).\end{gathered}$$

Since h is computable in polynomial time , we have thus shown $L \in \mathbf{\Sigma}_2^p$. □

5.2 Circuit lowerbounds

Theorem 10 implies that if $\mathbf{NP} \not\subseteq \mathbf{P}/\text{poly}$, then $\mathbf{P} \neq \mathbf{NP}$. The Karp-Lipton theorem gives hope that $\mathbf{NP} \not\subseteq \mathbf{P}/\text{poly}$. Can we resolve $\mathbf{P}$ versus $\mathbf{NP}$ by proving $\mathbf{NP} \not\subseteq \mathbf{P}/\text{poly}$? There is reason to invest hope in this approach as opposed to proving direct lowerbounds on Turing machines. By representing computation using circuits (Theorem 10) we seem to actually peer into the guts of it rather than treating it as a blackbox. Thus we may be able to get around the relativization results of Lecture 3.

Sadly, such hopes have not come to pass. After two decades, the best circuit size lowerbound for an $\mathbf{NP}$ language is only $4n$. (However, see the Problems for a better lowerbound for a language in $\mathbf{PH}$.) On the positive side, we have had notable success in proving lowerbounds for more restricted circuit models, as we will see later in the course.

By the way, it is easy to show using counting arguments that for large enough n, almost every boolean function on n variables requires circuits of size at least $2^n/10n$. The reason is that there are at most s^{3s} circuits of size s (just count the number of labelled directed graphs, where each node has indegree at most 2). For $s = 2^n/10n$, this number is miniscule compared 2^{2^n}, the number of boolean functions on n variables. Hence most boolean functions do not have such small circuits.

Of course, such simple counting arguments do not give an explicit boolean function that requires large circuits.

5.3 Turing machines that take advice

There is a way to define $\mathbf{P}/\text{poly}$ using TMs. We say that a Turing machine has *access to an advice family* $\{a_n\}_{n\geq 0}$ (where each a_n is a string) if while computing on an input of size n, the machine is allowed to examine a_n. The advice family is said to have *polynomial size* if there is a $c \geq 0$ such that $|a_n| \leq n^c$.

THEOREM 13
$L \in \mathbf{P}/\text{poly}$ iff L is decidable by a polynomial-time Turing machine with access to an advice family of polynomial size.

PROOF: If $L \in \mathbf{P}/\text{poly}$, we can provide the description of its circuit family as advice to a Turing machine. When faced with an input of size n, the machine just simulates the circuit for this circuit provided to it.

Conversely, if L is decidable by a polynomial-time Turing machine with access to an advice family of polynomial size, then we can use the construction of Theorem 10 to construct an equivalent circuit for each input size with the corresponding advice string hardwired into it. □

Exercises

§1 Show for every $k > 0$ that $\mathbf{PH}$ contains languages whose circuit complexity is $\Omega(n^k)$. (Hint: First show that such a language exists in $\mathbf{SPACE}(2^{\text{poly}(n)})$.)

§2 Show that if some unary language is **NP**-complete, then $\mathbf{P} = \mathbf{NP}$.

§3 (*Open*) Suppose we make a stronger assumption than $\mathbf{NP} \subseteq \mathbf{P}/\text{poly}$: every language in **NP** has linear size circuits. Can we show something stronger than $\mathbf{PH} = \mathbf{\Sigma}_2^p$?

Chapter 6

Randomized Computation

SCRIBE: *Manoj*

A randomized Turing Machine has a transition diagram similar to a nondeterministic TM: multiple transitions are possible out of each state. Whenever the machine can validly take more than one outgoing transition, we assume it chooses randomly among them. For now we assume that there is either one outgoing transition (a deterministic step) or two, in which case the machine chooses each with probability 1/2. (We see later that this simplifying assumption does not effect any of the theory we will develop.) Thus the machine is assumed to have a fair random coin.

The quantity of interest is the probability that the machine accepts an input. The probability is over the coin flips of the machine.

6.1 RP and BPP

DEFINITION 8 **RP** *is the class of all languages L, such that there is a polynomial time randomized TM M such that*

$$x \in L \Leftrightarrow \mathbf{Pr}[M \text{ accepts } x] \geq \frac{1}{2} \tag{6.1}$$

$$x \notin L \Leftrightarrow \mathbf{Pr}[M \text{ accepts } x] = 0 \tag{6.2}$$

We can also define a class where we allow two-sided errors.

DEFINITION 9 **BPP** *is the class of all languages L, such that there is a polynomial time randomized TM M such that*

$$x \in L \Leftrightarrow \mathbf{Pr}[M \text{ accepts } x] \geq \frac{2}{3} \tag{6.3}$$

$$x \notin L \Leftrightarrow \mathbf{Pr}[M \text{ accepts } x] \leq \frac{1}{3} \tag{6.4}$$

As in the case of NP, we can given an alternative definiton for these classes. Instead of the TM flipping coins by itself, we think of a string of coin flips provided to the TM as an additional input.

DEFINITION 10 **BPP** *is the class of all languages L, such that there is a polynomial time deterministic TM M and $c > 0$ such that*

$$x \in L \Leftrightarrow \mathbf{Pr}_{r \in \{0,1\}^{|x|^c}}[M \quad accepts\ (x, r)] \geq \frac{2}{3} \tag{6.5}$$

$$x \notin L \Leftrightarrow \mathbf{Pr}_{r \in \{0,1\}^{|x|^c}}[M \quad accepts\ (x, r)] \leq \frac{1}{3} \tag{6.6}$$

We make a few observations about randomized classed. First, **RP** $\subseteq$ **NP**, since a random string that causes the machine to accept is a certificate that the input is in the language. We do not know if **BPP** is in **NP**. Note that **BPP** is closed under complementation. We do not know of any complete problems for these classes. One difficulty seems to be that the problem of determining whether or not a given randomized polynomial time TM is an RP machine is undecidable.

6.1.1 Probability Amplification

In the definition of **BPP** above the actual values in the place of $\frac{2}{3}$ and $\frac{1}{3}$ are not crucial, as we observe below. These two numbers can be replaced by $1 - 1/2^n$ and $1/2^n$ without changing the class.

By repeatedly running a machine M accepting a **BPP** language L with probabilities as above, and taking the majority vote of the different decisions it makes, we can get a much better probability of a correct answer. Using a polynomial (in input size) number of repeated trials, we get an exponentially small probability of deciding the input wrongly. This follows from bounding the tail of a binomial distribution using Chernoff bounds.

6.2 Theorems about BPP

Now we show that all **BPP** languages have polynomial sized circuits.

THEOREM 14
BPP $\subseteq$ **P**/poly

PROOF: For any language $L \in$ **BPP** consider a TM M (taking a random string as additional input) which decides any input x with an exponentially low probability of error; such an M exists by the probability amplification arguments mentioned earlier. In particular let the probability $M(x, r)$ being wrong (taken over r) be $\leq 1/2^{(n+1)}$, where $n = |x|$. Say that an r is *bad* for x if $M(x, r)$ is an incorrect answer. Let t be the total number of choices for r. For each x, at most $t/2^{(n+1)}$ values of r are bad. Adding over all x, we conclude that at most $2^n \times t/2^{(n+1)}$ values of r are bad for some x. In other words, at least $t/2$ choices of r not bad for every x. In particular there is some one value of r for which $M(x, r)$ decides x correctly. We can hard-wire this r into a polynomial size circuit. Given an input x, the circuit simulates M on (x, r). Hence $L \in$ **P**/poly. □

The next theorem relates **BPP** to the polynomial hierarchy.

THEOREM 15
BPP $\subseteq \mathbf{\Sigma}_2^p \cap \mathbf{\Pi}_2^p$

PROOF: It is enough to prove that $\mathbf{BPP} \subseteq \mathbf{\Sigma}_2^p$ because **BPP** is closed under complementation.

Suppose $L \in \mathbf{BPP}$, and M is a randomized TM for L, that uses m random bits such that $x \in L \Rightarrow \mathbf{Pr}_r[M(x,r) \text{ accepts }] \geq 1 - 2^{-n}$ and $x \notin L \Rightarrow \mathbf{Pr}_r[M(x,r) \text{ accepts }] \leq 2^{-n}$.

Fix an input x, and let S_x denote the set of r for which M accepts (x,r). Then either $|S_x| \geq (1-2^{-n})2^m$ or $|S_x| \leq 2^{-n}2^m$, depending on whether or not $x \in L$. We will show how to guess, using two alternations, which of the two cases is true.

Consider r as an element of $\mathbf{GF(2)}^m$. For a set $S \subset \mathbf{GF(2)}^m$ we define the *shift of S by r_0* as $\{r + r_0 | r \in S\}$ (where the addition is as in $\mathbf{GF(2)}^m$, i.e., bit-wise XOR).

Suppose $x \in L$, so $|S_x| \geq (1-2^{-n})2^m$. We shall show that there are a small number of vectors such that the set of shifts of S_x by these vectors covers the whole space $\mathbf{GF(2)}^m$.

LEMMA 16
$\exists r_1, r_2, \ldots, r_k$, where $k = \lceil \frac{m}{n} \rceil + 1$ such that $\bigcup_{i=1}^k (S_x + r_i) = \mathbf{GF(2)}^m$.

PROOF: We shall show that $\mathbf{Pr}_{(r_1,r_2,\ldots,r_k)}[\bigcup_{i=1}^k (S_x + r_i) \neq \mathbf{GF(2)}^m] < 1/2$.

Let $z \in \mathbf{GF(2)}^m$. If r_1 is a random vector, then so is $z + r_1$. So $\mathbf{Pr}_{r_1}[z \notin S_x + r_1] = \mathbf{Pr}_{r_1}[z + r_1 \notin S_x] \leq 2^{-n}$. So, for each z,

$$\mathbf{Pr}_{r_1,r_2,\ldots,r_k}[z \notin \bigcup_{i=1}^{k} S_x + r_i] \leq 2^{-nk}$$

So, $\mathbf{Pr}_{r_1,r_2,\ldots,r_k}[\text{some } z \notin \bigcup_{i=1}^k S_x + r_i] \leq 2^{m-nk} < 1/2 < 1$. □

Now suppose $x \notin L$. Now $|S_x| \leq 2^{-n}2^m$. This is a small set, and the union of any k shifts of S_x can be of size at most $k2^{m-n} < 2^m$, and hence cannot equal $\mathbf{GF(2)}^m$.

Thus we have established that

$$\begin{aligned} x \in L \Leftrightarrow \exists r_1, r_2, \ldots, r_k \in \mathbf{GF(2)}^m \text{ such that} \\ \forall\, z \in \mathbf{GF(2)}^m \; M \text{ accepts } x \text{ using at least one of } z + r_1, z + r_2, \ldots, z + r_k \end{aligned} \tag{6.7}$$

Thus L is in $\mathbf{\Sigma}_2^p$. □

6.3 Model Independence

Now we argue that our specific assumptions about randomized TMs do not affect the classes **RP** and **BPP**.

We made the assumption that the Turing machine uses a two-sided fair coins for randomization. Now we shall see that it can use these to simulate a machine that uses k-sided coins. To simulate a k-sided fair coin do as follows: flip our two-sided coin for $\lceil \log_2 k \rceil$ times. If the binary number generated is in the range $[0, k-1]$ output it as the result, else repeat the experiment. This terminates in expected $O(1)$ steps, and the probability that it does not halt in $O(k)$ steps is at most 2^{-k}. Since the machine needs to simulate only poly(n) coin tosses, the probability that this simulation does not work can be made 2^{-n}, and so it

does not substantially affect the probability of acceptance (which is something like 1/2 or 2/3).

Another issue that arises in real life is that one may not have an unbiased coin, and the probability it comes up heads is an unknown quantity p. But we can simulate a an unbiased coin as follows: flip the (biased) coin twice (we are assuming independence of different flips). Interpret 01 as 0 and 10 as 1; on a 00 or 11 repeat the experiment. The probability that we fail to produce a 0 or 1 is at most $1 - p^2 - (1-p)^2$. Since p is constant, this failure probability is a constant. The expected number of repetitions before we produce a bit is $O(1)$.

6.4 Recycling Entropy

Randomness may be considered a resource, because it is likely that a random bit generator is much slower than the rest of the computer. So one would like to minimize the number of random bits used. One place where we can recycle random bits is while repeating a **BPP** or **RP** algorithm.

Consider running an **RP**-machine on some input x. If it accepts x we know x is in the language L. But if it rejects x, we repeat the experiment, up to k times. But to repeat the experiment each time, it is not necessary to acquire all new random bits all over again. Intuitively, we can "recycle" most of the randomness in the previous random string, because when the experiment fails to detect x being in L, all we know about the random string used is that it is in the set of *bad* random strings. Thus it still has a lot of "randomness" left in it.

DEFINITION 11 *If $d > 2$ and $\beta > 0$, a family of* (d, β)-expanders *is a sequence of graphs $\{G_n\}$ where G_n is a d-regular graph on i nodes, and has the property that every subset S of at most $n/2$ nodes has edges to at least $\beta |S|$ nodes outside S.*

Deep mathematics (and more recently, simpler mathematics) has been used to construct expanders. These constructions yield algorithms that, given n and an index of a node in G_n, can produce the indices of the neighbors of this node in $\text{poly}(\log n)$ time.

Suppose the **RP** algorithm uses m random bits. The naïve approach to reduce its error probability to 2^{-k} uses $O(mk)$ random bits. A better approach involving recycling is as follows. Pick the first random string as a node in an *expander graph* on 2^m nodes, take a random walk of length k from there, and use the indices of the nodes encounted in this walk (a total of mk bits) in your **RP** algorithm. Since the graph has constant degree, each step of the random walk needs $O(1)$ random bits. So a random walk of length k takes $m + O(k)$ random bits; m to produce the first node and $O(1)$ for each subsequent step. A surprising result shows that this sequence will be random enough to guarantee the probability amplification, but we will not give a proof here.

Of course, the same approach works for any randomized algorithm, such as a Monte Carlo simulation.

Chapter 7

Counting classes

SCRIBE:*Manoj*

First we define a few interesting problems:

Given a boolean function ϕ, #SAT is the problem of finding the number of satisfying assignments for ϕ. Given a graph G and two nodes s and t, #PATH is the problem of finding the number of simple paths from s to t in G. #CYCLE is the problem of finding the number of cycles in a given graph.

To study functions (rather than languages) which can be computed "efficiently" we make the following definition.

DEFINITION 12 **FP** *is the set of all functions* $f : \{0,1\}^* \to \mathbf{N}$ *computable in polynomial time.*

We would like to know if the problems defined above are in **FP**. The next theorem shows that counting problems may be harder than their corresponding decision problems, since it shows that #CYCLE is **NP**-hard, even though deciding whether a graph has a cycle is easy.

THEOREM 17
If #CYCLE $\in$ **FP**, *then* **P** = **NP**.

PROOF: We reduce HAMCYCLE (the decision problem of whether a digraph has a Hamiltonian cycle or not) to #CYCLE. Given a graph G with n nodes in the HAMCYCLE problem, we construct a graph G' for #CYCLE such that G has a HAMCYCLE iff G' has at least n^{n^2} cycles.

Each edge (u, v) in G is replaced by a gadget as shown in Figure 7.1. The gadget has $N = n \log_2 n$ levels. If G has a Hamiltonian cycle, the G' has at least $(2^N)^n$ cycles, because for each edge from u to v in G, there are 2^N paths from u' to v' in G', and there are n edges in the Hamiltonian cycle in G. On the other hand, if G has no Hamiltonian cycle, the longest cycle in G is of length at most $n - 1$. Also, the number of cycles is bounded above by n^{n-1}. Note that any cycle in G' corresponds to a cycle in G. So G' can have at most $(2^N)^{n-1} \times n^{n-1} < (2^N)^n$ cycles. □

The following class characterizes many counting problems of interest.

Figure 7.1: Reducing HAMCYCLE to #CYCLE

DEFINITION 13 *#**P** is the set of all functions $f : \{0,1\}^* \to \mathbf{N}$ such that there is a polynomial time TM M and a constant c such that*

$$\forall x, f(x) = |\{y : |y| \leq |x|^c \text{ and } M(x,y) \text{ accepts}\}|$$

For example, #CYCLE $\in$ #**P**, because one can construct a polynomial time TM to verify a canonical representation of a cycle in a graph.

DEFINITION 14 *A function $f \in$ #**P** is* #**P**-complete *if $\forall g \in$ #**P**, $g \in \mathbf{FP}^f$*

Counting versions of **NP**-complete languages naturally lead to #**P**-complete languages. This follows from the way reductions preserve the number of certificates. For instance, the Cook-Levin reduction of an **NP** language to SAT gives a one-one correspondence between the satisfying assignments of the boolean function produced and the accepting tableaux of the **NP**-machine.

THEOREM 18
#SAT, #CLIQUE *etc are* #**P**-*complete*

Now we study another problem. The *permanent* of an $n \times n$ matrix A is defined as

$$\text{perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i\sigma(i)} \tag{7.1}$$

where S_n denotes the set of all permutations of n elements. (Recall that the expression for the determinant is similar, but it involves an additional "sign" term.)

Every $n \times n$ 0-1 matrix A corresponds to a bipartite graph $G(X, Y, \Delta)$, with $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_n\}$ and $\{x_i, y_j\} \in \Delta$ iff $A_{ij} = 1$. Then perm(A) is the number of perfect matchings in G. Another useful way to look at a matrix A is as the adjacency matrix of an n-node digraph (with possible self loops); then perm(A) is the number of cycle covers of A. (A *cycle cover* is a subgraph in which each node has in-degree and out-degree 1; such a subgraph must be composed of cycles.)

For a 0/1 matrix perm$(A) = \left|\left\{\sigma : \prod_{i=1}^{n} a_{i\sigma(i)} = 1\right\}\right|$, which reveals that the perm is in #**P** for this case. If A is a $\{-1, 0, 1\}$ matrix, then finding the permanent is in $\mathbf{P}^{\#\text{SAT}}$, because then perm$(A) = \left|\left\{\sigma : \prod_{i=1}^{n} a_{i\sigma(i)} = 1\right\}\right| - \left|\left\{\sigma : \prod_{i=1}^{n} a_{i\sigma(i)} = -1\right\}\right|$. One can show for general integer matrices that computing the permanent is in $\mathbf{FP}^{\#\text{SAT}}$.

The next theorem came as a surprise to researchers in the 1970s, since the permanent seems quite similar to the determinant, which is easy to compute.

THEOREM 19 (VALIANT)
PERM *is* #**P**-*complete*

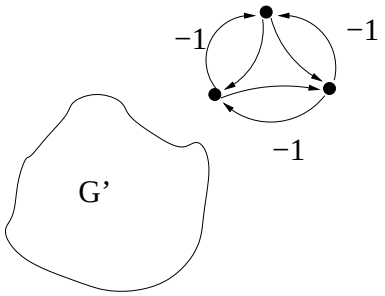


Figure 7.2: This graph has permanent 0

As a warm-up for the proof, we first use an example.

EXAMPLE 6 Consider the graph in Figure 7.2. (Unmarked edges have unit weight. We follow this convention through out this lecture). Even without knowing what the subgraph G' is, we can conclude that the permanent of the whole graph is 0, because for each cycle cover in G' there are exactly two cycle covers for the three nodes, one with weight 1 and one with weight -1 (and any non-zero weight cycle cover of the whole graph is composed of a cycle cover for G' and one of these two cycle covers).

Now we prove Valiant's Theorem.

PROOF: We shall reduce the #**P**-complete problem #3SAT to PERM. Given a boolean formula ϕ with n variables and m clauses, first we shall show how to construct an integer matrix A' with negative entries such that $\text{perm}(A') = 4^m \cdot (\#\phi)$. ($\#\phi$ stands for the number of satisfying assignments of ϕ). Later we shall show how to to get a 0-1 matrix A from A' such that knowing $\text{perm}(A)$ allows us to compute $\text{perm}(A')$.

The main idea is that there are two kinds of cycle covers in the digraph G' associated with A': those that correspond to satisfying assignments (we will make this precise) and those that don't. Recall that $\text{perm}(A')$ is the sum of weights of all cycle covers of the associated digraph, where the weight of a cycle cover is the product of all edge weights in it. Since A' has negative entries, some of these cycle covers may have negative weight. Cycle covers of negative weight are crucial in the reduction, since they help cancel out contributions from cycle covers that do not correspond to satisfying assignments. (The reasoning to prove this will be similar to that in Example 6.) On the other hand, each satisfying assignment contributes 4^m to $\text{perm}(A')$, so $\text{perm}(A') = 4^m \cdot (\#\phi)$.

To construct G' from ϕ, we use three kinds of gadgets as shown in Figures 7.3, 7.4 and 7.5. There is a variable gadget per variable and a clause gadget per clause. There are two possible cycle covers of a variable gadget, corresponding to an assignment of 0 or 1 to that variable. Assigning 1 corresponds to a single cycle taking all the external edges ("true-edges"), and assigning 0 correspond to taking all the self-loops and taking the "false-edge". Each external edge of a variable is associated with a clause in which the variable appears.

The clause gadget is such that the only possible cycle covers exclude at least one external edge. Also for a given (proper) subset of external edges used there is a unique cycle cover (of weight 1). Each external edge is associated with a variable appearing in the clause.

We will also use a graph called the XOR gadget (Figure 7.5) which has the following purpose: we want to ensure that exactly one of the edges uu' and vv' (see the schematic

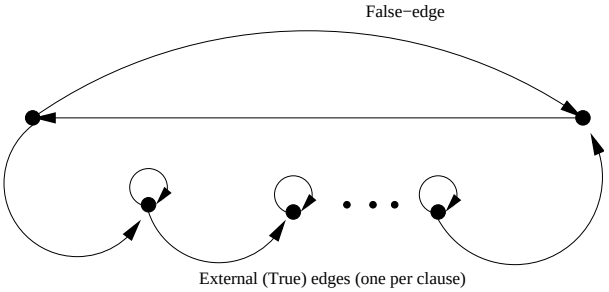


Figure 7.3: The Variable-gadget

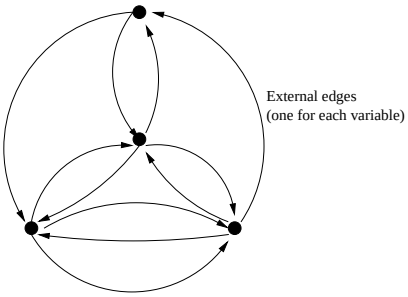


Figure 7.4: The Clause-gadget

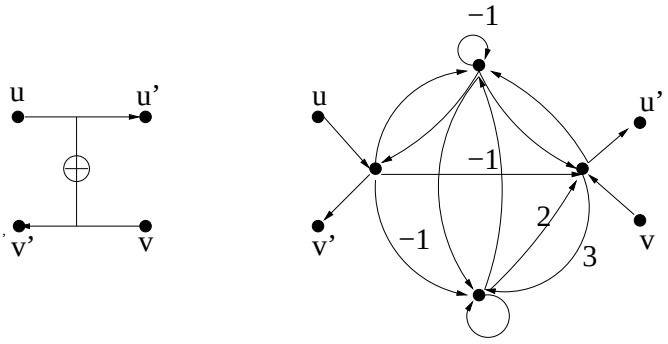


Figure 7.5: The XOR-gadget

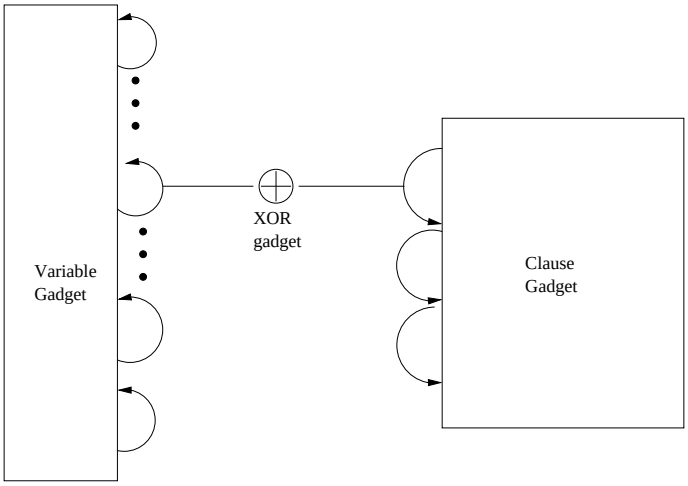


Figure 7.6: For each clause and variable appearing in it, an XOR-gadget connects the corresponding external edges. There are $3m$ such connections in total.

representation in Figure 7.5) is used in a cycle cover that contributes to the total count. So after inserting the gadget, we want to count only those cycle covers which either enter the gadget at u and leave it at u' or enter it at v and leave it at v'. This is exactly what the gadget guarantees: one can check that the following cycle covers have total weight of 0: those that do not enter or leave the gadget; those that enter at u and leave at v', or those that enter at v and leave at u'. In other words, the only cycle covers that have a nonzero contribution are those that either (a) enter at u and leave at u' (which we refer to as using "edge" uu') or (b) enter at v and leave at v' (referred to as using edge vv'). These are cycle covers in the "schematic graph" (which has edges as shown in Figure 7.5) which *respect* the XOR gadget.

The XOR gadgets are used to connect the variable gadgets to the corresponding clause gadgets so that only cycle covers corresponding to a satisfying assignment need be counted towards the total number of cycle covers. Consider a clause, and a variable appearing in it. Each has an external edge corresponding to the other, connected by an XOR gadget (figure 7.6). If the external edge in the clause is not taken (and XOR is respected) the external edge in the variable must be taken (and the variable is true). Since at least one external edge of each clause gadget has to be omitted, each cycle cover respecting all the XOR gadgets corresponds to a satisfying assignment. (If the XOR is not respected, we need not count such a cycle cover as its weight will be cancelled by another cover, as we argued above). Conversely, for each satisfying assignment, there is a cycle cover (unique, in the schematic graph) which respects all the XOR gadgets.

Now, consider a satisfying assignment and the corresponding cycle cover in the schematic graph. Passing (exactly one of) the external edges through the XOR gadget multiplies the weight of each such cover by 4. Since there are $3m$ XOR gadgets, corresponding to each satisfying assignment there are cycle covers with a total weight of 4^{3m} (and all other cycle covers total to 0). So $\text{perm}(G') = 4^{3m}\#\phi$.

Finally we have to reduce finding $\text{perm}(G')$ to finding $\text{perm}(G)$, where G is an un-

weighted graph. First we reduce it to finding perm(G) modulo $2^N + 1$ for a large enough N (but still polynomial in $|G'|$). For this, we can replace -1 edges with edges of weight 2^N, which can be converted to N edges of weight 2 in series. Changing edges of (small) positive integral weights (i.e., multiple or parallel edges) to unweighted edges is as follows: cut each (repeated) edge into two and insert a node to connect them; add a self loop to the node. This does not change the permanent, and the new graph has only unweighted edges. □

Exercises

§1 Show that computing the permanent for matrices with integer entries is in $\mathbf{FP}^{\#\mathrm{SAT}}$.

Chapter 8

Toda's Theorem

SCRIBE: *Arora/Khot*

Toda's Theorem, $\mathbf{PH} \subseteq \mathbf{P}^{\#\mathrm{SAT}}$, came as a big surprise in 1989, since researchers believed that alternation (the feature of the polynomial hierarchy) could not have anything do with the ability of count the number of solutions (a feature of $\#P$).

In this lecture we will use a (yet another) characterization of the class **PH** uniform circuit families, which is left as an exercise.

DEFINITION 15 *Let $\{\mathcal{C}_n\}_{n\geq 1}$ be a circuit family of size $S(n) \geq n$ for input of length n. We say that this is a Direct Connect uniform (DC uniform) family if all the following functions can be computed in deterministic $O(poly(n)log(S(n)))$ time :*

- *$TYPE(n, i)$ = the type (AND, OR, NOT, INPUT, OUTPUT) of gate i in circuit $\mathcal{C}_n$.*
- *$IN(n, i, j) = k$ where k is j^{th} in the ordered list of indices to the gates that feed into gate i, $k = NONE$ if there is no j^{th} gate.*
- *$OUT(n, i, j) = k$ where k is j^{th} in the ordered list of indices to the gates that gate i feeds into, $k = NONE$ if there is no j^{th} gate.*

THEOREM 20
$L \in PH$ iff L can be computed by a DC uniform circuit family $\{\mathcal{C}_n\}$ that

- *uses AND, OR, NOT gates.*
- *has size $2^{n^{O(1)}}$ and constant depth.*
- *gates can have unbounded (exponential) fanin.*
- *the NOT gates appear only at the input level.*

Note that the circuits have exponential size, but they have a succinct representation in terms of a TM which can systematically generate any required portion of the circuit.

We now define a class $\oplus P$ which we use as an intermediate class in the proof of Toda's Theorem.

DEFINITION 16 *A language $L \in \oplus\mathbf{P}$ iff there exists a polynomial time NTM M such that $x \in L$ iff the number of accepting paths of M on input x is odd.*

DEFINITION 17 $\oplus SAT = \{\varphi :$ *boolean formula φ has an odd number of satisfying assignments*$\}$.

A simple modification of the Cook-Levin theorem shows that $\oplus$SAT is complete for $\oplus\mathbf{P}$. The next Theorem is analogous to Theorem 20.

THEOREM 21
$L \in \oplus\mathbf{P}$ iff L can be computed by a DC uniform circuit family $\{\mathcal{D}_n\}$ that

- *uses AND, OR, NOT, XOR gates.*
- *has size $2^{n^{O(1)}}$ and constant depth.*
- *XOR gates can have unbounded (exponential) fanin, but the AND, OR gates have fanin at most $n^{O(1)}$.*
- *NOT gates can appear anywhere in the circuit.*

Now define a randomized reduction between two languages.

DEFINITION 18 *Language A reduces to language B under a randomized polynomial time reduction, denoted $A \leq_r B$, if there exists a deterministic, polynomial time computable function $f : \{0,1\}^* \to \{0,1\}^*$ and a polynomial $p()$ such that*

$$\forall x \in A \qquad \mathbf{Pr}_{y \in \{0,1\}^{p(|x|)}}[f(x,y) \in B] \geq 2/3 \tag{8.1}$$

$$\forall x \notin A \qquad \mathbf{Pr}_{y \in \{0,1\}^{p(|x|)}}[f(x,y) \in B] \leq 1/3 \tag{8.2}$$

We prove Toda's Theorem in two steps. First we show that any $L \in \mathbf{PH}$ can be randomly reduced to $\oplus$SAT and then we show that this reduction can be transformed into a (deterministic) reduction to a $\#\mathbf{P}$ problem.

8.1 Random reduction from PH to $\oplus$SAT

In this section we prove our main lemma that every language in polynomial hierarchy is randomly reducible to $\oplus$SAT.

LEMMA 22
$\forall L \in PH, \quad L \leq_r \oplus SAT.$

We will need the Valiant-Vazirani lemma.

DEFINITION 19 *If $a_1, \ldots, a_k \in GF(2)^N$, then*

$$(a_1, \ldots, a_k)^{\perp} = \left\{ s \in GF(2)^N : \quad s \cdot a_1 = s \cdot a_2 = \ldots = s \cdot a_k = 0 \right\}.$$

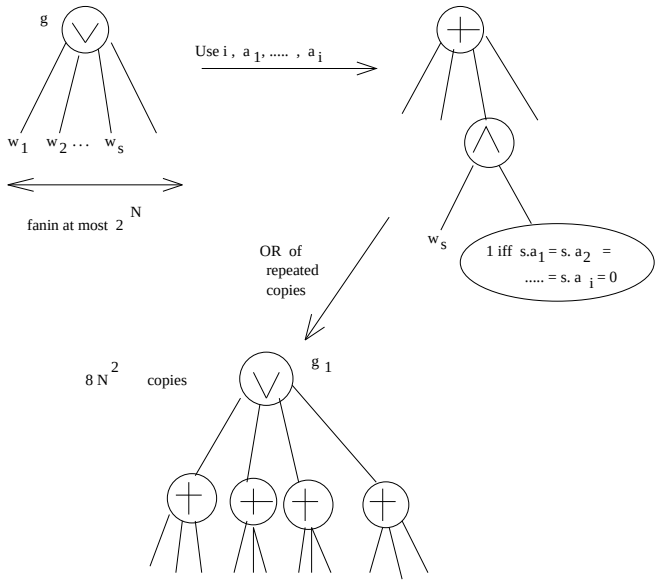


Figure 8.1: Probabilistic conversion of OR gate to $\oplus$ gate

Imagine that each vector in $GF(2)^N$ represents an individual, and $S \subseteq GF(2)^N$ is a hobby club. The Valiant Vazirani lemma considers a simple way of picking representatives from S: pick $i \in [0, N]$ randomly and then $a_1, a_2, \ldots, a_i \in GF(2)^N$ randomly and choose the individuals in $S \cap (a_1, \ldots, a_i)^\perp$ as the representatives. The lemma shows that this simple method has a probability at least $1/4N$ of yielding a unique representative for the club. The beauty of course is that we not need to know what S is. The proof of the Lemma is left as an exercise. (Note: Below, we only need (8.3) with the "1"replaced with "is an odd number," which seems a weaker fact but seems no easier to prove.)

Lemma 23 (Valiant-Vazirani)
If $S \subseteq GF(2)^N$ is nonempty and $i \in \{0, 1, \ldots, N\}$ and $a_1, \ldots, a_i \in GF(2)^N$ are chosen uniformly at random then

$$\mathbf{Pr}[\Big|S \cap (a_1, \ldots, a_i)^\perp\Big| = 1] \geq \frac{1}{4N}. \tag{8.3}$$

Now we prove Lemma 22.

Proof:(Lemma 22) Theorem 20 gives a characterization of L in terms of DC uniform circuit family $\{\mathcal{C}_n\}$. We use a randomized reduction to transform $\{\mathcal{C}_n\}$ into another DC uniform family $\{\mathcal{D}_n\}$ which uses $\oplus$ gates with exponential fanin and AND, OR gates with fanin $n^{O(1)}$. Theorem 21 then shows that the language computed by $\{\mathcal{D}_n\}$ can be expressed as a $\oplus$SAT problem (since $\oplus$SAT is complete for $\oplus\mathbf{P}$).

The reduction will use the Valiant-Vazirani lemma, and the deep fact that 1 is an odd number and 0 is even. Let the size of circuit $\mathcal{C}_n$ be 2^N where $N = n^{O(1)}$. Consider conversion of an OR gate g into XOR gate (Figure 1). The inputs $w_1, w_2, \ldots$ to the gate g can be indexed by strings in $GF(2)^N$. We choose i randomly from $\{0, 1, \ldots, N\}$ and strings $a_1, \ldots, a_i$ randomly from $GF(2)^N$. Then we replace the gate g by an $\oplus$ gate with the same

number of inputs, but the input w_s is replced by logical AND of w_s and a predicate p_s which is 1 iff $s \cdot a_1 = \ldots = s \cdot a_i = 0$. (This predicate is computable by a poly(n) size circuit.) We note that if output of gate g is 0, so is output of the $\oplus$ gate. If the output of gate g is 1, there is a nonempty set S of strings s such that $w_s = 1$. With probability $\geq 1/4N$ (over the choice of $i, a_1, \ldots, a_i$, the number of strings $s \in S$ such that $s \in (a_1, \ldots, a_i)^{\perp}$ is odd and the output of $\oplus$ gate is 1. Thus the $\oplus$ gate is equivalent to gate g with probability at least $1/4N$. We can amplify this probability by taking OR of $8N^2$ independent copies of this $\oplus$ gate. The resulting circuit g_1 satisfies,

$$\mathbf{Pr}[g_1(w_1, w_2 \ldots) \neq g(w_1, w_2, \ldots)] \leq (1 - 1/4N)^{8N^2} \leq e^{-2N}. \tag{8.4}$$

We apply this transformation to every gate in circuit $\mathcal{C}_n$. (The AND gates are first transformed into OR's using De-Morgan's law.) We use the same random choice of $i, a_1, \ldots, a_i$ for every gate, thus using only $n^{O(1)}$ random bits total. The new circuit $\mathcal{D}_n$ depends on the choice of the random bits. There are only 2^n different inputs for the circuit, and each imposes a set of values ($w_1, w_2, \ldots$, in the above description) on every gate. Since each gate transformation fails with probability at most e^{-2N}, we conclude that the old circuit agrees with $\mathcal{C}_n$ on all inputs of lengh n with probability $\geq 1 - 2^n 2^N e^{-2N} \geq 2/3$. It is clear that $\{\mathcal{D}\}_n$ is a DC uniform family satisfying conditions of Theorem 21. As observed before, we can construct a $\oplus$SAT question that decides whether an input is accepted by circuit $\mathcal{D}_n$. $\square$

8.2 Proof of Toda's Theorem

For $L \in PH$, Lemma 22 gives a reduction $x \to \phi_r$ from input x to a SAT formula ϕ_r using a random string r of length m such that

$$x \in L \Longrightarrow \mathbf{Pr}_r[\phi_r \in \oplus\text{SAT}] \geq 2/3$$

$$x \notin L \Longrightarrow \mathbf{Pr}_r[\phi_r \in \oplus\text{SAT}] \leq 1/3$$

We will construct a $\#\mathbf{P}$ question that counts the number of random strings r such that $\phi_r \in \oplus$SAT. Using a $\#\mathbf{P}$-oracle, we then know the probability $\mathbf{Pr}_r[\phi_r \in \oplus\text{SAT}]$ and we can decide whether $x \in L$, proving Toda's theorem.

We define a transformation of ϕ_r to another SAT formula ϕ_r^* such that

$$\phi_r \in \oplus\text{SAT} \Longrightarrow \#(\phi_r^*) \equiv -1 (mod\ 2^{m+1})$$

$$\phi_r \notin \oplus\text{SAT} \Longrightarrow \#(\phi_r^*) \equiv 0\ (mod\ 2^{m+1}).$$

Here $\#(\psi)$ is the number of satisfying assignments to formula ψ.

Now we construct a NP machine that guesses a random string r, computes ϕ_r and ϕ_r^*, branches out according to an assignment for ϕ_r^* and then checks whether this assignment satisfies ϕ_r^*. Clearly,

$$\#accepting\ paths = \sum_r \#(\phi_r^*) \tag{8.5}$$

$$\equiv -|\{r|\phi_r \in \oplus\text{SAT}\}|\ (mod\ 2^{m+1}) \tag{8.6}$$

Since there are only 2^m random strings r, the number of accepting paths reduced modulo 2^{m+1} gives the number of random strings r such that $\phi_r \in \oplus$SAT. By definition of $\#\mathbf{P}$, a $\#\mathbf{P}$-oracle can tell the number of accepting paths of this NTM on given input and the proof is complete.

We now give the transformation $\phi \to \phi^*$. For formulas $\varphi(x_1, \ldots, x_n)$ and $\psi(y_1, \ldots, y_m)$, let $\varphi + \psi$ and $\varphi \cdot \psi$ be formulas such that

$$\#(\varphi + \psi) = \#(\varphi) + \#(\psi)$$
$$\#(\varphi \cdot \psi) = \#(\varphi) \cdot \#(\psi)$$

Such formulas can be constructed easily with only constant factor increase in size. One can easily check that

$$\#(\psi) \equiv -1 (mod\ 2^{2^i}) \Longrightarrow \#(4\psi^3 + 3\psi^4) \equiv -1 (mod\ 2^{2^{i+1}})\ and \tag{8.7}$$
$$\#(\psi) \equiv 0\ (mod\ 2^{2^i}) \Longrightarrow \#(4\psi^3 + 3\psi^4) \equiv 0\ (mod\ 2^{2^{i+1}}). \tag{8.8}$$

Let $\phi_0 = \phi$ and $\phi_{i+1} = 4\phi_i^3 + 3\phi_i^4$. Let

$$\phi^* = \phi_{\lceil \log(m+1) \rceil}$$

Repeated use of equations 8.7, 8.8 shows that if $\#(\phi)$ is odd, $\#(\phi^*) \equiv -1\ (mod\ 2^{m+1})$ and if $\#(\phi)$ is even, $\#(\phi^*) \equiv 0\ (mod\ 2^{m+1})$. Also, size of ϕ^* is only polynomially larger than size of ϕ.

8.3 Open Problems

- What is the exact power of $\oplus$SAT and $\#SAT$?
- What is the complexity of computing approximation to permanent?
- What is the average case complexity of $n \times n$ permanent modulo small prime, say 3 or 5 ? Note that for a prime $p > n$, random self reducibility of permanent implies that if permanent is hard to compute on at least one input then it is hard to compute on $1 - O(p/n)$ fraction of inputs, i.e. hard to compute on average [**?**].

Exercises

§1 Prove Theorems 20 and 21.

§2 Prove the Valiant-Vazirani Lemma.

§3 Describe the constructions of the formulae $\varphi + \psi$ and $\varphi \cdot \psi$ as used in Toda's construction. (Be careful; φ and ψ could have different number of variables.)

§4 Let $f, g : \{0,1\}^* \to \mathbf{N}$ be functions and $c > 1$. We say that f *approximates* g *within a factor* c if for every string x, $g(x) \leq f(x) \leq c \cdot g(x)$. Show that for every $g \in \#\mathbf{P}$ and every $\epsilon > 0$, there is a function in $FP^{\mathbf{\Sigma}_3^p}$ that approximates g within a factor $1 + \epsilon$. (Hint: Use hashing and ideas similar to those in the proof of $\mathbf{BPP} \subseteq \mathbf{PH}$.)

Chapter 9

One-way functions and hard-core bit theorem

SCRIBE: *Edith Elkind*

9.1 Examples and definitions

DEFINITION 20 (INFORMAL DEFINITION) *A family of functions* $\{f_n : \{0,1\}^n \mapsto \{0,1\}^n\}$ *is called* one-way*, if* f_n *are*

- *easy to compute, but*
- *hard to invert for "many" inputs.*

EXAMPLE 7 $f(p,q) = pq$, where pq is n bit long. This function seems hard to invert, when p and q are roughly the same size and $p \equiv q \equiv 3 \pmod 4$. By Prime Number Theorem, about $1/n^2$ of all integers of n bits are products of two such primes.

DEFINITION 21 (FORMAL DEFINITION) *A family of functions* $\{f_n : \{0,1\}^n \mapsto \{0,1\}^n\}$ *is* one-way *with* security $s(n)$ *if there is a polynomial time Turing machine that computes them and furthermore for every algorithm* A *that runs in time* $s(n)$*,*

$$\mathbf{Pr}_{x \in \{0,1\}^n}[A \text{ inverts } f_n(x)] \leq \frac{1}{s(n)}. \tag{9.1}$$

REMARK 2 Here by inverting a (possibly many-to-one) function we mean finding any preimage of a given element. Also, we can define one-way functions with respect to other classes of adversaries, such as probabilistic Turing machines, or deterministic circuits.

A more general definition of a one-way function would allow the inversion probability of the adversary to be a general function $\delta(n)$, instead of demanding that it should be at most $1/s(n)$. In fact, it may seem risky to use our stringent definition, since a function satisfying it may not exist! For example, the multiplication function in Example 7 is hard to invert only occasionally (on $1/n^2$ of the outputs, as noted) and hence does not satisfy Definition 21

with even $S(n) = n$. Luckily, Yao has proven for us (although we will not show this) that if factoring or some other function is hard to invert on $1/\text{poly}(n)$ fraction of inputs then a one-way function exists that is hard to invert on almost all inputs, and hence satisfies Definition 21 with some $s(n) = \Omega(n^c)$ for every $c > 1$..

9.2 Goldreich-Levin Theorem

A a one-way function family $\{f_n\}$ is from $\{0,1\}^n$ to $\{0,1\}^n$ is called a *one-way permutation* if each f_n is one-to-one and onto.

EXAMPLE 8 The following is a conjectured one-way permutation. Let $p_1, p_2, \ldots$ be a sequence of primes where p_i has i bits. Let g_i be the generator of the group Z_p^* the set of numbers that're nonzero mod p. Since g_i is a generator, for every $y \in 1, .., p_i - 1$, there is a unique $x \in \{1, .., p-1\}$ such that

$$g_i^x \equiv y \pmod{p_i}.$$

Then $x \to g_i^x \pmod{p_i}$ is a permutation on $1, .., p_i - 1$ and is conjectured to be one-way. The inversion problem is called the *Discrete Log* problem.

Consider two random strings x and r, $|x| = |r| = n$. We use f as a shorthand for f_n. Consider the string $(f(x), r, x \odot r)$, where $x \odot r$ denotes the scalar product of x and r. As f is a permutation, $f(x)$ is information-theoretically equivalent to x, so the last bit of the concatenated string is completely determined by the first $2n$ bits. However, it turns out that this string looks completely random to any reasonable adversary.

THEOREM 24 (GOLDREICH, LEVIN '86)
Suppose that f_n is a one-way permutation and has security $s(n)$. Then for all algorithms A running in time $s^{1/4}(n)$

$$\mathbf{Pr}_{x,r\in\{0,1\}^n}[A(f_n(x), r) = x \odot r] \leq \frac{1}{2} + O(\frac{1}{s(n)}). \tag{9.2}$$

PROOF: Suppose that A can predict $x \odot r$ with probability $1/2 + \delta$. We show how to invert $f_n(x)$ for $O(\delta)$ fraction of the inputs in $O(n/\delta^2)$ time, from which the theorem follows.

LEMMA 25
Suppose that

$$\mathbf{Pr}_{x,r\in\{0,1\}^n}[A(f_n(x), r) = x \odot r] \geq \frac{1}{2} + \delta. \tag{9.3}$$

Then for at least δ fraction of x's

$$\mathbf{Pr}_{r\in\{0,1\}^n}[A(f_n(x), r) = x \odot r] \geq \frac{1}{2} + \frac{\delta}{2}. \tag{9.4}$$

PROOF: We use an averaging argument. Suppose that p is the fraction of x's satisfying (9.4). We have $p \cdot 1 + (1-p)(1/2 + \delta/2) \geq 1/2 + \delta$. Solving this with respect to p, we obtain

$$p \geq \frac{\delta}{2(1/2 - \delta/2)} \geq \delta.$$

□

We construct an inversion algorithm that given $f_n(x)$, where $x \in_R \{0,1\}^n$, will try to recover x. It "succeeds" with high probability if x is such that (9.4) holds. Note that the algorithm can always check the correctness of its answer, since it has $f_n(x)$ available to it and it can apply f_n to its answer and see if this gives f_n.

Today, we give the proof for the simpler case (which nevertheless contains all essential ideas) when $\mathbf{Pr}_{r\in\{0,1\}^n}[A(r) = x \odot r] \geq \frac{3}{4} + \delta$. In this case, for each i, $i = 1, \ldots, n$, we pick a random string r and query A to obtain $A(r)$ and $A(r \oplus e_i)$, where e_i is the ith basis vector, and $\oplus$ denotes addition modulo 2. We know that $A(r)$ is "often" equal to $x \odot r$ and $A(r \oplus e_i)$ is "often" equal to $x \odot (r \oplus e_i)$, so with a high enough probability $A(r) \oplus A(r \oplus e_i) = (x \odot r) \oplus (x \odot (r \oplus e_i) = x \odot e_i = x_i$. More formally, the algorithm guesses that $x_i = A(r) \oplus A(r \oplus e_i)$ and $\mathbf{Pr}_r[\text{the guess for } x_i \text{ is incorrect}] \leq 2(1/4 - \delta) = 1/2 - 2\delta$. (Here we use the fact that if r is random, then $r \oplus e_i$ is random as well, and apply the union bound). We can repeat this experiment sufficient number of times and take the majority vote to amplify the probability of guessing correctly. Furthermore, the probability of guessing the whole word correctly can then be bounded from below by using the union bound once again. □

-11

Chapter 10

One-way permutations and Goldreich-Levin bit theorem

SCRIBE: *Edith Elkind*

10.1 Proof of Goldreich–Levin theorem (continued)

In the last lecture, we formulated Goldreich–Levin theorem, which says that if $\{f_n\}$ is a one-way permutation, then the mapping $(x, r) \mapsto (f(x), r, x \odot r)$ extends $2n$ bits to $2n + 1$ bits in a pseudorandom fashion. More formally, for all algorithms A running in time $s^{1/4}(n)$

$$\mathbf{Pr}_{x,r\in\{0,1\}^n}[A(f_n(x), r) = x \odot r] \leq \frac{1}{2} + O\left(\frac{1}{s(n)}\right). \tag{10.1}$$

PROOF:[continued from Lecture 9] Last time, we gave a proof for the case when RHS of (10.1) is $3/4 + \delta$. The idea for the general case is very similar, the only difference being that this time we want to pick $r_1, \ldots, r_m$ so that we already "know" $x \odot r_i$. The preceding statement may appear ridiculous, since knowing the inner product of x with $m \geq n$ random vectors is, with high probability, enough to reconstruct x (check this!). The catch will of course be that the r_i's will not be completely random. Instead, they will be pairwise independent.

DEFINITION 22 *Random variables $x_1, \ldots, x_m$ are* pairwise independent *if*

$$\forall i, j \ \forall a, b \ \mathbf{Pr}[x_i = a, x_j = b] = \mathbf{Pr}[x_i = a]\mathbf{Pr}[x_j = b]. \tag{10.2}$$

Pairwise independent random variables are useful because of the Chebyshev inequality. Suppose that $x_1, \ldots, x_m$ are pairwise independent with $E(x_i) = \mu$, $Var(x_i) = \sigma^2$. Then we have

$$Var(\sum x_i) = E[(\sum_i x_i)^2] - E[\sum_i x_i]^2 = \sum_i (E[x_i^2] - E[x_i]^2),$$

where we have used pairwise independence in the last step. Thus $Var(\sum x_i) = m\sigma^2$. By the Chebyshev inequality, $\mathbf{Pr}[|\sum x_i - m\mu| > k\sqrt{m}\sigma] \leq 1/k^2$. So, the sum of the variables is somewhat concentrated about the mean. This is in contrast with the case of complete

independence, when Chernoff bounds would give an exponentially stronger concentration result (the $1/k^2$ would be replaced by $\exp(\Theta(-k^2))$).

EXAMPLE 9 In $\mathbf{Z}_p$, choose a and b randomly and independently. Then the random variables $a+b, a+2b, \ldots, a+(p-1)b$ are pairwise independent. Indeed, for any $t, s, j \neq k \in \mathbf{Z}_p$, $\mathbf{Pr}[a+jb=t] = 1/p$, and $\mathbf{Pr}[a+jb=t, a+kb=s] = 1/p^2$, because this linear system is satisfied by exactly one (a,b)-pair out of p^2.

EXAMPLE 10 Let $m = 2^k - 1$. The set $[1 \ldots m]$ is in $1-1$ correspondence with the set $2^{[1\ldots k]} \setminus \emptyset$. We will construct m random variables corresponding to all nonempty subsets of $[1 \ldots k]$. Pick uniformly at random k binary strings $t_1, \ldots, t_k$ of length n and set $Y_S = \sum_{i \in S} t_i \pmod 2$, where $S \subset [1 \ldots k]$, $S \neq \emptyset$. For any $S_1 \neq S_2$, the random variables Y_{S_1} and Y_{S_2} are independent, because one can always find an i such that $i \in S_2 \setminus S_1$, so the difference between Y_{S_1} and Y_{S_2} is always a sum of several uniformly distributed random vectors, which is a uniformly distributed random vector itself. That is, even if we fix the value of Y_{S_1}, we still have to toss a coin for each position in Y_{S_2}, and

$$\mathbf{Pr}[Y_{S_1} = \vec{s}, Y_{S_2} = \vec{t}] = \mathbf{Pr}[Y_{S_1} \oplus Y_{S_2} = \vec{t} \oplus \vec{s} | Y_{S_1} = \vec{s}] = \frac{1}{2^{2n}}.$$

Now let us return to the proof of Goldreich–Levin theorem and describe the observation at the heart of the proof. Suppose that our random strings $r_1, \ldots, r_m$ are $\{Y_S\}$ from the previous example. Then $x \odot Y_S = x \odot (\sum_{i \in S} t_i) = \sum_{i \in S} x \odot t_i$. Now, if we know $x \odot t_i$ for $i = 1, \ldots, k$, we also know $x \odot Y_S$. Of course, we don't know $x \odot t_i$ for $i = 1, \ldots, k$, but we can just try all 2^k possibilities for this vector and run the rest of the algorithm for each of them. This multiplies the running time by a factor 2^k, which is only m. This is how we can assume that we know $x \odot Y_S$ for each subset S.

The details of the rest of the algorithm are similar to before. By Lemma 2, we know that if the theorem were not true, then for at least $\delta = \alpha/s(n)$ fraction of x's, where α is some constant, the probability over r that A gives the correct answer is at lest $1/2 + \delta/2$. We now concentrate our attention on those x's. Pick m pairwise independent vectors Y_S's as described above, calculate $A(f_n(x), Y_S \oplus e_i) - x \odot Y_S$, and take the majority vote. The expected number of correct answers is $m(1/2+\delta/2)$, so for the majority vote to result in the incorrect answer it must be the case that the number of incorrect values deviates from its expectation by more than $m\delta/2$. Now, we can bound the variance of this random variable and apply Chebyshev's inequality.

Formally, let ξ_S denote the event that A produces the correct answer on Y_S; we have $E(\xi_S) = 1/2+\delta/2$ and $Var(\xi_S) = E(\xi_S)(1-E(\xi_S)) < 1$. Let $\xi = \sum_S \xi_S$ denote the number of correct answers on a sample of size m. By linearity of expectation, $E[\xi] = m(1/2+\delta/2)$. Furthermore, the Y_S's are pairwise independent, which implies that the same is true for the outputs ξ_S's produced by the algorithm A on them. Hence by pairwise independence $Var(\xi) < m$. Now, by Chebyshev's inequality, the probability that the majority vote is incorrect is at most $\frac{4Var(\xi)}{m^2\delta^2} \leq \frac{4}{m\delta^2}$. Recalling that $\delta = \alpha/s(n)$, we see that if we set $m = \Omega(ns^2(n))$, the probability of guessing the ith bit incorrectly is at most $1/2n$, and by the union bound, the probability of guessing the whole word incorrectly is at most $1/2$. Hence, on a "good" x, we can find the preimage of $f(x)$ with a good probability, and the

number of "good" x's is non-negligible, which contradicts our assumption that f is one-way. □

10.2 Applications

10.2.1 Playing poker over the phone

How can two parties A and B play poker over the phone? Specifically, how do they deal the cards in a fair way? Clearly, they need a way to toss a fair coin over the phone. If only one of them actually tosses a coin, there is nothing to prevent him from lying about the result. The following fix suggests itself: both players toss a coin and they take the XOR as the shared coin. Even if B does not trust A to use a fair coin, he knows that as long as his bit is random, the XOR is also random. Unfortunately, this idea does not work because the player who reveals his bit first is at a disadvantage: the other player could just "adjust" his answer to get his desired coin toss.

This problem is addressed by the following scheme, which assumes that A and B are polynomial time turing machines that cannot invert one-way permutations. The protocol itself is called *bit commitment*. First, A chooses two strings x_A and r_A of length n and sends a message $(f_n(x_A), r_A)$, where f_n is a one-way permutation. This way, A commits the string x_A without revealing it. Similarily, B chooses x_B and r_B and sends A a message $(f_n(x_B), r_B)$. After that, both parties are ready to reveal their strings, so A sends B a message $(x_A, x_A \odot r_A)$, and B sends A a message $(x_B, x_B \odot r_B)$. Here, $x_A \odot r_A$ serves as A's coin toss; B can verify that x_A is the same as in the first message by applying f_n, therefore A cannot change her mind after learning B's bit. On the other hand, by Goldreich–Levin theorem, B cannot predict $x_A \odot r_A$ from A's first message, so this scheme is secure.

Note that the second stage of this protocol is redundant: B can simply announce his bit after receiving A's first message.

10.2.2 Pseudorandom generation

Now we describe another application of one-way functions: to "stretch" n truly random bits to obtain n^c random-looking bits?

First, we have to define what it means for a string to look random. Kolmogorov gave one definition ("the length of the smallest Turing machine that outputs this string when started on an empty tape") but that is not very useful because of noncomputability issues. Blum and Micali proposed instead that we should define randomness for *distributions* rather than for strings. The distribution is declared pseudorandom if its samples "look" random to every polynomial time Turing machine. The next definition (due in this form to Yao) formalizes this notion.

Definition 23 (Yao'82) *A family* $\{g_n\}$, $g_n \{0,1\}^n \mapsto \{0,1\}^{n^c}$, *is called a* pseudorandom generator *if for any algorithm A running in time $s(n)$ and for all large enough n*

$$|\mathbf{Pr}_{y\in\{0,1\}^{n^c}}[A(y) = accept] - \mathbf{Pr}_{x\in\{0,1\}^n}[A(g_n(x)) = accept]| \le \delta(n), \tag{10.3}$$

where $\delta(n)$ is the distinguishing probability *and $s(n)$ is the* security parameter.

Suppose that f_n is a one-way permutation with security $s(n)$, and distinguishing probability $1/s(n)$, say. Then a pseudorandom generator can be built as follows. Take $2n$ random bits; denote the first n bits by x and the last n bits by r. Construct a string $(f_n(x), r, x \odot r)$; repeat the same procedure with the last $2n$ bits of this string; keep doing this until you get n^c bits.

THEOREM 26
No algorithm running in time $\frac{s(n)^{1/4}}{n^c}$ can distinguish a string obtained in this way from a random string with probability higher than $\frac{2n^c}{s(n)}$.

PROOF:[Yao's hybrid argument] By D_0 denote the distribution $\{g(z)\}$, $z \in \{0,1\}^{2n}$, and by F denote the uniform distribution on n^c bits. We are going to construct a sequence of distributions that starts with D_0 and gradually transforms it to F. Namely, let D_i, $i = 1, \ldots, n^c$ be a distribution in which the first i bits are random, while other bits are obtained by applying g to the $2n$ bits that precede them, just as we did for D_0 (or, if $i \leq 2n$, the bits from $i+1$st to $2n$th position are the last $2n - i$ bits of $f_n(z)$). Note that $D_{n^c} = F$. Also, since f_n is a permutation, the first $2n$ bits of D_0 are uniformly distributed, so $D_1 = \ldots = D_{2n} = D_0$.

Now, suppose that there is an algorithm that can distinguish between D_0 and D_{n^c}. Then this algorithm can also distinguish between D_i and D_{i+1} for some i, which means that it can predict Goldreich–Levin bit. More formally, suppose that there is an algorithm A such that

$$|\mathbf{Pr}_{y \in D_{n^c}}[A(y) = \text{accept}] - \mathbf{Pr}_{y \in D_0}[A(y) = \text{accept}]| \geq \delta. \tag{10.4}$$

Then there exists an i such that

$$|\mathbf{Pr}_{y \in D_i}[A(y) = \text{accept}] - \mathbf{Pr}_{y \in D_{i+1}}[A(y) = \text{accept}]| \geq \frac{\delta}{n^c}. \tag{10.5}$$

The only difference between D_i and D_{i+1} is in the ith bit: in D_i, it is obtained from $2n$ previous bits by Goldreich–Levin construction, while in D_{i+1} it is random.

Consider an algorithm B that given $f(x)$, r, and a bit b_0 (allegedly, $x \odot r$) constructs a string that starts with $i - 2n$ random bits followed by $f(x)$, r, and b_0; the remaining bits are produced as described above. Obviously, for random x and r if, indeed, $b_0 = x \odot r$, this string is distributed according to D_i, while if b_0 is uniformly distributed over $\{0,1\}$, this string is distributed according to D_{i+1}. Then B runs A on this string and accepts iff A accepts.

We have $\mathbf{Pr}_{x,r,r'}[B(f(x), r, x \odot r) = \text{accept}] = p_1$, $\mathbf{Pr}_{x,r,b,r'}[B(f(x), r, b) = \text{accept}] = p_2$, where r' stands for B's internal coin tosses. Without loss of generality, $p_2 < p_1$, hence $p_2 < p_1 - \delta/n^c$. Note that $\mathbf{Pr}_{x,r,b}[B(f(x), r, b) = \text{accept}] = \mathbf{Pr}_{x,r}[B(f(x), r, x \odot r) = \text{accept}] \times \frac{1}{2} + \mathbf{Pr}_{x,r}[B(f(x), r, \overline{x \odot r}) = \text{accept}] \times \frac{1}{2}$, nence if $p_2 < p_1 - \delta$, then $\mathbf{Pr}_{x,r}[B(f(x), r, \overline{x \odot r}) = \text{accept}] \leq p_1 - 2\delta$. An averaging argument similar to the one given in the previous lecture shows that for a non-negligible fraction of x's we can find the Goldreich-Levin bit with a significant probability by the following procedure. Run B several times and accept if B accepts on at least $p_1 - \delta$ fraction of the input. The correctess of this approach can be justified by Chernoff or Chebyshev inequality. □

As a corollary, Yao could give another definition of pseudorandom generators: they are distributions that pass the "next bit test" (i.e., that given the first i bits, a polynomial time algorithm cannot predict the $i+1$th bit with good probability).

COROLLARY 27
A pseudorandom generator is secure if and only if it passes the next bit test.

PROOF: A hybrid argument again, only this time the random bits are shifted in from right to left. □

THEOREM 28
If there is a pseudorandom generator that is secure against circuits of size n^c, then $\mathbf{BPP} \subseteq \cap_{\varepsilon>0}\mathbf{DTIME}(2^{n^\varepsilon})$.

In words, pseudorandom generators imply subexponential algorithms for **BPP**. For this reason, this theorem is usually referred to as *derandomization* of **BPP**.
PROOF: Let us fix an $\varepsilon > 0$ and show that $\mathbf{BPP} \subseteq \mathbf{DTIME}(2^{n^\varepsilon})$.

Suppose that M is a **BPP** machine running in n^k time. We can build another probabilistic machine M' that takes n^ε random bits, streches them to n^k bits using the pseudorandom generator and then simulates M using this n^k bits as a random string. Obviously, M' can be simulated by going over all binary strings n^ε, running M' on each of them, and taking the majority vote.

It remains to prove that M and M' accept the same language. Suppose otherwise. Then there exists an infinite sequence of inputs $x_1, \ldots, x_n, \ldots$ on which M distinguishes a truly random string from a pseudorandom string with a high probability, because for M and M' to produce different results, the probability of acceptance should drop from 2/3 to below 1/2. Hence we can build a distinguisher similar to the one described in the previous theorem by hardwiring these inputs into a circuit family. □

The relationship between hardness and randomness is a subject of recent research.

10.3 Average-case hardness

One-way functions and pseudorandom generators were covered in this course to give some idea of complexity theory as applied to the "average" case. Leonid Levin has developed a more general theory of average-case complexity. The main difference from the above study is that he defines a distribution over all (infinitely many) input strings rather than just a distribution over strings of a fixed length as we have done here (and which suffices for cryptography since typically the key-size used in a publicly available package such as DES is fixed).

Levin (1986) defines a distributional problem as a pair $(\pi, \mathcal{D})$, where π is a decision problem and $\mathcal{D}$ is a polynomial-time samplable distribution on inputs[1]. Levin's class *Avg-*

[1]The restriction to polynomial-time samplable distributions seems reasonable if we believe in the strong form of the Church-Turing thesis, which asserts that the probabilitic Turing machine can simulate every physically realizable computational model with polynomial slowdown. Then we can view the world as a simulatable system, and the instances produced by it it would then come from a polynomial-time samplable distribution.

P contains π for which $(\pi, \mathcal{D})$ has a polynomial-time average case algorithm for every $\mathcal{D}$. There are some subtleties in this definition that we will not explore.

Interestingly, Levin can prove that certain distributional problems are complete for this class under probabilistic polynomial time reductions. Many of these problems are specialized and not very natural. A very interesting and largely open problem is whether TSP or any other natural hard problem is complete (or hard) for Levin's class.

Chapter 11

Interactive Proofs

SCRIBE: *Carl Kingsford*

Recall the certificate definition of **NP**. We can think of this characterization of **NP** as an interaction between two entities P and V. For a language $L \in \mathbf{NP}$, x is in L if and only if P can send V a certificate (that depends on x) that V can use to check in polynomial time that x is indeed in L. P is the "prover" and V is the "verifier." For **NP** only one interaction is allowed. We can expand the power of the verifier TM: let it be a *probabilistic* TM that can make a *polynomial number* of queries of the prover. Now the verifier and the prover exchange a polynomial number of polynomial length messages. Both the prover and verifier see the input x. The prover is trying to convince the verifier that $x \in L$. Formally, we define the class **IP** of languages that have such interactive proofs:

DEFINITION 24 (**IP**) *A language L is in* **IP** *if there is probabilistic, polynomial-time TM V with coin flips r that interacts with an all-powerful prover, where in round i its query $q_i(x, r, a_1, \ldots, a_{i-1})$ depends on the input, the random string, and the prover's responses $a_1, \ldots, a_{i-1}$ in the previous rounds. The verifier has the property that:*

(i) $x \in L \Rightarrow \exists P \quad \Pr_r[V \text{ accepts } x \text{ after interaction with } P] \geq 2/3$

(ii) $x \notin L \Rightarrow \forall P \quad \Pr_r[V \text{ rejects } x \text{ after interacting with } P] \geq 2/3.$

Since P cannot see the coin flips of V, we say the protocol is private coin. *We further define* $\mathbf{IP}[k]$ *(for $k \geq 2$) be the set of all languages that have a k round interactive proof in this private coin model, where a "round" is either a query or a response.*

If V were not allowed to be probabilistic it is easy to see that **IP** is equivalent to **NP**: the prover can compute all the queries the verifier will make ahead of time and offer this entire transcript to the verifier straightaway. By allowing V random bits, we get a more powerful class.

The probabilities of correctly classifying an input can be made arbitrarily large by using the same boosting technique we used for **BPP**: sequentially repeat the protocol k times. If $x \in L$ then the same prover can be used on each repetition. If $x \notin L$ then on each repetition, the chance that the verifier will accept x is less than $1/3$.

We can define similar classes **AM** and $\mathbf{AM}[k]$ in which P *does* see the coin flips of V—this is the *public coin model*. We state the following comments about $\mathbf{IP}[\cdot], \mathbf{AM}[\cdot]$, without proof:

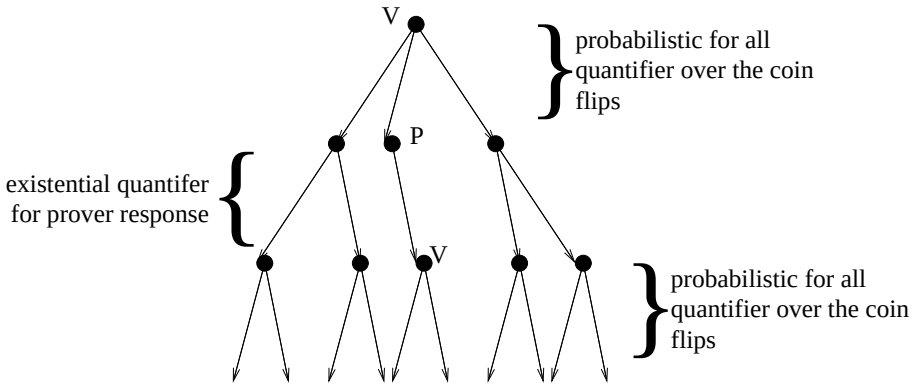


Figure 11.1: $\mathbf{AM}[k]$ looks like $\prod_k^p$

(i) $\mathbf{IP}[k] \subseteq \mathbf{AM}[k+2]$ for all constants k.

(ii) For constants $k \geq 2$ we have $\mathbf{AM}[k] = \mathbf{AM}[2]$. This is surprising because $\mathbf{AM}[k]$ seems similar to **PH** with the $\forall$ quantifiers changed to "probabilistic $\forall$" quantifiers, where *most* of the branches lead to acceptance. See figure 11.1.

It is open whether there is any nice characterization of $\mathbf{AM}[\sigma(n)]$, where $\sigma(n)$ is a suitably slow growing function of n, such as $\log \log n$.

(iii) Changing 2/3 to 1 in the definition of **IP** does not change the class. That is, defining the **IP** in a manner similar to **coRP** is equivalent to defining it like **BPP**.

Whereas **BPP** is a probabilistic version of **P**, $\mathbf{AM}[2]$ is as probabilistic version of **NP**. $\mathbf{AM}[2]$ can be thought of as "**BP-NP**"—languages for which there is a bounded probabilistic *nondeterministic* TM.

(iv) It is relatively easy to see that $\mathbf{AM}[poly(n)] \subseteq \mathbf{PSPACE}$; the proof is left as an exercise. It is possible, in **PSPACE**, to come up with a good strategy for choosing the $\exists$ edges in figure 11.1.

Let us look at an example of a language in **IP**.

EXAMPLE 11 (GRAPH NON-ISOMORPHISM) Given two graphs G_1 and G_2 we say they are isomorphic to each other if there is a permutation π of the labels of the nodes of G_1 such that $\pi G_1 = G_2$. The graphs in figure 11.2, for example, are isomorphic with $\pi = (12)(3654)$. If G_1 and G_2 are isomorphic, we write $G_1 \equiv G_2$. The GRAPH NON-ISOMORPHISM problem is this: given two graphs, are they *not* isomorphic?

It is clear that the complement of GRAPH NON-ISOMORPHISM is in **NP**— a certificate is simply the permutation π that is an isomorphism. What is more surprising is that GRAPH NON-ISOMORPHISM is in **IP**. To show this, we give a private-coin protocol that satisfies definition 24:

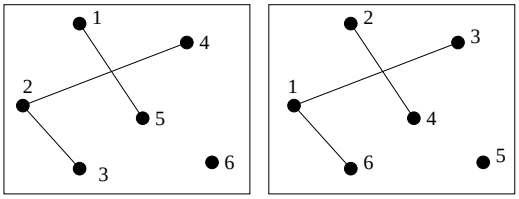


Figure 11.2: Two isomorphic graphs.

Protocol: Private-coin Graph Non-isomorphism

V: pick $i \in \{1, 2\}$ uniformly randomly. Randomly permute the vertices of G_i to get a new graph H. Send H to P.

P: identify which of G_1, G_2 was used to produce H. Let G_j be that graph. Send j to V.

V: accept if $i = j$; reject otherwise.

To see that definition 24 is satisfied by the above protocol, note that if $G_1 \not\equiv G_2$ then there exists a prover such that $\Pr[V \text{ accepts}] = 1$, because if the graphs are non-isomorphic, an all-powerful prover can certainly tell which one of the two is isomorphic to H. On the other hand, if $G_1 \equiv G_2$ the best any prover can do is to randomly guess, because a random permutation of G_1 looks exactly like a random permutation of G_2; that is $\Pr[V\text{accepts}] \leq 1/2$.

The above example depends crucially on the fact that P cannot see the random bits of V. If P knew those bits, P would know i and so could trivially always guess correctly. By comment (i), any problem with a private-coin interactive proof with a constant number of rounds has a public-coin proof. We now present such a protocol for Graph Non-isomorphism.

Example 12 (Public-coin protocol for Graph Non-isomorphism) To develop a public-coin protocol, we need to look at the problem in a different way. Consider the set $S = \{H : H \equiv G_1 \text{ or } H \equiv G_2\}$. The size of this set depends on whether G_1 is isomorphic to G_2. For a graph of n vertices, there are $n!$ possible ways to label the vertices, so we have

$$\text{if } G_1 \not\equiv G_2 \text{ then } |S| = 2n!$$
$$\text{if } G_1 \equiv G_2 \text{ then } |S| = n!$$

We can amplify the gap between the two cases. Choose an integer m such that $n! \leq 2^m$. Let S' be the Cartesian product of S with itself a sufficient number of times so that the following hold:

$$\text{if } G_1 \not\equiv G_2 \text{ then } \left|S'\right| \geq 100 \cdot 2^m \quad \text{(A)}$$
$$\text{if } G_1 \equiv G_2 \text{ then } \left|S'\right| \leq \frac{1}{10} 2^m \quad \text{(B)}$$

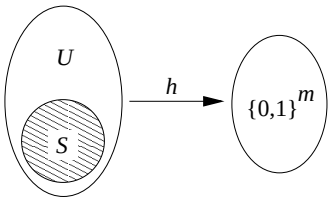


Figure 11.3: The bigger S' is, the more likely $h(S')$ will hit a given point in $\{0,1\}^m$.

We can now use the size of S' to determine if $G_1 \equiv G_2$. The question that remains is: how do we design an interactive protocol that can distinguish between cases (A) and (B) above?

Let $\mathcal{H}$ be a set of 2-universal hash functions from $\mathcal{U}$ to $\{0,1\}^m$, where $\mathcal{U}$ is a superset of S'. Given $h \in \mathcal{H}$, if S' is much bigger than $\{0,1\}^m$, as it is in case (A), then for every $y \in \{0,1\}^m$ it is very likely that there is some $x \in S'$ such that $h(x) = y$. Conversely, if the size of S' is much smaller than $\{0,1\}^m$, —if (B) holds— then for most $y \in \{0,1\}^m$ there is no $x \in S'$ such that $h(x) = y$. This is the basis of our protocol: V gives P an $h \in \mathcal{H}$ and $y \in \{0,1\}^m$, and if P can find an $x \in S'$ that h maps to y, the verifier guesses that the set is large, otherwise, we guess that the set is small. See figure 11.3. More formally, we have:

Protocol: Goldwasser-Sipser Set Lowerbound

V: Randomly pick $h \in \mathcal{H}$, and $y \in_R \{0,1\}^m$. Send h, y to P.

P: Try to find an $x \in S'$ such that $h(x) = y$. Send such an x to V, or send a random element in $\mathcal{U}$ if no such x exists.

V: If $x \in S'$ and $h(x) = y$, accept; otherwise reject.

It remains to be shown that the above protocol fits Definition 24. Suppose (B) holds, then since the domain (S') is 1/10th the size of the range ($\{0,1\}^m$), we have $\Pr[V \text{ accepts if } G_1 \equiv G_2] \leq 1/10$. To find the probability of acceptance if case (A) holds we need a lemma that tells us that if S' is bigger than a certain size, any hash function we pick will likely map S' to a large fraction its range. The following lemma's proof is left as an exercise:

Lemma 29
Suppose $S \geq \mu 2^m$. *Then* $\Pr_{h \in \mathcal{H}}\left[|h(S)| \geq \left(1 - \frac{1}{\sqrt{\mu}}\right) 2^m\right] \geq 1 - \frac{1}{\sqrt{\mu}}$.

Here we have $\mu = 100$. Even if the hash function that we choose in the protocol is good, there is still a chance $1/\sqrt{\mu}$ that we will pick a $y \in \{0,1\}^m$ that is not mapped to by S. Hence,

$$\Pr[V \text{ accepts if } G_1 \equiv G_2] \geq 1 - \frac{1}{\sqrt{100}} - \frac{1}{\sqrt{100}} = \frac{8}{10}.$$

That Graph Non-isomorphism $\in$ **AM** follows from the definition.

We now turn to our main theorem.

Theorem 30 (LFKN, Shamir, 1990)
IP = **PSPACE**.

PROOF: By comment (iv), we need only show that **PSPACE** $\subseteq$ **IP**[$poly(n)$]. To do so, we'll show that TQBF $\in$ **IP**[$poly(n)$]. This is sufficient because every $L \in$ **PSPACE** is polytime reducible to TQBF.

Again, we change the representation of the problem. For any Boolean formula of n variables $\phi(b_1, b_2, \ldots, b_n)$ there is a polynomial $P_\phi(x_1, x_2, \ldots, x_n)$ that is 1 if ϕ is true and 0 if ϕ is false. To see that this is true, consider the following correspondence between formulas and polynomials:

$$\begin{aligned} x \wedge y &\longleftrightarrow X \cdot Y \\ x \vee y &\longleftrightarrow 1 - (1 - X)(1 - Y) \\ \neg x &\longleftrightarrow 1 - X \end{aligned}$$

For example, $\phi = x \vee y \vee \neg z \longleftrightarrow 1 - (1 - X)(1 - Y)Z$. If ϕ is a 3CNF formula with n variables and m clauses then we can write such a polynomial for each clause and multiply those polynomials to get a polynomial P_ϕ in n variables, with degree at most m in each variable. This conversion of ϕ to P_ϕ is called *arithmetization* and will be useful in our protocol.

Rather than tackle the job of finding a protocol for TQBF right away, we first present a protocol for $\#\mathbf{SAT}_L$, where

$$\#\mathbf{SAT}_L = \{\langle \phi, K \rangle : K \text{ is the number of sat. assignments of } \phi\}.$$

and ϕ is a 3CNF formula of n variables and m clauses.

THEOREM 31
$\#\mathbf{SAT}_L \in$ **IP**

PROOF: Given ϕ, we construct, by arithmetization, P_ϕ. The number of satisfying assignments $\#\phi$ of ϕ is:

$$\#\phi = \sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} P_\phi(b_1, \ldots, b_n) \tag{11.1}$$

There is a general protocol, *Sumcheck*, for verifying equations such as (11.1).

Sumcheck protocol. Given a degree d polynomial $g(x_1, \ldots, x_n)$ and an integer K, we present an interactive proof for the claim

$$K = \sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(x_1, \ldots, x_n). \tag{11.2}$$

V simply needs to be able to arbitrarily evaluate g. Define

$$h(x_1) = \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(x_1, b_2 \ldots, b_n).$$

If (11.2) is true, it must be the case that $h(0) + h(1) = K$.

To start, V randomly picks a prime p in the interval $[n^3, n^4]$ and instructs the prover to reduce all numbers modulo p in the remaining protocol. (For instance, if the prover wants

to send a polynomial, it only needs to send the coefficients modulo p.) All computations described below are also modulo p. Consider the following protocol:

Protocol: Sumcheck protocol to check claim (11.2)

V: If $n = 1$ check that $g(1) + g(0) = K$. If so accept, otherwise reject. If $n \geq 2$, ask P to send $h(x_1)$ as defined above.

P: Send $h(x_1)$.

V: Let $s(x_1)$ be the polynomial received. Reject if $s(0)+s(1) \neq K$; otherwise pick a random a. Recursively use this protocol to check that

$$s(a) = \sum_{b \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(a, b_2, \ldots, b_n).$$

If Claim (11.2) is true, the prover that always returns the correct polynomial will always convince V. We prove by induction on n that if (11.2) is false, V rejects with high probability; we prove

$$\Pr[V \text{ rejects } \langle K, g \rangle] \geq \left(1 - \frac{d}{p}\right)^n. \tag{11.3}$$

With our choice of p, the right hand side is about $1 - dn/p$, which is very close to 1 since $p \geq n^3$.

We prove (11.3) by induction on n. The statement is true for $n = 1$ since V simply evaluates $g(0), g(1)$ and rejects with probability 1 if their sum is not K. Assume the hypothesis is true for degree d polynomials in $n - 1$ variables.

In the first round, the prover P is supposed to return the polynomial h. Suppose that P returns $s \neq h$. If $s(0) + s(1) \neq K$, then V rejects with probability 1. So assume that $s(0) + s(1) = K$. In this case, V picks a random a. If s and h are two different degree d polynomials, then there are at most d values of x_1 such that $s(x_1) = h(x_1)$. Thus,

$$\Pr_a[s(a) \neq h(a)] \geq 1 - \frac{d}{p}. \tag{11.4}$$

If $s(a) \neq h(a)$ then the prover is left with an incorrect claim to prove in the recursive step. By the induction hypothesis, with probability $\geq \left(1 - \frac{d}{p}\right)^{n-1}$, P cannot prove this false claim. Thus we have

$$\Pr[V \text{ rejects}] \geq \left(1 - \frac{d}{p}\right) \cdot \left(1 - \frac{d}{p}\right)^{n-1} = \left(1 - \frac{d}{p}\right)^n \tag{11.5}$$

This finishes the induction.

We still have to justify why it is OK to perform all operations modulo p. The fear, of course, is that equation (11.2) might be false over the integers, but true over p, which happens if p divides the difference of the two sides of (11.2). The following lemma implies that the chance that this happens for a random choice of p is small, since an n-bit integer (which is what K is) has at most n prime divisors.

LEMMA 32 (FINGERPRINTING)
Suppose x, y, $x \neq y$ are n-bit integers. Then there are at most n primes that divide $|x - y|$.

An interactive proof for $\#\mathbf{SAT}_L$ is obtained by letting $g = P_\phi$. □

We use a very similar idea to obtain a protocol for TQBF. Given a true, fully qualified Boolean formula $\exists x_1 \forall x_2 \exists x_3 \cdots \forall x_n \phi(x_1, \ldots, x_n)$, we use arithmetization to construct the polynomial P_ϕ. We have that $\phi \in$ TQBF if and only if

$$0 < \sum_{b_1 \in \{0,1\}} \prod_{b_2 \in \{0,1\}} \sum_{b_3 \in \{0,1\}} \cdots \prod_{b_n \in \{0,1\}} P_\phi(b_1, \ldots, b_n)$$

A first thought is that we could use the same protocol as in the $\#\mathbf{SAT}_L$ case, except check that $s(0) \cdot s(1) = K$ when you have a $\prod$. But, alas, multiplication, unlike addition, increases the degree of the polynomial — after n steps when V must evaluate g directly, the degree could be 2^n. There could be exponentially many coefficients. The solution is to observe that we are only interested in $\{0, 1\}$ values. You can always approximate a polynomial with a multi-linear function if you only evaluate it at $\{0, 1\}^n$. Let Rx_i be a linearization operator defined as

$$Rx_1[p(x_1, \ldots, x_n)] = (1 - x_1)p(0, x_2, \ldots, x_n) + (x_1)p(1, x_2, \ldots, x_n). \tag{11.6}$$

Now, instead of working with $\exists x_1 \forall x_2 \exists x_3 \cdots \forall x_n \phi(x_1, \ldots, x_n)$, work with

$$\exists x_1 Rx_1 \forall x_2 Rx_1 Rx_2 \exists x_3 Rx_1 Rx_2 R_x 3 \cdots \phi(x_1, \ldots, x_n) \tag{11.7}$$

The size of the expression is $1 + 2 + 3 + \cdots + n \approx n^2$. The protocol for $\#\mathbf{SAT}_L$ can be used suitably modified, where in rounds involving the linearization operator, the verifier uses (11.6). The use of the linearization operator ensures that the polynomial which the prover needs to send at every round is linear, and hence the blowup in degrees is avoided. □

Chapter 12

PCP Theorem and the hardness of approximation

SCRIBE: *None*

Lectures 14-16 gave an introduction to the PCP Theorem and its use in proving the hardness of approximation.

There are no scribe notes because we used some sources that were already available. A writeup by Arora and Trevisan is under preparation.

Chapter 13

Decision Tree Complexity

SCRIBE: *Jared Kramer*

From now on the topic in the course will be concrete complexity, the study of lower-bounds on models of computation such as decision trees, communication games, circuits, etc. Algorithms or devices considered in this lecture take inputs of a fixed size n, and we study the complexity of these devices as a function of n.

13.1 Decision Trees

A *decision tree* is a model of computation used to study the number of bits of an input that need to be examined in order to compute some function on this input. Consider a function $f : \{0,1\}^n \to \{0,1\}$. A decision tree for f is a tree for which each node is labelled with some x_i, and has two outgoing edges, labelled 0 and 1. Each tree leaf is labelled with an output value 0 or 1. The computation on input $x = x_1 x_2 \dots x_n$ proceeds at each node by inspecting the input bit x_i indicated by the node's label. If $x_i = 1$ the computation continues in the subtree reached by taking the 1-edge. The 0-edge is taken if the bit is 0. Thus input x follows a path through the tree. The output value at the leaf is $f(x)$. An example of a simple decision tree for the majority function is given in Figure 13.1

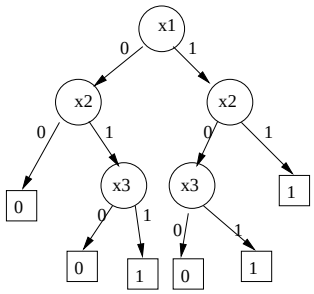


Figure 13.1: A decision tree for computing the majority function $Maj(x_1, x_2, x_3)$ on three bits. Outputs 1 if at least two input bits are 1, else outputs 0.

Recall the use of decision trees in the proof of the lower bound for comparison-based sorting algorithms. That study can be recast in the above framework by thinking of the

input —which consisted of n numbers — as consisting of $\binom{n}{2}$ bits, each giving the outcome of a pairwise comparison between two numbers.

We can now define two useful decision tree metrics.

DEFINITION 25 *The cost of tree t on input x, $cost(t, x)$, is the number of bits of x examined by t.*

DEFINITION 26 *The* **decision tree complexity** *of function f, $D(f)$, is defined as follows, where T below refers to the set of decision trees that decide f.*

$$D(f) = \min_{t \in T} \max_{x \in \{0,1\}^n} cost(t, x) \tag{13.1}$$

The decision tree complexity of a function is the number of bits examined by the most efficient decision tree on the worst case input to that tree. We are now ready to consider several examples.

EXAMPLE 13 (*Graph connectivity*) Given a graph G as input, in adjacency matrix form, we would like to know how many bits of the adjacency matrix a decision tree algorithm might have to inspect in order to determine whether G is connected. We have the following result.

THEOREM 33
Let f be a function that computes the connectivity of input graphs with m vertices. Then $D(f) = \binom{m}{2}$.

The idea of the proof of this theorem is to imagine an adversary that constructs a graph, edge by edge, in response to the queries of a decision tree. For every decision tree that decides connectivity, the strategy implicitly produces an input graph which requires the decision tree to inspect each of the $\binom{m}{2}$ possible edges in a graph of m vertices.

Adversary Strategy:
Whenever the decision tree algorithm asks about edge e_i,
answer "no" unless this would force the graph to be disconnected.

After i queries, let N_i be the set of edges for which the adversary has replied "no", Y_i the set of edges for which the adversary has replied "yes". and E_i the set of edges not yet queried. The adversary's strategy maintains the invariant that Y_i is a disconnected forest for $i < \binom{m}{2}$ and $Y_i \cup E_i$ is connected. This ensures that the decision tree will not know whether the graph is connected until it queries every edge.

EXAMPLE 14 (*OR Function*) Let $f(x_1, x_2, \ldots x_n) = \bigvee_{i=1}^{n} x_i$. Here we can use an adversary argument to show that $D(f) = n$. For any decision tree query of an input bit x_i, the adversary responds that x_i equals 0 for the first $n - 1$ queries. Since f is the OR function, the decision tree will be in suspense until the value of the nth bit is revealed. Thus $D(f)$ is n.

EXAMPLE 15 Consider the AND-OR function, with $n = 2^k$. We define f_k as follows.

$$f_k(x_1, \ldots, x_n) = \begin{cases} f_{k-1}(x_1, \ldots x_{2^{k-1}-1}) \wedge f_{k-1}(x_{2^{k-1}}, \ldots x_{2^k}) & \text{if } k \text{ is even} \\ f_{k-1}(x_1, \ldots x_{2^{k-1}-1}) \vee f_{k-1}(x_{2^{k-1}}, \ldots x_{2^k}) & \text{if } k > 1 \text{ and is odd} \\ x_i & \text{if } k = 1 \end{cases} \quad (13.2)$$

A diagram of a circuit that computes the AND-OR function is shown in Figure 13.2. It is left as an exercise to prove, using induction, that $D(f_k) = 2^k$.

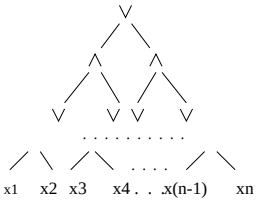


Figure 13.2: A circuit showing the computation of the AND-OR function. The circuit has k layers of alternating gates, where $n = 2^k$.

13.2 Certificate Complexity

We now introduce the notion of *certificate complexity*, which, in a manner analogous to decision tree complexity above, tells us the minimum amount of information needed to be convinced of the value of a function f on input x.

DEFINITION 27 *Consider a function $f : \{0,1\}^n \to \{0,1\}$. If $f(x) = 0$, then a* **0-certificate** *for x is a sequence of bits in x that proves $f(x) = 0$. If $f(x) = 1$, then a* **1-certificate** *is a sequence of bits in x that proves $f(x) = 1$.*

DEFINITION 28 *The* **certificate complexity** *$C(f)$ of f is defined as follows.*

$$C(f) = \max_{x:input} \{number\ of\ bits\ in\ the\ smallest\ 0\text{- or }1\text{- certificate for } x\} \quad (13.3)$$

EXAMPLE 16 If f is a function that decides connectivity of a graph, a 0-certificate for an input must prove that some cut in the graph has no edges, hence it has to contain all the possible edges of a cut of the graph. When these edges do not exist, the graph is disconnected. Similarly, a 1-certificate is the edges of a spanning tree. Thus for those inputs that represent a connected graph, the minimum size of a 1-certificate is the number of edges in a spanning tree, $n - 1$. For those that represent a disconnected graph, a 0 certificate is the set of edges in a cut. The size of a 0-certificate is at most $(n/2)^2 = n^2/4$, and there are graphs (such as the graph consisting of two disjoint cliques of size $n/2$) in which no smaller 0-certificate exists. Thus $C(f) = n^2/4$.

EXAMPLE 17 We show that the certificate complexity of the AND-OR function f_k of Example 15 is $2^{\lceil k/2 \rceil}$. Recall that f_k is defined using a circuit of k layers. Each layer contains

only OR-gates or only AND-gates, and the layers have alternative gate types. The bottom layer receives the bits of input x as input and the single top layer gate outputs the answer $f_k(x)$. If $f(x) = 1$, we can construct a 1-certificate as follows. For every AND-gate in the tree of gates we have to prove that both its children evaluate to 1, whereas for every OR-gate we only need to prove that *some* child evaluates to 1. Thus the 1-certificate is a subtree in which the AND-gates have two children but the OR gates only have one each. Thus the subtree only needs to involve $2^{\lceil k/2 \rceil}$ input bits. If $f(x) = 0$, a similar argument applies, but the role of OR-gates and AND-gates, and values 1 and 0 are reversed. The result is that the certificate complexity of f_k is $2^{\lceil k/2 \rceil}$, or about $\sqrt{n}$.

The following is a rough way to think about these concepts in analogy to Turing machine complexity as we have studied it.

$$\text{low decision tree complexity} \leftrightarrow \quad \mathbf{P} \tag{13.4}$$

$$\text{low 1-certificate complexity} \leftrightarrow \quad \mathbf{NP} \tag{13.5}$$

$$\text{low 0-certificate complexity} \leftrightarrow \quad \mathbf{coNP} \tag{13.6}$$

The following result shows, however, that the analogy may not be exact since in the decision tree world, $\mathbf{P} = \mathbf{NP} \cap \mathbf{coNP}$.

THEOREM 34
For function f, $D(f) \leq C(f)^2$.

It should be noted that there are some functions, such as the AND-OR function, for which this inequality is tight.

PROOF: Let S_0, S_1 be the set of minimal 0-certificates and 1-certificates, respectively, for f. Let $k = C(f)$, so each certificate has at most k bits.

REMARK 3 Note that every 0-certificate must share a bit position with every 1-certificate, and furthermore, assign this bit differently. If this were not the case, then it would be possible for both a 0-certificate and 1-certificate to be asserted at the same time, which is impossible.

The following decision tree algorithm then determines the value of f in at most k^2 queries.

Algorithm: Repeat until the value of f is determined: Choose a remaining 0-certificate from S_0 and query all the bits in it. If the bits are the values that prove the f to be 0, then stop. Otherwise, we can prune the set of remaining certificates as follows. Since all 1-certificates must intersect the chosen 0-certificate, for any $c_1 \in S_1$, one bit in c_1 must have been queried here. Eliminate c_1 from consideration if the certifying value of c_1 at at location is different from the actual value found. Otherwise, we only need to consider the remaining $k - 1$ bits of c_1.

This algorithm can repeat at most k times. For each iteration, the unfixed lengths of the uneliminated 1-certificates decreases by one. This is because once some values of the input have been fixed due to queries, for any 0-certificate, it remains true that all 1-certificates must intersect it in at least one location that has not been fixed, otherwise it would be

possible for both a 0-certificate and a 1-certificate to be asserted. With at most k queries for at most k iterations, a total of k^2 queries is used. □

13.3 Randomized Decision Trees

There are two equivalent ways to look at randomized decision trees. We can consider decision trees in which the branch taken at each node is determined by the query value and by a random coin flip. We can also consider probability distributions over deterministic decision trees. The analysis that follows uses the latter model.

We will call $\mathcal{P}$ a probability distribution over a set of decision trees $\mathcal{T}$ that compute a particular function. $\mathcal{P}(t)$ is then the probability that tree t is chosen from the distribution. For a particular input x, then, we define $c(\mathcal{P}, x) = \sum_{tin\mathcal{T}} \mathcal{P}(t) cost(t, x)$. $c(\mathcal{P}, x)$ is thus the expected number of queries a tree chosen from $\mathcal{T}$ will make on input x. We can then characterize how well randomized decision trees can operate on a particular problem.

DEFINITION 29 *The* **randomized decision tree complexity**, $\mathcal{R}(f)$, *of* f, *is defined as follows.*

$$\mathcal{R}(f) = \min_{\mathcal{P}} \max_{x} c(\mathcal{P}, x) \tag{13.7}$$

The randomized decision tree complexity thus expresses how well the best possible probability distribution of trees will do against the worst possible input for a particular probability distribution of trees. We can observe immediately that $\mathcal{R}(f) \geq C(f)$. This is because $C(f)$ is a minimum value of $cost(t, x)$. Since $\mathcal{R}(f)$ is just an expected value for a particular probability distribution of these *cost* values, the minimum such value can be no greater than the expected value.

EXAMPLE 18 Consider the majority function, $f = Maj(x_1, x_2, x_3)$. It is straightforward to see that $D(f) = 3$. We show that $\mathcal{R}(f) \leq 8/3$. Let $\mathcal{P}$ be a uniform distribution over the (six) ways of ordering the queries of the three input bits. Now if all three bits are the same, then regardless of the order chosen, the decision tree will produce the correct answer after two queries. For such x, $c(\mathcal{P}, x) = 2$. If two of the bits are the same and the third is different, then there is a 1/3 probability that the chosen decision tree will choose the two similar bits to query first, and thus a 1/3 probability that the cost will be 2. There thus remains a 2/3 probability that all three bits will need to be inspected. For such x, then, $c(\mathcal{P}, x) = 8/3$. Therefore, $\mathcal{R}(f)$ is at most 8/3.

How can we prove *lowerbounds* on randomized complexity? For this we need another concept.

13.4 Distributional Complexity

We now consider a related topic, distributional complexity. Where randomized complexity explores distributions over the space of decision trees for a problem, distributional complexity considers probability distributions on inputs. It is under such considerations that we can speak of "average case analysis." These two concepts turn out to be related in a useful

way by Yao's Lemma. Let $\mathcal{D}$ be a probability distribution over the space of input strings of length n. Then, if A is a deterministic algorithm, such as a decision tree, for a function, then we define the distributional complexity of A on a function f with inputs distributed according to $\mathcal{D}$ as the expected cost for algorithm A to compute f, where the expectation is over the distribution of inputs.

DEFINITION 30 *The* **distributional complexity** $d(A, \mathcal{D})$ *of algorithm A given inputs distributed according to* $\mathcal{D}$ *is defined as:*

$$d(A, \mathcal{D}) = \sum_{x:input} \mathcal{D}(x) cost(A, x) = \mathbf{E}_{x \in \mathcal{D}}[cost(A, x)] \tag{13.8}$$

From this we can characterize distributional complexity as a function of a single function f itself.

DEFINITION 31 *The* **distributional** *decision tree complexity,* $\Delta(f)$ *of function f is defined as:*

$$\Delta(f) = \max_{\mathcal{D}} \min_{A} d(A, \mathcal{D}) \tag{13.9}$$

Where A above runs over the set of decision trees that are deciders for f.

So the distributional decision tree complexity measures the expected efficiency of the most efficient decision tree algorithm works given the worst case distribution of inputs.

Yao's lemma relates distributional decision tree complexity to the earlier introduced notion of randomized decision tree complexity.

THEOREM 35
For all computational models in which both the set of inputs and the set of algorithms is finite, for any funcion f,

$$\mathcal{R}(f) = \Delta(f) \tag{13.10}$$

Yao's Lemma holds for decision trees, as they take inputs of fixed length, and as there is a finite number of labelled trees for inputs of any length. The Lemma is a version of von Neumann's minmax theorem from game theory. Here we consider one player to choose from a number of decision trees, and another player to choose from among a set of inputs, and we let the payoff equal the cost of running a tree on an input. The Lemma states that the cost is the same whether we let one player or the other choose first - whether we consider distributions over inputs or distributions over trees.

So in order to find a lower bound on some randomized algorithm, it suffices to find a lower bound on $\Delta(f)$. Such a lower bound can be found by postulating an input distribution $\mathcal{D}$ and seeing whether every algorithm has expected cost at least equal to the desired lower bound.

EXAMPLE 19 We return to considering the majority function, and we seek to find a lower bound on $\Delta(f)$. Consider a distribution over inputs such that inputs in which all three bits match, namely 000 and 111, occur with probability 0. All other inputs occur with probability 1/6. For any decision tree, that is, for any order in which the three bits are examined, there is exactly a 1/3 probability that the first two bits examined will be the

same value, and thus there is a 1/3 probability that the cost is 2. There is then a 2/3 probability that the cost is 3. Thus the overall expected cost for this distribution is 8/3. This implies that $\Delta(f) \geq 8/3$ and in turn that $\mathcal{R}(f) \geq 8/3$. So $\Delta(f) = \mathcal{R}(f) = 8/3$.

Chapter 14

Communication complexity

SCRIBE: *Tony Wirth*

Although our discussion of communication is brief, it is a necessary one, as the concept will be useful in future lectures on circuit complexity.

We consider a function f that maps $2n$ bits, partitioned into two equal length halves, into a single bit. That is,

$$f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$$

In the two party (or player) model, each party has an n-bit string; say player 1 has x and player 2 has y. Neither party knows anything about the other's bit string. Both want to know $f(x,y)$, and to do this they communicate. (The parties are not adversaries; they help and trust each other.) The complexity of the function f is the number of bits that need to be exchanged for the worst case inputs. Each party has unlimited computational power.

Formally, a t-round communication protocol for f is a sequence of function pairs $(S_1, C_1), (S_2, C_2), \ldots, (S_t, C_t), (f_1, f_2)$. The input of S_i is the communication pattern of the first $i-1$ rounds and the output is from $\{1,2\}$, indicating which player will communicate in the ith round. The input of C_i is the input string of this selected player as well as the communication pattern of the first $i-1$ rounds. The output of C_i is the bit that this player will communicate in the ith round. Finally, f_1, f_2 are 0/1-valued functions that the players apply at the end of the protocol to their inputs as well as the communication pattern in the t rounds in order to compute the output. These two outputs must be $f(x,y)$.

As our example for this lecture, consider the equality function:

$$\mathrm{EQ}(x,y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

We claim the (deterministic) complexity of EQ is in fact n. For contradiction's sake, suppose a protocol exists whose complexity is at most $n-1$. Then there are only 2^{n-1} communication patterns possible between the players. Consider the set of all 2^n pairs (x,x). Using the pigeonhole principle we conclude there exist two pairs (x,x) and (x',x') on which the communication pattern is the same. Of course, thus far we have nothing to object to, since the answers $\mathrm{EQ}(x,x)$ and $\mathrm{EQ}(x',x')$ on both pairs are 1. However, now imagine giving one player x and the other player x' as inputs. A moment's thought shows that the communication pattern will be the same as the one on (x,x) and (x',x'). (Formally, this can be

shown by induction. If player 1 communicates a bit in the first round, then clearly this bit is the same whether his input is x or x'. If player 2 communicates in the 2nd round, then his bit must also be the same on both inputs since he receives the same bit from player 1. And so on.) Hence the player's answer on (x, x) must agree with their answer on (x, x'). But then the protocol must be incorrect, since $\mathrm{EQ}(x, x') = 0 \neq \mathrm{EQ}(x, x)$.

The lowerbound argument above is called a *crossing sequence* argument.

Now we give another way to perform the analysis. Consider the matrix of f, denoted $M(f)$, which is a $2^n \times 2^n$ matrix whose (x, y)'th entry is $f(x, y)$. See Figure 14.1. We

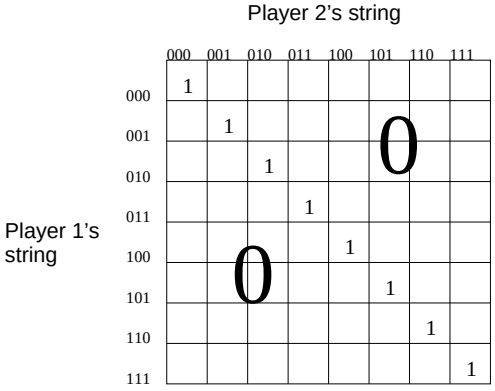


Figure 14.1: Two-way communication matrix, M(f), for the equality function with 3-bit inputs. The numbers inside the matrix are the values of f on the inputs.

visualize the communication protocol in terms of this matrix. After each bit of communication, we divide the matrix into two parts, each of which is a rectangle. The rectangles represent the possible combinations of input strings that lead to a particular communication sequence. (Note that our definition of *rectangle* here is that $A \times B$ is a rectangle in $X \times Y$ whenever $A \subseteq X$ and $B \subseteq Y$.) For example, if the communication protocol has the first player sending one bit and then the second player sending one bit, then the communication matrix might look like Figure 14.2. After k steps, the matrix has been partitioned into 2^k

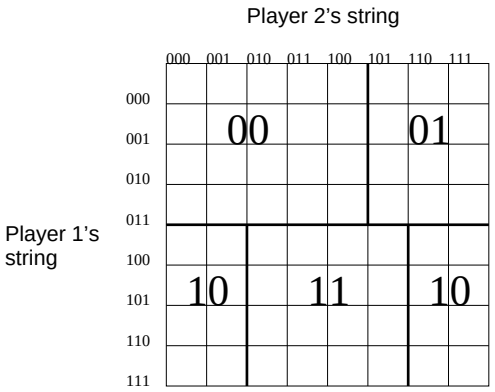


Figure 14.2: Two-way communication matrix after two steps. The large number labels are the concatenation of the bit sent by the first party with the bit sent by the second party.

rectangles. If the protocol stops, then the value of $f(x, y)$ is determined, and must be a constant. Thus the communication (so far) must have led to a *monochromatic* rectangle—that is a rectangle with all ones or all zeros. (Once again, denoting the rectangle by $A \times B$, the rectangle is *monochromatic* if for all x in A and y in B, $f(x, y)$ is the same.) The adversary's strategy is to give inputs to the players so that at each stage of the protocol, they end up in the rectangle with the largest number of pairs of the form (z, z), for some z in $\{0, 1\}^n$. Since the number of 1s in the (equality) matrix is 2^n, if the number of steps is less than n, then under this strategy the rectangle is not monochromatic.

REMARK 4 There was a question in class about the model of communication. Can a player communicate by not saying anything? (After all, they have three options: send a 0, or 1, or not say anything in that round.) If this silence were allowed, then we would essentially regard the communication as having a ternary, not binary, alphabet. But the same lowerbound arguments would apply.

DEFINITION 32 *A monochromatic tiling of $M(f)$ is a sequence of disjoint monochromatic rectangles whose union is $M(f)$. We denote by $\chi(f)$ the minimum number of rectangles in any monochromatic tiling of $M(f)$.*

The following theorem is immediate from our discussion above.

THEOREM 36
If f has communication complexity C then it has a monochromatic tiling with at most 2^C rectangles.

It follows that the communication complexity of f is at least $\lceil \log \chi(f) \rceil$.

Now we introduce a way to lowerbound $\chi(f)$ (and hence communication complexity). Recall the high-school notion of *rank* of a square matrix: it is the size of the largest subset of rows/colums that are independent. The following is another definition.

DEFINITION 33 *The* rank *of an $n \times n$ matrix M is the minimum value of l such that M can be expressed as*

$$M = \sum_{i=1}^{l} \alpha_i B_i,$$

where $\alpha_i \in \mathbf{R}$ and each B_i is an $n \times n$ matrix of rank 1.

The following theorem is trivial, since each monochromatic rectangle can be viewed as a rank 1 matrix (by filling out entries outside the rectangle with 0's).

THEOREM 37
For every function f,

$$\chi(f) \geq \mathit{rank}(M(f)).$$

Thus in this lecture we have covered three main ways of obtaining a lower bound for communication complexity.

(i) Applying a crossing sequence argument.

(ii) Lowerbound $\chi(f)$.

(iii) Lowerbound $\text{rank}(M(f))$.

Method 1 is the strongest method, followed by Method 2 and the weakest is Method 3. For instance, having a minimal tiling provides an adversary strategy for a crossing sequence argument. See Exercise i. Also, we can separate the power of these lowerbound arguments. For instance, we know functions for which there is a significant gap between $\log \chi(f)$ and $\log \text{rank}(M(f))$.

In the late 1970s and early 1980s, communication complexity was used as a model for parallel computation; in particular, for modelling space/time tradeoffs. Yao pointed out that communication complexity could provide lower bounds for the resources used in a VLSI circuit. For instance, in a VLSI chip that is an $m \times m$ grid, if the communication complexity for a function is greater than c, then the time required to compute it is at least c/m.

We will not explicitly study randomized communication, except to note that randomization can significantly reduce the need for communication. For instance with with public random bits, we can use fingerprinting with random primes (explored in Lecture 13), allows us to compute the equality function by exchanging $O(\log n)$ bits: the players just pick a random prime p of $O(\log n)$ bits and exchange $x \pmod p$ and $y \pmod p$.

Exercises

§i Suppose we know that $\text{rank}(M(f))$ is C. Give a crossing sequence argument that proves the communication complexity is at least $\lceil \log C \rceil$.

§ii Consider x, y as vectors over $GF(2)^n$ and let $f(x, y)$ be their inner product mod 2. Prove that the communication complexity is n. (Hint: Use the rank lowerbound.)

§iii For any graph G with n vertices, consider the following communication problem: Player 1 receives a clique C in G, and Player 2 receives an independent set I. They have to communicate in order to determine $|C \cap I|$. (Note that this number is either 0 or 1.) Prove an $O(\log^2 n)$ upperbound on the communication complexity.

Can you improve your upperbound or prove a lower bound better than $\Omega(\log n)$?

§iv Associate the following communication problem with any function $f : \{0,1\}^n \to \{0,1\}$. Player 1 gets any input x such that $f(x) = 0$ and player 2 gets any input y such that $f(y) = 1$. They have to communicate in order to determine a bit position i such that $x_i \neq y_i$.

Show that the communication complexity of this problem is *exactly* the minimum depth of any circuit that computes f.

§v Use the previous question to show that computing the parity of n bits requires depth at least $2 \log n$.

Chapter 15

Circuit complexity

SCRIBE: *Tony Wirth*

Complexity theory's Waterloo

As we saw in an earlier lecture, if **PH** $\neq {\Sigma_2}^P$ then **NP** $\not\subseteq$ **P**/poly [KL 1980]. Since we believe that the polynomial hierarchy doesn't collapse, we believe that **NP** problems don't have polynomial size circuits. Many researchers believe that circuits with their gates and wires should be easier to reason about than Turing machines and have tried hard to prove circuit lowerbounds.

Until recently, the best lower bound for a function in **NP** was about $4n$. At STOC 2001, Lachish and Raz will demonstrate a $4.5n - o(n)$ circuit lower bound. (It is easy to prove that n is a lower bound for any evasive function, since all the input bits need to be processed.)

To make life (comparatively) easier, researchers have focussed on restricted circuit classes, and have been successful in proving some decent lowerbounds. However, after some early successes, this approach also seems stuck now.

Today we cover the first major circuit lowerbound, proven in 1981.

15.1 AC^0 and Håstad Switching Lemma

AC^0 is the class of languages computable by circuit families of constant depth, polynomial size, and whose gates have unbounded fanin. (Restricting ourselves to fanin 2 with constant depth circuits isn't much fun, as the output could only depend on a constant number of inputs.) The burning question in the late 1970s was whether problems like Clique and TSP have AC^0 circuits. In 1981, Furst, Saxe and Sipser showed that the parity function is not in AC^0. (Parity is the function $\bigoplus$, where $\bigoplus(X_1, X_2, \ldots, X_n) = \sum_i X_i \pmod 2$.) Often in computer architecture courses one shows using Karnaugh maps that the parity function requires exponentially many gates if the depth is two. (Basically, a Karnaugh map can be used to simplify a circuit by grouping *adjacent* input pairs that have the same output. Unfortunately the Karnaugh map for the parity function is like a chess board so no such grouping can occur and the number of gates must be linear in the number of inputs on which the function is 1 —which is exponential in the input size.) The Karnaugh map technique

does not seem to give any lowerbounds for even depth 3 circuits, however. Furst, Saxe, and Sipser introduced a new combinatorial technique.

Main idea in FSS proof: Suppose C is any AC^0 circuit of size n^p and depth d. We randomly select $n - n^{1/2^d}$ input variables and assign 0 or 1 to them independently at random. This process is called a *random restriction*. As a result, the outputs of many gates become fixed and so we can delete these gates and propagate their values. The claim — proved below— is that the circuit is simplified to such an extent that it is now computing a function whose decision tree complexity is $O(1)$. This suggests that the function computed by C is fairly simple: fixing $n - n^{1/2^d}$ input bits randomly yields a function dependent only on $O(1)$ bits. However, the parity function is not simple in this sense: its restriction is still the parity (or its negation) function on the remaining $n^{1/2^d}$ variables and its decision tree complexity remains $n^{1/2^d}$. So C could not have been computing the parity function.

Why did decision trees prop up in the proof? They simply provide a convenient way to reason about AND and OR gates. Suppose a function f has a k-DNF representation shown in Figure 15.1. Then the 1-certificate complexity of the function is at most k. Likewise, if

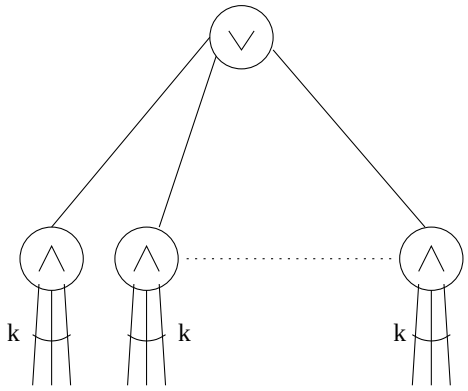


Figure 15.1: The function f has depth two DNF representation in which the OR gate has huge fanin, but the AND gates have fanin at most k.

the function has a k-CNF representation (AND of ORs, where each OR has fanin k), then its 0-certificate complexity is at most k. Now consider a function that is *both* a k-CNF and a k-DNF. Its decision tree complexity is at most k^2, according to the theorem involving the pruning argument in Lecture 17. Conversely, if a function has decision tree complexity of k then it is representable as both a k-CNF and a k-DNF.

Now we prove the main lemma about how a circuit simplifies under a random restriction, namely, when we pick a random set of some t variables and assign 0 or 1 to each independently at random. If ρ denotes this restriction, then $f|_\rho$ denotes the function after this restriction is imposed. We use $D(f)$ to denote the decision tree complexity of a function f.

Lemma 38 (Håstad's switching lemma (1986))
Suppose f is expressible as a k-DNF, and ρ is a restriction *that assigns random values to t*

randomly selected input bits. Then

$$\mathbf{Pr}_\rho[D(f|_\rho) > s] \leq \left(\frac{7(n-t)k}{n}\right)^s.$$

When this lemma is used, the interesting values of the parameters are $n - t \approx n^{1/2}$ with k and s large constants. In this situation, the probability of a *bad random restriction* is the reciprocal of some large power of n—a tiny probability. This tiny probability is good since we will want to apply the inequality at each gate of a circuit whose size is some arbitrary polynomial in n.

We can use Håstad's lemma to prove that parity is not in AC^0. We start with any AC^0 circuit and assume that the circuit has been simplified as follows (the simplifications are straightforward to do and are left as exercises): (a) all fanouts are 1; the circuit is a tree (b) all *not* gates to the input level of the circuit; in other words, there are now $2n$ input wires, and last n of them are the negations of the first n (c) $\vee$ and $\wedge$ gates alternate —at worst this assumption doubles the depth of the circuit. Specifically, even levels have $\wedge$ gates and the odd levels have $\vee$ gates. (d) We think of the bottom level as having AND gates of fanin 1.

We proceed to randomly restrict more and more variables, hoping to show that the circuit simplifies to represent a function of constant decision tree complexity. Each step with high probability reduces the depth of the circuit by 1. Letting n_i stand for the number of unrestricted variables after step i, we restrict $n_i - \sqrt{n_i}$ variables at step $i + 1$. Since $n_0 = n$, we have $n_i = n^{1/2^i}$. Let k_i stand for the fanin at the bottom level of the circuit after i steps, with $k_0 = 1$. Håstad's switching Lemma shows that with high probability the k_i-DNF circuits at the lowest level turn into k_{i+1}-CNF circuits (see Figures 15.2 and 15.3).

Collapsing the new $\wedge$ nodes with their parent we reduce the circuit depth by one. We

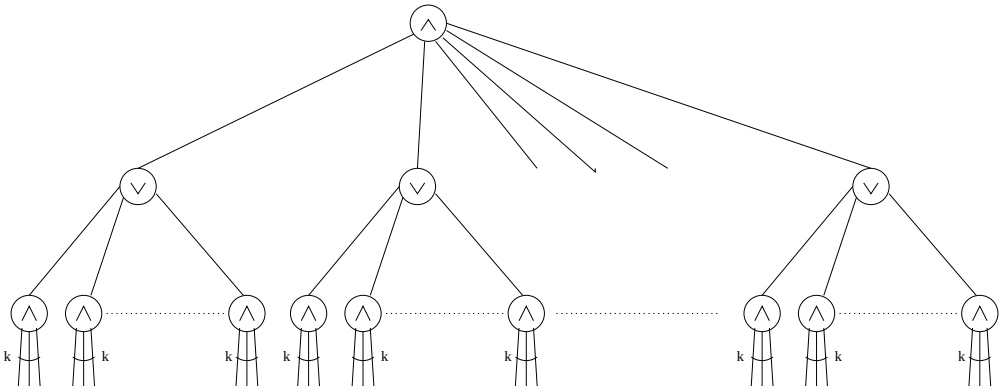


Figure 15.2: Circuit before Håstad switching transformation.

continue until the circuit is of depth two, and then a random restriction gives us, by Håstad's lemma, a function of constant decision tree complexity.

Now we indicate the choices of parameters that show that the probability that any of the steps fails is less than 1/3 (in other words, the probability that all the steps succeed is more than 2/3). Every gate in the circuit, except the $\vee$ gates at the bottom level, is

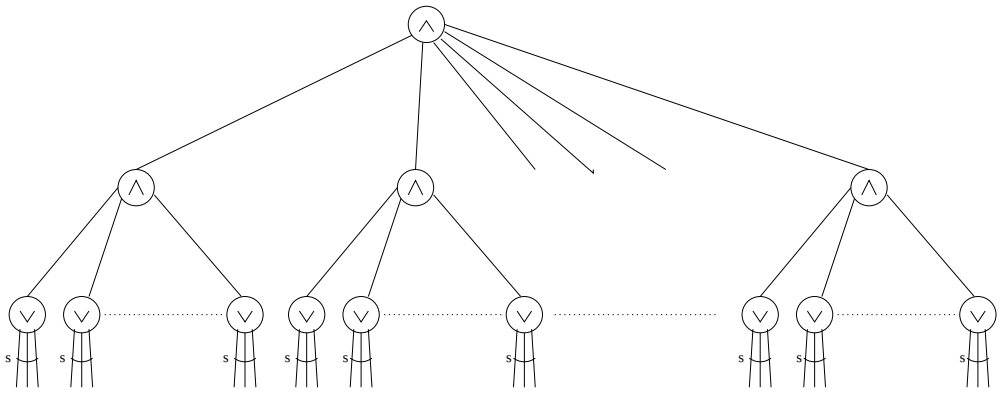


Figure 15.3: Circuit after Håstad switching transformation. Notice that the new layer of $\wedge$ gates can be collapsed with the single $\wedge$ parent gate, to reduce the number of levels by one.

transformed so we would like the probability of *failure* at each switch to be at most $1/3S$, where S is the size of the circuit. In Step i, the failure probability is

$$\left(\frac{7k_i}{n^{1/2^i}}\right)^{k_{i+1}}. \tag{15.1}$$

Since the circuit is of polynomial size, we can assume that S is dominated by some n^b. In order to make expression (15.1) sufficiently small, we set $k_i = 2^i b + 1$. If i is constant, then so is k_i, and so for sufficiently large n, the failure probability will be less than $1/3S$.

Thus we have proved the following.

THEOREM 39
If function f is computed by a depth d circuit of size n^b, then a random restriction of $n - n^{1/2^d}$ input variables with probability at least $2/3$ leaves a function of decision tree complexity at most $2^d b + 1$.

Clearly, the parity function does not satisfy the conclusion of thie Theorem for any constants d, b, so it is not in AC^0.

15.2 Proof of Håstad Switching Lemma

Now we prove the Switching Lemma. The original proof was more complicated; this one is due to Razborov

Let R_t denote the set of all restrictions to t variables, where $t \geq n/2$. Then

$$|R_t| = \binom{n}{t} 2^t. \tag{15.2}$$

The set of *bad restrictions* —those ρ for which $D(f|_\rho) > s$ is greater than s—is a subset of these. To show that this subset is small, we give a one-to-one mapping from it to the cartesian product of three sets: R_{t+s}, the set of restrictions to $(t + s)$ variables, a set $\mathrm{code}(k, s)$ of size $k^{O(s)}$, and the set $\{0, 1\}^s$. (The set $\mathrm{code}(k, s)$ is explained below.) This

cartesian product has size $\binom{n}{t+s}2^{t+s}k^{O(s)}2^s$. Thus the probability of picking a bad restriction is bounded by

$$\frac{\binom{n}{t+s}2^{t+s}k^{O(s)}2^s}{\binom{n}{t}2^t}. \tag{15.3}$$

Intuitively, this ratio is small because k, s are to be thought of as constant and $t > n/2$, so

$$\binom{n}{t}2^t \gg \binom{n}{t+s}2^{t+s}.$$

Formally, by using the correct constants as well as the approximation $\binom{n}{a} \approx (ne/a)^a$, we can upperbound the ratio in (15.3) by

$$\left(\frac{7(n-t)k}{n}\right)^s.$$

Thus to prove the Lemma it suffices to describe the one-to-one mapping mentioned above. This uses the notion of a *canonical decision tree* for f. We take the k-DNF circuit for f, and order its terms (i.e., the $\wedge$ gates in Figure 15.1) arbitrarily and within each term we order the variables. The canonical decision tree queries all the variables in the first term in order, then all the variables in the second term, and so on until the function value is determined.

Suppose that restriction ρ is bad, that is, $D(f|_\rho) > s$. The canonical decision tree for $f|_\rho$ is defined in the same way as for f, using the same order for terms and variables. (Note that this canonical decision tree for $f|_\rho$ can skip over any terms whose value has been fixed by ρ.) Since the decision tree complexity of $f|_\rho$ is at least s, there is a path of length at least s from the root to a leaf. This path defines a partial assignment to the input variables; denote it by π. The rough intuition is that the one-to-one mapping takes ρ to itself plus π.

Let us reason about restriction ρ. None of the $\wedge$ gates outputs 1 under ρ, otherwise $f|_\rho$ would be determined and would not require a decision tree. Some terms output 0, but not all, since that would also fix the overall output. Imagine walking down the path π. Let t_1 be the first term that is not set to zero under ρ. Then π must query all the (unfixed) variables in t_1. Denote the part of path π that deals with t_1 by π_1; that is, π_1 is an assignment to the variables of t_1. Since f is not determined at this point, we conclude that π_1 sets t_1 to zero. Let t_2 be the next term not yet set to zero by ρ and π_1; again, the path must set t_2 to zero. Let π_2 denote the assignment to the variables of t_2 along the path. Then π_2 also sets t_2 to zero. This process continues until we have dealt with m terms (or are dealing with the mth) when π has reached depth s. Each of these terms was not set by ρ and, except perhaps for π_m, is set to zero after s queries in π. The (disjoint) union of these π_i terms contains assignments for at least s variables that were unfixed in ρ.

Our mapping will map ρ to

$$([\rho \cup \sigma_1 \cup \sigma_2 \cup \ldots \cup \sigma_m], c, z)$$

where the term σ_i is the (unique) set of assignments that makes t_i true, $c = c_1c_2 \ldots c_m$ is in code(k, s) (this set is defined below) and $z \in \{0,1\}^s$. In defining this mapping we are crucially relying on the fact that there is only one way to make a term true, namely to set all its literals to 1.

To show that the mapping is one-to-one, we show how to invert it uniquely. This is harder than it looks since *a priori* there is no way to identify ρ from $\rho \cup \sigma_1 \cup \sigma_2 \cup \ldots \cup \sigma_m$. The main idea is that the information in c and z allows us to extract ρ from the union.

Suppose we are given the assignment $\rho \cup \sigma_1 \cup \sigma_2 \cup \ldots \cup \sigma_m$. We can plug this assignment into f and then infer which term serves as t_1: it is the first one to be set true. The first k bits of c, say c_1, are an indicator string showing which variables in t_1 are set by σ_1 (and indeed π_1). We can reconstruct π_1 from σ_1 using the string z, which indicates which of the s bits fixed in the decision tree differ between the π assignments and the σ assignments.

Having reconstructed π_1, we can work out which term is t_2: it is the first *true* term under the restriction $\rho \cup \pi_1 \cup \sigma_2 \cup \ldots \cup \sigma_m$. The next k bits of c, denoted c_2, give us σ_2 whence we obtain —using some help from the next few bits of z— the assignment π_2. We continue this process until we have processed all m terms and figured out what $\sigma_1, \ldots, \sigma_m$ are. Thus we have figured out ρ, so the mapping is one-to-one.

Finally, we define the set $\mathrm{code}(k, s)$: this is the set of all sequences of k-bit binary strings in which each string has at least one 1 bit and the total number of 1 bits is at most s. It can be shown by induction on s that

$$|\mathrm{code}(k, s)| \leq \left(\frac{k}{\ln 2}\right)^s. \tag{15.4}$$

Exercises

§i Show that given a circuit of size S we can easily construct an equivalent circuit with the same depth and size $2S$ which does not contain NOT gates but instead has n additional inputs that are negations of the original n inputs. (Hint: each gate in the old circuit gets a twin that computes its negation.)

§ii Show that if a function is in AC^0 then in fact it has AC^0 circuits with fanout 1.

§iii Prove assertion (15.4) about $\mathrm{code}(k, s)$.

Chapter 16

Circuit Lower Bounds Using Multiparty Communication Complexity

SCRIBE:*Iannis Tourlakis*

We begin by defining some new circuit complexity classes which lie on the "frontier" of classes for which we can prove separations. Results known for such classes will be given. Next we will define multi-party communication and give its connection to circuit complexity.

16.1 Circuits With "Counters"

DEFINITION 34 *A language L is in $\mathbf{ACC}^0$ if there exists a circuit family $\{C_n\}$ with constant depth and polynomial size (and unbounded fan-in) consisting of $\wedge$, $\vee$, $\neg$ and $\mathrm{mod}\, m_1, \ldots, \mathrm{mod}\, m_k$ gates accepting L.*

Note that the number k of different moduli in the above definition is constant. It will be convenient to have a notation that indicates precisely the moduli allowed in an $\mathbf{ACC}^0$ family. To that end, let $\mathbf{ACC}^0(q)$ indicate the class of languages accepted by $\mathbf{ACC}^0$ circuits where all the mod gates have modulo q.

We do have some lower bounds for $\mathbf{ACC}^0(q)$ circuits when q is a prime:

THEOREM 40 (RAZBOROV,SMOLENSKY)
For distinct primes p and q, the function $\mathrm{mod}\, p$ is not in $\mathbf{ACC}^0(q)$.

However, when q is not a prime (a prime power, to be exact) we reach the limits of our knowledge. Indeed it is consistent with current knowledge that the majority of n bits —to give one example— can be computed by linear size circuits of constant depth consisting entirely of $\mathrm{mod}\, 6$ gates. The "high-level" reason for why the proof of Theorem 40 does not work in the $\mathrm{mod}\, 6$ case is that the proof relies on polynomial representations of $\mathbf{ACC}^0$ circuits: in this representation, polynomials $\mathrm{mod}\, 6$ seem to have surprising power.

Frontier 1: Is Clique in $\mathbf{ACC}^0(6)$?

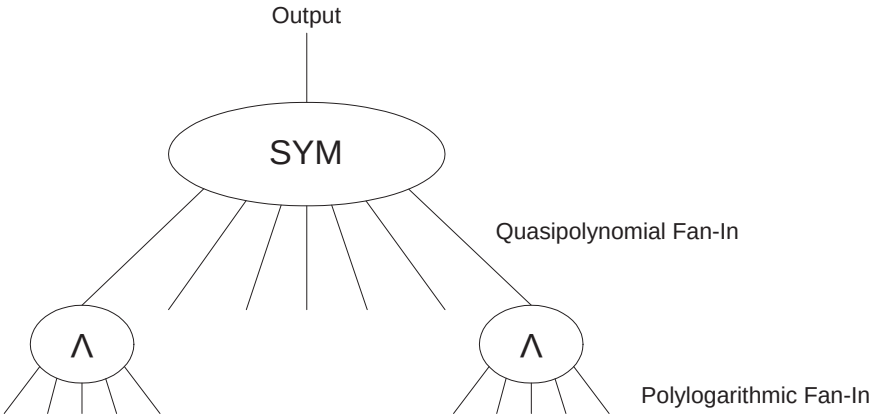


Figure 16.1: The depth 2 circuit with a symmetric output gate from Theorem 41.

It is worth noting that Theorem 40 concerns *non-uniform* circuits. Moreover, in the (weaker) uniform case some fairly strong lower bounds are known: Allender and Gore have shown that a decision version of the Permanent (and hence the Permanent itself) requires exponential size "Dlogtime-uniform" $\mathbf{ACC}^0$ circuits. (A circuit family $\{C_n\}$ is *Dlogtime uniform* if there exists a deterministic Turing machine M that given a triple (n, g, h) determines in linear time —i.e., $O(\log n)$ time when $g, h \leq \text{poly}(n)$— what types of gates g and h are and whether g is h's parent in C_n.)

Let us return again to the nonuniform $\mathbf{ACC}^0$. Let a *symmetric* gate be a gate whose output depends only on the number of inputs that are 1. For example, majority and mod gates are symmetric. Yao has shown that $\mathbf{ACC}^0$ circuits can be simplified to give an equivalenet depth 2 circuits with a symmetric gate at the output (figure 16.1). Beigel and Tarui subsequently improved Yao's result:

Theorem 41 (Yao, Beigel and Tarui)
If $f \in \mathbf{ACC}^0$, then f can be computed by a depth 2 circuit C with a symmetric gate with quasipolynomial (i.e., $2^{\log^k n}$) fan-in at the output level and $\vee$ gates with polylogarithmic fan-in at the input level.

We will revisit this theorem below in section 16.4.

16.2 Linear Circuits With Logarithmic Depth

In this section concentrate on circuits with *bounded* fan-in in contrast to the previous section. To that end, consider the question of finding an explicit function $f : \{0,1\}^n \to \{0,1\}$ that cannot be computed by bounded fan-in circuits of linear size and logarithmic depth. (Note that by counting one can easily show that *some* function on n bits requires superpolynomial size circuits and hence bounded fan-in circuits with more than logarithmic depth).

Frontier 2: Find an explicit function that cannot be computed by circuits of linear size and logarithmic depth.

Valiant thought about this problem in the '70s. His initial candidates for lowerbounds boiled down to showing that a certain graph called a *superconcentrator* needed to have superlinear size. He failed to prove thisand instead ended up proving that such superconcentrators do exist!

Another sideproduct of Valiant's investigations was the following important lemma concerning depth-reduction for such circuits.

LEMMA 42 (VALIANT)
In any circuit with m edges and depth d, there are $km/\log d$ edges whose removal leaves a circuit with depth at most $d/2^{k-1}$.

This lemma can be applied as follows. Suppose we have a circuit C of depth $c \log n$ with n inputs $\{x_1, \ldots, x_n\}$ and n outputs $\{y_1, \ldots, y_n\}$, and suppose $2^k \sim c/\epsilon$ where $\epsilon > 0$ is arbitrarily small. Removing $O(n/\log\log n)$ edges from C then results in a circuit with depth at most $\epsilon \log n$. But then, since C has *bounded* fan-in, we must have that each output y_i is connected to at most $2^{\epsilon \log n} = n^\epsilon$ inputs. So each output y_i in C is completely determined by n^ϵ inputs and the values of the omitted edges. So we have a "dense" encoding for the function $f_i(x_1, \ldots, x_n) = y_i$. We do not expect this to be the case for any reasonably difficult function.

16.3 Multiparty Communication Protocols

Multiparty communication games can be used to model some circuit classes. The model of multiparty communication that we use is the "number on the forehead" model (found for example in math puzzles that involve people in a room, each person having a bit on their head). More concretely, when trying to compute $f(x_1, \ldots, x_k)$, $x_i \in \{0,1\}^n$ for some function f, the communicating parties have as input the x_i's where the ith party can see all the x_j such that $j \neq i$. The parties have a protocol for communicating where all communication is posted on a "public blackboard". At the end of the protocol all parties must know $f(x_1, \ldots, x_k)$.

In the 2-party model we introduced the notion of a monochromatic rectangles in order to prove lower bounds. For proving lower bounds in the multiparty model we have the notion of cylinder intersections. A *cylinder in dimension i* is a subset S of the inputs such that if $(x_1, \ldots, x_k) \in S$ then for all x_i' we have that $(x_1, \ldots, x_{i-1}, x_i', x_{i+1}, \ldots, x_k) \in S$ also. A *cylinder intersection* is simply an intersection of cylinders. So, similarly to the 2-party case, if f has a multiparty protocol that communicates c bits, then its matrix has a tiling using at most 2^c monochromatic cylinder intersections.

EXAMPLE 20 Consider computing the function $f(x_1, x_2, x_3) = \bigoplus_{i=1}^{n} \mathrm{maj}(x_{1i}, x_{2i}, x_{3i})$ in the 3-party model where the x_i are n bits number. The communication complexity of this function is 3: each player counts the number of i's such that she can determine the majority of x_{1i}, x_{2i}, x_{3i} by examining the bits available to her. She writes the parity of this number on the blackboard, and the final answer is the parity of the players' bits. This protocol is correct because the majority for each row is known by either 1 or 3 players.

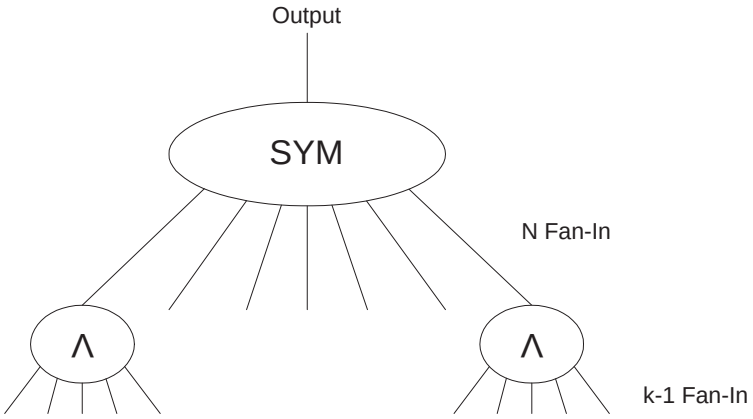


Figure 16.2: If f is computed by the above circuit, then f has a k-party protocol of complexity $k \log N$.

EXAMPLE 21 Consider computing the generalized inner product function $f(x_1, \ldots, x_k) = \bigoplus_{i=1}^{n} \bigwedge(x_{1i}, x_{2i}, x_{3i})$. There is a multiparty protocol with communication complexity $n/2^k$. Babai, Nisan and Szegedy (1989) have shown a lower bound of $n/4^k$ bits of communication for this problem.

16.4 Tying Everything Together

In this section we show how to reduce the problem of computing lower bounds for $\mathbf{ACC}^0$ circuits and for linear size logarithmic depth circuits to communication complexity questions using the machinery developed in the previous sections.

16.4.1 $\mathbf{ACC}^0$ Circuits

Suppose $f(x_1, \ldots, x_k)$ has a depth-2 circuit with a symmetric gate with fan-in N at the output and $\wedge$ gates with fan-in $k-1$ at the input level (figure 2). The claim is that f's k-party communication complexity is at most $k \log N$. To see the claim, first partition the $\wedge$ gates amongst the players. Each bit is not known to exactly one player, so the input bits of each $\wedge$ gate are known to at least one player; assign the gate to such a player with the lowest index. Players then broadcast how many of their gates output 1. Since this number has at most $\log N$ bits, the claim follows.

Our hope is to employ this connection with communication complexity in conjunction with Theorem 41 to obtain lower bounds on $\mathbf{ACC}^0$ circuits. For example, note that the function in Example 21 above cannot have $k < \log n/4$. However, this is not enough to obtain a lower bound on $\mathbf{ACC}^0$ circuits since we need to show that k is not polylogarithmic to employ Theorem 41. Thus a strengthening of the Babai Nisan Szegedy lowerbound to $\Omega(n/\text{poly}(k))$ for say the CLIQUE function would close Frontier 1.

16.4.2 Linear Size Logarithmic Depth Circuits

Suppose that $f : \{0,1\}^n \times \{0,1\}^{\log n} \to \{0,1\}^n$ has bounded fan-in circuits of linear size and logarithmic depth. If $f(x,j,i)$ denotes the ith bit of $f(x,j)$, then Valiant's Lemma implies that $f(x,j,i)$ has a simultaneous 3-party protocol—that is, a protocol where all parties speak only once and write simultaneously on the blackboard (i.e., non-adaptively)—where,

- (x,j) player sends $n/\log\log n$ bits;
- (x,i) player sends n^ϵ bits; and
- (i,j) player sends $O(\log n)$ bits.

So, if we can show that a function does not have such a protocol, then we would have a lower bound for the function on linear size logarithmic depth circuits with bounded fan-in.

Conjecture: The function $f(x,j,i) = x_{j \oplus i}$, where $j \oplus i$ is the bitwise xor, is conjectured to be hard, i.e., f should not have a compact representation.

Chapter 17

Algebraic Computation Models

SCRIBE: *Loukas Georgiadis*

We think of numerical algorithms –root-finding, gaussian elimination etc. —as operating over **R** or **C**, even though the underlying representation of the real or complex numbers involves finite precision. Therefore, it is natural to ask about the complexity of computations over the real numbers or even computations over arbitrary fields. Such an idealized model may not be implementable, but it provides a useful approximation to the asymptotic behavior as computers are allowed to use more and more precision in their computations. However, coming up with a meaningful, well-behaved model is not an easy task, as the following example suggests.

EXAMPLE 22 (PITFALLS AWAITING DESIGNERS OF SUCH MODELS) A real number can encode infinite amount of information. For example, a single real number is enough to encode the answer to every instance of SAT (of any language, in general). Thus, a model that can store any real number with infinite precision may not be realistic. Shamir has shown how to factor any integer n in poly($\log n$) time on a computer that can do real arithmetic with arbitrary precision.

The usual way to avoid this pitfall is to restrict the algorithms' ability to access individual bits (e.g., the machine may require more than polynomial time to extract a particular digit from a real number).

In this lecture we define two algebraic computation models: algebraic trees and algebraic Turing Machines. The latter is closely related to the standard Turing Machine model and allows us to define similar complexity classes. Then, we refer to the relation of algebraic and boolean classes. We conclude with a discussion on decidability for the algebraic Turing Machine.

17.1 Algebraic Computation Trees

Recall the comparison based sorting algorithms; they only allow questions of the type $x_i > x_j$, which is the same as asking whether $x_i - x_j > 0$. The left hand side term of this last inequality is a linear function. We examine the effect of allowing i) the use of any

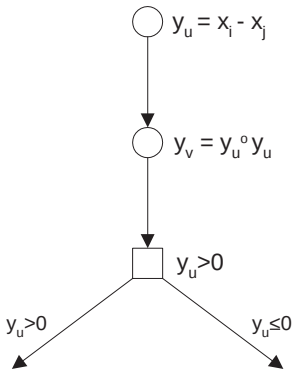


Figure 17.1: An Algebraic Computation Tree

polynomial function and ii) the introduction of new variables together with the ability to ask questions about them. These lead to the following definition:

DEFINITION 35 *An Algebraic Computation Tree is a binary tree where each of the nodes has one of the following types:*

- *Leaf labelled "Accept" or "Reject".*
- *Computation node v labelled with y_v, where $y_v = y_u \circ y_w$ and u, w are either inputs or the labels of ancestor nodes and the operator $\circ$ is in $\{+, -, \times, \div, \sqrt{}\}$ (Note: this is the standard definition for algebraic trees over fields. We don't allow division in the case of rings).*
- *Branch node with out-degree 2. The branch that is taken depends on the evaluation of some condition, such as $y_u = 0$ or $y_u \geq (\leq) 0$, and y_u is some designated ancestor.*

The path from the root to an accepting or rejecting leaf depends on the input $x_1, \ldots, x_n \in R$. The complexity of the computations is measured with respect to the following cost model:

- $+, -$ *are free.*
- $\times, \div, \sqrt{}$ *are charged unit cost.*

A fragment of an algebraic decision tree is shown in figure 17.1. The following examples illustrate some of the languages (over real numbers) whose complexity we want to study.

EXAMPLE 23 [Element Distinctness] Given n numbers $x_1, x_2, \ldots, x_n$ we ask whether they are all distinct. This is equivalent to the question whether $\prod_{i \neq j}(x_i - x_j) \neq 0$.

EXAMPLE 24 [Real number version of subset sum] Given a set of n real numbers $X = \{x_1, x_2, \ldots, x_n\}$ we ask whether there is a subset $S \subseteq X$ such that $\sum_{i \in S} x_i = 1$.

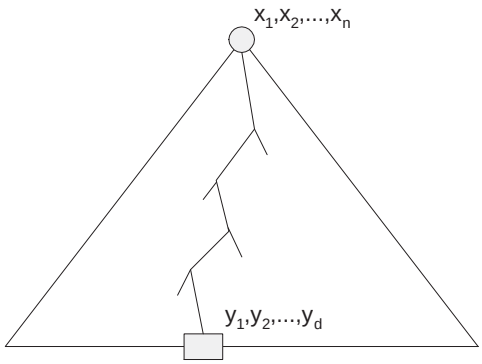


Figure 17.2: A computation path p of length d defines a set of constraints over the n input variables x_i and d additional variables y_j, which correspond to the nodes on p.

REMARK 5 If the depth of a path is d and $y_1, \ldots, y_d$ are variables denoting values of nodes along this path, then the set of inputs $(x_1, \ldots, x_n)$ that follow this path satisfy a set $\mathcal{C}$ of constraints of the type

$$p_i(y_1, \ldots, y_d, x_1, \ldots, x_n) \bowtie 0,$$

where p_i is a degree 2 polynomial (this is an effect of introducing a new variable at each node) and $\bowtie$ is in $\{\leq, \geq, =, \neq\}$. For example, $y_v = y_u \div y_w \leftrightarrow y_v y_w - y_u = 0$. The set S of points that end up in the same path are the solution to an algebraic system (see figure 17.2).

By applying the following trick we can replace the "$\neq$" constraints from $\mathcal{C}$. Then, S is defined as a semi-algebraic set and we can apply some known results to bound its number of connected components (see Definition 37).

REMARK 6 (RABINOVITCH'S TRICK) We can change the "$\neq$" to "$=$" in some constraint

$$p_i(y_1, \ldots, y_m) \neq 0,$$

by introducing a new variable z and asking whether there is a value of z such that

$$q_i(y_1, \ldots, y_m, z) \equiv 1 - z p_i(y_1, \ldots, y_m) = 0$$

(this transformation holds for all fields). Note that we have increased the degree by 1. Alternatively, one can ask whether

$$p_i^2(y_1, \ldots, y_m) > 0,$$

which doubles the degree and does not hold for all fields (e.g., the complex numbers).

Note that after this conversion the maximum degree of the constraints in $\mathcal{C}$ remains 2, because the trick is used only for the branch $y_u \neq 0$ which is converted to $1 - z_v y_u = 0$. (We find Rabinovitch's trick useful also in section 17.2.2 where we prove a completeness result for Hilbert's Nullstellensatz.)

DEFINITION 36 *A set $S \subseteq \mathbf{R}^n$ is* connected *if for all $x, y \in S$ there is path p that connects x and y and lies entirely in S.*

DEFINITION 37 *For $S \subseteq \mathbf{R}^n$ we define $\#(S)$ to be the number of connected components of S.*

The main idea for proving lowerbounds for the Algebraic Computation Tree model is to show that $\#(S)$ is large. To that end, we will find the following result useful.

THEOREM 43 (SIMPLE CONSEQUENCE OF MILNOR-THOM)
If $S \subseteq \mathbf{R}^n$ is defined by degree d constraints with m equalities and h inequalities then

$$\#(S) \leq d(2d-1)^{n+h-1}$$

REMARK 7 Note that the above upperbound is independent of m.

DEFINITION 38 *Let $W \subseteq \mathbf{R}^n$. We define the algebraic decision tree complexity of W as*

$$C(W) = \min_{\substack{\textit{computation} \\ \textit{trees } C \textit{ for } W}} \{\textit{depth of } C\}$$

THEOREM 44

$$C(W) = \Omega\Big(\log\left(\max\left\{\#(W), \#(\mathbf{R}^n - W)\right\}\right) - n\Big)$$

PROOF: Suppose that the depth of a computation tree for W is d, so that there are at most 2^d leaves. We will use the fact that if $S \subseteq \mathbf{R}^n$ and $S|_k$ is the set of points in S with their $n-k$ coordinates removed (projection on the first k coordinates) then $\#(S|_k) \leq \#(S)$ (figure 17.3).

For every leaf there is a set of degree 2 constraints. So, consider a leaf ℓ and the corresponding constraints $\mathcal{C}_\ell$, which are in variables $x_1, \ldots, x_n, y_1, \ldots, y_d$. Let $W_\ell \subseteq \mathbf{R}^n$ be the subset of inputs that reach ℓ and $S_\ell \subseteq \mathbf{R}^{n+d}$ the set of points that satisfy the constraints $\mathcal{C}_\ell$. Note that $W_\ell = \mathcal{C}_\ell|_n$ i.e., W_ℓ is the projection of $\mathcal{C}_\ell$ onto the first n coordinates. So, the number of connected components in W_ℓ is upperbounded by $\#(\mathcal{C}_\ell)$. By Theorem 43 $\#(\mathcal{C}_\ell) \leq 2 \cdot 3^{n+d-1} \leq 3^{n+d}$. Therefore the total number of connected components is at most $2^d 3^{n+d}$, so $d \geq \log(\#(W)) - O(n)$. By repeating the same argument for $\mathbf{R}^n - W$ we have that $d \geq \log(\#(\mathbf{R}^n - W)) - O(n)$. □

Now we can apply the previous result to get a $\Omega(n \log n)$ lowerbound for ELEMENT DISTINCTNESS.

THEOREM 45 (LOWERBOUND FOR ELEMENT DISTINCTNESS)
Let $W = \{(x_1, \ldots, x_n) | \prod_{i \neq j} (x_i - x_j) \neq 0\}$. Then,

$$\#(W) \geq n!$$

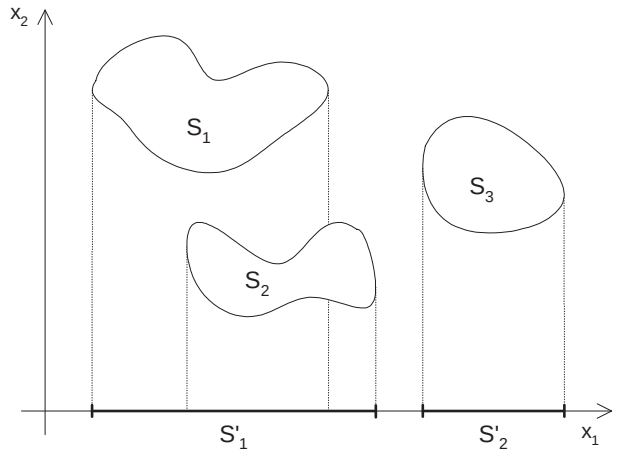


Figure 17.3: Projection can merge but not add connected components

PROOF: For each permutation σ let

$$W_\sigma = \{(x_1, \ldots, x_n) \mid x_{\sigma(1)} < x_{\sigma(2)} < \ldots < x_{\sigma(n)}\}.$$

That is, let W_σ be the set of n-tuples $(x_1, \ldots, x_n)$ to which σ gives order. We claim that for all $\sigma' \neq \sigma$ the sets W_σ and $W_{\sigma'}$ are not connected. The theorem follows immediately from this claim.

Now consider two distinct permutations σ and σ'. There exist two distinct i, j with $1 \leq i, j \leq n$, such that $\sigma^{-1}(i) < \sigma^{-1}(j)$ but $\sigma^{-1}(i) > \sigma^{-1}(j)$. Thus, in W_σ we have $X_j - X_i > 0$ while in $W_{\sigma'}$ we have $X_i - X_j > 0$. But as we move from the first condition to the second, at some point $X_j - X_i$ must become 0 (by the intermediate value theorem of calculus). Definition 36 then implies that W_σ and $W_{\sigma'}$ cannot be connected. □

17.2 The Blum-Shub-Smale Model

Here we consider a Turing Machine that computes over some arbitrary field K (e.g. $K = \mathbf{R}, \mathbf{C}, \mathbf{Z}_2$). This is a generalization of the standard Turing Machine model which operates over the ring $\mathbf{Z}_2$. Each cell can hold an element of K with a finite number of cells initially not blank. In the standard model the computation and branch operations can be executed in the same step. Here we perform these operations separately. So we divide the set of states into the following three categories:

- Shift state: move the head to the left or to the right of the current position.
- Branch state: if the content of the current cell is a then goto state q_1 else goto state q_2.
- Computation state: replace the contents of the current cell with a new value. The machine has a hardwired function f and the new contents of the cell become $a \leftarrow f(a)$.

In the standard model for rings, f is a polynomial over K, while for fields f is a rational function p/q where p, q are polynomials in $K[x]$ and $q \neq 0$. In either case, f can be represented using a constant number of elements of K.

- The machine has a "register" onto which it can copy the contents of the cell currently under the head. This register's contents can be used in the computation.

In the next section we define some complexity classes related to the BSS model. As usual, the time and space complexity of these Turing Machines is defined with respect to the input size, which is the number of cells occupied by the input.

REMARK 8 The following examples show that some modifications of the BSS model can increase significantly the power of an algebraic Turing Machine.

- If we allow the branch states to check if $a > 0$, for real a then, the model becomes unrealistic because it can decide problems that are undecidable on the normal Turing machine. In particular, such a machine can compute $\mathbf{P}/poly$ in polynomial time. (Recall that we showed that $\mathbf{P}/poly$ contains undecidable languages.) If a language is in $\mathbf{P}/poly$ we can represent its circuit family by a single real number hardwired into the Turing machine (specifically, as the coefficient of of some polynomial $p(x)$ belonging to a state). The individual bits of this coefficient can be accessed by dividing by 2, so the machine can extract the polynomial length encoding of each circuit. Without this ability we can prove that the individual bits cannot be accessed.
- If we allow rounding (computation of $\lfloor x \rfloor$) then it is possible to factor in polynomial time, a result due to Shamir.

Even without these modifications, the BSS model is in some sense more powerful than real-world computers: Consider the execution of the operation $x \leftarrow x^2$ for n times. Since we allow each cell to store a real number, the Turing machine can compute and store in one cell (without overflow) the number x^{2^n} in n steps.

17.2.1 Complexity Classes over the Complex Numbers

Now we define the corresponding to **P** and **NP** complexity classes over **C**:

DEFINITION 39 ($\mathbf{P_C}$,$\mathbf{NP_C}$) *$\mathbf{P_C}$ is the set of languages that can be decided by a Turing Machine over **C** in polynomial time. $\mathbf{NP_C}$ is the set of languages L for which there exists a language L_0 in $\mathbf{P_C}$, such that an input x is in L iff there exists a string $(y_1, \ldots, y_{n^c})$ in $\mathbf{C}^{n^c}$ such that (x, y) is in L_0.*

The following definition is a restriction on the inputs of a TM over **C**. These classes are useful because they help us understand the relation between algebraic and binary complexity classes.

DEFINITION 40 (0-1-$\mathbf{NP_C}$)

$$\text{0-1-}\mathbf{NP_C} = \{L \cap \{0,1\}^* \mid L \in \mathbf{NP_C}\}$$

Note that the input for an $\mathbf{NP_C}$ machine is binary but the nondeterministic "guess" may consist of complex numbers. Trivially, 3SAT is in 0-1-$\mathbf{NP}_C$: even though the "guess" consists of a string of complex numbers, the machine first checks if they are all 0 or 1 using equality checks. Having verified that the guess represents a boolean assignment to the variables, the machine continues as a normal Turing Machine to verify that the assignment satisfies the formula.

It is known that 0-1-$\mathbf{NP_C} \subseteq \mathbf{PSPACE}$. In 1997 Koiran proved that if one assumes the Riemann hypothesis, then 0-1-$\mathbf{NP_C} \subseteq \mathbf{AM}[2]$. Recall that $\mathbf{AM}[2]$ is $\mathbf{BP} \cdot \mathbf{NP}$ so Koiran's result suggests that 0-1-$\mathbf{NP_C}$ may not be much bigger than $\mathbf{NP}$.

17.2.2 Hilbert's Nullstellensatz

The language $\mathbf{HN_C}$ is defined as the decision version of Hilbert's Nullstellensatz over $\mathbf{C}$. The input consists of m polynomials p_i of degree d over $x_1, \ldots, x_n$. The output is "yes" iff the polynomials have a common root $a_1, \ldots, a_n$. Note that this problem is general enough to include SAT. We illustrate that by the following example:

$$x \vee y \vee z \leftrightarrow (1-x)(1-y)(1-z) = 0.$$

Next we use this fact to prove that the language 0-1-$\mathbf{HN_C}$ (where the polynomials have 0-1 coefficients) is complete for 0-1-$\mathbf{NP_C}$.

THEOREM 46 (BSS)
0-1-$\mathbf{HN_C}$ *is complete for* 0-1-$\mathbf{NP_C}$.

PROOF: (Sketch) It is straightforward to verify that 0-1-$\mathbf{HN_C}$ is in 0-1-$\mathbf{NP_C}$. To prove the hardness part we imitate the proof of the Cook-Levin theorem; we create a computation tableau and show that the verification is in 0-1-$\mathbf{HN_C}$.

To that end, consider the usual computation tableau of a Turing Machine over $\mathbf{C}$ and as in the case of the standard Turing Machines express the fact that the tableau is valid by verifying all the 2×3 windows, i.e., it is sufficient to perform local checks (Figure 17.4). Reasoning as in the case of algebraic computation trees, we can express these local checks with polynomial constraints of bounded degree. The computation states $c \leftarrow q(a,b)/r(a,b)$ are easily handled by setting $p(c) \equiv q(a,b) - cr(a,b)$. For the branch states $p(a,b) \neq 0$ we can use Rabinovitch's trick to convert them to equality checks $q(a,b,z) = 0$. Thus the degree of our constraints depends upon the degree of the polynomials hardwired into the machine. Also, the polynomial constraints use real coefficients (involving real numbers hardwired into the machine). Converting these polynomial constraints to use only 0 and 1 as coefficients requires work. The idea is to show that the real numbers hardwired into the machine have no effect since the input is a binary string. We omit this mathematical argument here. □

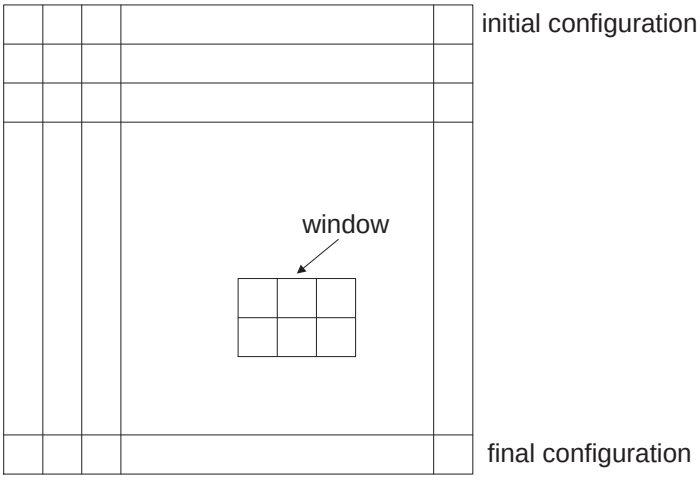


Figure 17.4: Tableau of Turing Machine configurations

17.3 Decidability Questions

Since the Blum-Shub-Smale model is more powerful than the ordinary Turing Machine, it is plausible to reconsider decidability questions. In this section we show that some problems do indeed remain undecidable. In particular we study the decidability of the Mandelbrot set with respect to Turing Machines over **C**. Roger Penrose had raised this question in his meditation regarding artificial intelligence.

17.3.1 The Mandelbrot Set

DEFINITION 41 (MANDELBROT SET DECISION PROBLEM) *Let $P_C(Z) = Z^2 + C$. Then, the Mandelbrot set is defined as*

$$\mathcal{M} = \{C \mid \textit{the sequence } P_C(0), P_C(P_C(0)), P_C(P_C(P_C(0))) \ldots \textit{ is bounded } \}.$$

Note that the complement of $\mathcal{M}$ is recognizable if we allow inequality constraints. This is because the sequence is unbounded iff some number $P_C^k(0)$ has complex magnitude greater than 2 for some k (exercise!) and this can be detected in finite time. However, detecting that $P_C^k(0)$ is bounded for every k seems harder. Indeed, we have:

THEOREM 47
$\mathcal{M}$ is undecidable by a machine over **C**.

PROOF: (Sketch) The proof uses the topology of the Mandelbrot set. Let $\mathcal{M}$ be any TM over the complex numbers that supposedly decides this set. Consider T steps of the computation of this TM. Reasoning as in Theorem 46 and in our theorems about algebraic computation trees, we conclude that the sets of inputs accepted in T steps is is a finite union of semialgebraic sets (i.e., sets defined using solutions to a system of polynomial equations). Hence the language accepted by $\mathcal{M}$ is a countable union of semi-algebraic sets, which implies that its Hausdorft dimension is 1. But it is known Mandelbrot set has Hausdorft dimension 2, hence $\mathcal{M}$ cannot decide it. □

Chapter 18

Natural Proofs

SCRIBE: *Iannis Tourlakis*

Why have we not been able to prove strong lower bounds for circuits? In this lecture we consider the notion of "natural proofs" for separating complexity classes as put forth by Razborov and Rudich in 1994. In their paper, Razborov and Rudich show that current lowerbound arguments involve "natural" techniques, and show that obtaining strong lowerbound with such techniques would violate a widely believed cryptographic assumptions.

Basically, a natural technique is one that proves a lowerbound for a random function and is "constructive." We will formalize "constructive" later but let us consider why current techniques apply to random functions.

18.1 Formal Complexity Measures

We begin with an example.

EXAMPLE 25 Recall that a formula is a circuit with one output and in which each gate has in-degree 2 and outdegree 1. How would one go about proving lower bounds on formula size? Perhaps by induction? To that end, suppose we have a function like the one in Figure 18.1 that we believe to be "hard". If the function computed at the output is "complicated", intuitively it should be that at least one of the functions on the incoming edges to the output should also be complicated.

Let's try to formalize our intuition in the above example. Let μ be a function that maps each boolean function on $\{0,1\}^n$ to a nonnegative integer. (The input to μ is the truth table of the function.) We say that μ is a *formal complexity measure* if it satisfies the following properties: First, $\mu(x_i) \leq 1$ and $\mu(\bar{x}_i) \leq 1$ for all i. Second, we require that

- $\mu(f \wedge g) \leq \mu(f) + \mu(g)$ for all f, g; and
- $\mu(f \vee g) \leq \mu(f) + \mu(g)$ for all f, g.

For instance, the following function ρ is trivially a formal complexity measure

$$\rho(f) = \text{the smallest formula size for } f. \tag{18.1}$$

In fact, it is easy to prove the following by induction.

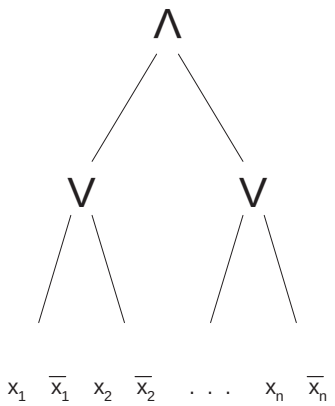


Figure 18.1: A formula for a hard function.

THEOREM 48
If μ is a formal complexity measure, then $\mu(f)$ is a lowerbound on the formula complexity of f.

Thus to prove a lowerbound for the formula size complexity of the CLIQUE, we would define a measure μ that is high for CLIQUE. For example, one could try "fraction of inputs for which the function agrees with the CLIQUE function" or some suitably scaled version of this. Unfortunately, such simple measures don't do the job, as you should check using the next few paragraphs. In general, one would imagine that a measure that lets us prove a good lowerbound for CLIQUE would involve some deep theorem about the CLIQUE function. The next lemma seems to show, however, that even though all we care about is the CLIQUE function, we have to in fact reason about random functions.

LEMMA 49
Suppose that μ is a formal complexity measure and that there exists a function $f : \{0,1\}^n \to \{0,1\}$ such that $\mu(f) \geq c$ for some large number c. Then for at least $1/4$ of all functions $g : \{0,1\}^n \to \{0,1\}$ we must have that $\mu(g) \geq c/4$.

PROOF: Let $g : \{0,1\}^n \to \{0,1\}$ be random. Write f as $f = h \oplus g$ where $h = f \oplus g$. Note that h is also random. So $f = (\bar{h} \wedge g) \vee (h \wedge \bar{g})$. Now suppose that $\{g : \mu(g) < c/4\}$ contains more than $3/4$ of all functions. Then $\Pr[\text{All of } h, \bar{h}, g, \bar{g} \text{ have } \mu < c/4] > 0$. Hence $\mu(f) < c$, which contradicts the assumption. Thus the lemma is proved. □

In fact, the following stronger theorem holds:

THEOREM 50
If $\mu(f) > c$ then for all $\epsilon > 0$ and for at least $1 - \epsilon$ of all functions g we have that,

$$\mu(g) \geq \Omega\left(\frac{c}{(n + \log(1/\epsilon))^2}\right).$$

The idea behind the proof of the theorem is to write f as the boolean combination of a small number of functions and then proceed similarly to the proof of the lemma.

18.2 Natural Properties

A *property* Φ is a map from boolean functions to $\{0,1\}$. A $\mathbf{P}$*-natural property useful against* $\mathbf{P}/poly$ is a property Φ such that:

(i) $\Phi(f) = 1$ for at least a $1/2^n$ fraction of all functions on n bits (recall that there are 2^{2^n} functions on n bits);

(ii) $\Phi(f) = 1$ implies that $f \notin \mathbf{P}/poly$ (or concretely, say, f has circuit complexity $n^{\log n}$); and

(iii) Φ is computable in $2^{O(n)}$ time.

Note that the property is called $\mathbf{P}$-natural because of requirement (3). The property is useful against $\mathbf{P}/poly$ because of requirement (2). Note that requirement (3) is somewhat controversial in that there is no broad consensus on whether it is reasonable.

Example 26 We have seen a natural property useful against $\mathbf{AC}^0$: this is the property that there does not exist a restriction on $n - n^\epsilon$ inputs function that turns the function into a constant function.

In fact, all known combinatorial techniques for obtaining circuit lowerbounds essentially involve constructing a natural property useful against the circuit class in question. In the case of $\mathbf{AC}^0$, the natural property is quite explicit in the proof, whereas for other classes, the natural property is hidden deeper inside the proof.

The following theorem by Razborov and Rudich possibly explains why we have not been able to use the same techniques to obtain an upper bound on $\mathbf{P}/poly$: constructing a natural property useful against $\mathbf{P}/poly$ violates widely believed cryptographic assumptions.

Theorem 51 (Razborov, Rudich)
Suppose a $\mathbf{P}$*-natural property* Φ *exists that is useful against* $\mathbf{P}/poly$*. Then there are no strong pseudorandom function generators. In particular, factoring and discrete log have subexponential algorithms.*

Before giving the proof we define pseudorandom function generators. Note that we will be tailoring a more general definition for our narrow purposes. We begin by defining pseudorandom function ensembles (again tailored to our needs).

Definition 42 *A* function ensemble *is a sequence* $F = \{F_n\}_{n=1}^\infty$ *of random variables where* F_n *assumes values in the set of functions mapping* $\{0,1\}^n$ *to* $\{0,1\}$*. In the special case where* F_n *is uniformly distributed over all such functions,* F *is called the* uniform function ensemble *and is denoted by* $H = \{H_n\}_{n=1}^\infty$*. A function ensemble* $F = \{F_n\}_{n=1}^\infty$ *is* pseudorandom *if for each Turing machines* M *running in time* $2^{O(n)}$*, and for all sufficiently large* n*,*

$$|\Pr[M(F_n) = 1] - \Pr[M(H_n) = 1]| < \frac{1}{2^{n^2}}.$$

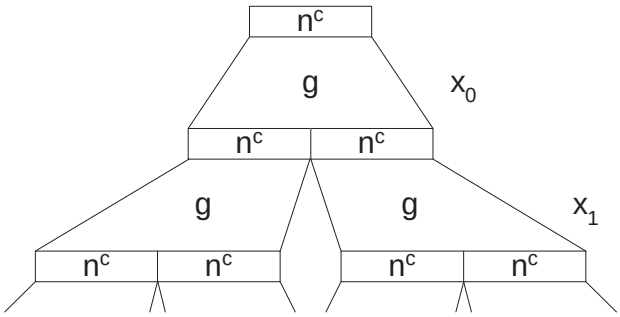


Figure 18.2: Constructing a pseudorandom function generator from a pseudorandom generator.

DEFINITION 43 *A* pseudorandom function generator *is a function $f(k,x)$ computable in polynomial time where the key k has n^c bits and the input x has n bits with the following property: if we let F_n be the random variable defined by uniformly selecting $k \in \{0,1\}^{n^c}$ and setting F_n to $f(k,\cdot)$, then the function ensemble $F = \{F_n\}_{n=1}^{\infty}$ is pseudorandom. We will denote $f(k,\cdot)$ by f_k.*

Intuitively, the above definition says that if f is a pseudorandom function generator, then for a random k, the probability is high that f_k "looks like a random function" to all Turing machines running in time $2^{O(n)}$. Note that f_k cannot look random to machines that run in $2^{O(n^c)}$ time since they can just guess the key k. Thus restricting the running time to $2^{O(n)}$ (or to some other fixed exponential function such as $2^{O(n^2)}$) is crucial.

As an aside, note that we can construct a pseudorandom function generator $f(k,x)$ using a pseudorandom generator g that stretches n^c random bits to $2n^c$ pseudorandom (also see Figure 18.2): Let $g_0(k)$ and $g_1(k)$ denote, respectively, the first and last n^c bits of $g(k)$. Then the following function is a pseudorandom function generator, where $\mathrm{MSB}(x)$ refers to the first bit of a string x:

$$f(k,x) = \mathrm{MSB}(g_{x_n} \circ g_{x_{n-1}} \circ \cdots \circ g_{x_2} \circ g_{x_1}(k)).$$

This construction is correct if the security parameter of g is large enough. Such a pseudorandom generator exists —by the Goldreich Levin theorem— if a oneway permutation exists that is extremely hard to invert. The discrete log problem —a permutation— is conjectured to be oneway. Many researchers believe that there is a small $\epsilon > 0$ such that the discrete log problem cannot be solved in 2^{n^ϵ} time. (They also believe the same for factoring.) If this belief is correct, then pseudorandom function generators exist as outlined above. (Exercise.)

We can now give the proof of the above theorem:

PROOF:[Theorem 51] Suppose the property Φ exists, and f is a pseudorandom function generator. We show that a Turing machine can use Φ to distinguish f_k from a random function. First note that $f_k \in \mathbf{P}/poly$ for every k (just hardwire k into the circuit for f_k) so the contrapositive of property (2) implies that $\Phi(f_k) = 0$. In addition, property (1) implies

that $\Pr_{H_n}[\Phi(H_n) = 1] \geq 1/2^n$. Hence,

$$\Pr_{H_n}[\Phi(H_n)] - \Pr_{k \in \{0,1\}^{n^c}}[\Phi(f_k)] \geq 1/2^n,$$

and thus Φ is a distinguisher against f. $\square$

We finish by noting that there is one technique that we have seen for proving lower bounds that does *not* fall under the category of natural proofs: diagonalization. Sadly, as we saw in Lecture 3, diagonalization can only be used to prove relativizing results.

Exercises

§i Prove Theorem 50.

§ii Show that if the hardness assumption for discrete log is true, then pseudorandom function generators exist.

§iii Show that a natural proof cannot prove a 2^{n^ϵ} lowerbound on the circuit size needed to invert discrete log, where $\epsilon > 0$ is any fixed constant.

Chapter 19

Quantum Computation

SCRIBE: *Zhifeng Chen, Jia Xu*

This lecture concerns quantum computation, an area that has become very popular recently because it promises to solve certain difficult problems —factoring and discrete logarithm— in polynomial time.

19.1 Quantum physics

Quantum phenomena are counterintuitive. To see this, consider the basic experiment of quantum mechanics that proves the wave nature of electrons: the 2-slit experiment. (See Figure 19.1.) A source fires electrons one by one at a wall. The wall contains two tiny slits. On the far side are small detectors that light up whenever an electron hits them. We measure the number of times each detector lights up during the hour. The results are as follows. When we cover one of the slits, we observe the strongest flux of electrons right behind the open slit, as one would expect. However, when both slits are open, we will see the "interference" phenomenon of electrons coming through two slits. In particular, at several detectors the total electron flux is *lower* when both slit are open as compared to when a single slit is open. This defies explanation if electrons behave like little balls, as implied in some high school textbooks. A ball could either go through slit 1 or slit 2, and hence the flux when both slits are open should be a simple sum of the fluxes when one of them is open.

The only explanation physics has for this experiment is that an electron does not behave as a ball. It should be thought of as *simultaneously* going through both slits at once, kind of like a wave. It has an *amplitude* for going through each slit, and this amplitude is a complex number (in particular, it can be a negative number). The chance of an electron appearing at a particular point p on the other side of the wall is related to

$$\text{amplitude of reaching } p \text{ via slit 1} + \text{amplitude of reaching } p \text{ via slit 2}.$$

Thus the points where the electron flux decreases when we open both slits are those where the two amplitudes have opposite sign.

"Nonsense!" you might say. "I need proof that the electron actually went through both slits." So you propose the following modification to the experiment. Position two detectors

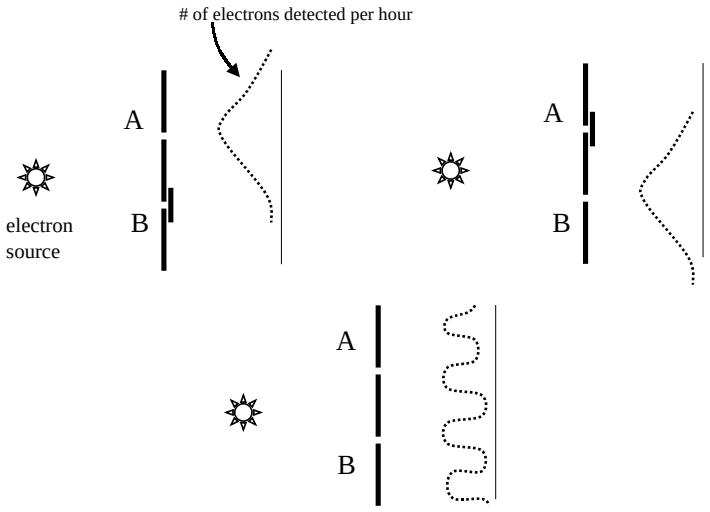


Figure 19.1: 2-slit experiment

at the slits; these light up whenever an electron passed through the slit. Now you can test the hypothesis that the electron went through both slits simultaneously.

Unfortunately, when you put such detectors at the slits, the interference phenomenon disappears on the other side of the wall! The explanation is roughly as follows: the quantum nature of particles disappers when they are under observation. More specifically, a quantum system has to evolve according to certain laws, and "nonreversible" operations —such as observation from nosy humans and their detectors— are not allowed. If these nonreversible operations happen, the quantum state collapses. (One moral to draw from this is that quantum computers, if they are ever built, will have to be carefully isolated from external influences and noise, since noise tends to be an irreversible operation. Of course, we can never completely isolate the system, which means we have to make quantum computation tolerant of a little noise. This is a topic of ongoing research.)

19.2 Quantum superpositions

Now we describe a *quantum register*, a basic component of the quantum computer. Recall the classical register, the building block of the memory in your desktop computer. An n-bit classical register with n bits consists of n particles. Each of them can be in 2 states: up and down, or 0 and 1. Thus there are 2^n possible configurations, and at any time the register is in one of these configurations.

The n-bit quantum register is similar, except at any time it can exist in a *superposition* of all 2^n configurations. (And the "bits" are called "qubits.") Each configuration $S \in \{0,1\}^n$ has an associated amplitude $\alpha_S \in \mathbf{C}$ where $\mathbf{C}$ is the set of complex numbers.

$$\alpha_S = \text{amplitude of being in configuration } S$$

Physicists like to denote this system state succinctly as $\sum_S \alpha_S |S\rangle$. This is their notation for describing a general vector in the vector space $\mathbf{C}^{2^n}$, expressing the vector as a linear

combination of basis vectors. The basis contains a vector $|S\rangle$ for each configuration S. The choice of the basis used to represent the configurations is immaterial so long as we fix a basis once and for all.

At every step, actions of the quantum computer —physically, this may involve shining light of the appropriate frequency on the quantum register, etc.— update α_S according to some physics laws. Each computation step is essentially a linear transformation of the system state. Let $\vec{\alpha}$ denote the current configuration (i.e., the system is in state $\sum_S \alpha_S |S\rangle$) and U be the linear operator. Then the next system state is $\vec{\beta} = U\vec{\alpha}$. Physics laws require U to be unitary, which means $UU^\dagger = I$. (Here $U^\dagger$ is the matrix obtained by transposing U and taking the complex conjugate of each entry.) Note an interesting consequence of this fact: the effect of applying U can be reversed by applying the operator $U^\dagger$: thus quantum systems are *reversible*. This imposes strict conditions on which kinds of computations are permissible and which are not.

As already mentioned, during the computation steps, the quantum register is isolated from the outside world. Suppose we open the system at some time and observe the state of the register. If the register was in state $\sum_S \alpha_S |S\rangle$ at that moment, then

$$\mathbf{Pr}[\text{we see configuration S}] = |\alpha_S|^2 \tag{19.1}$$

In particular, we have $\sum_S |\alpha_S|^2 = 1$ at all times. Note that observation is an irreversible operator. We get to see one configuration according to the probability distribution described in (19.1) and and the rest of the configurations are lost forever.

What if we only observe a few bits of the register —a so-called *partial observation*? Then the remaining bits still stay in quantum superposition. We show this by an example.

Example 27 Suppose an n-bit quantum register is in the state

$$\sum_{s\in\{0,1\}^{n-1}} \alpha_s |0\rangle |s\rangle + \beta_s |1\rangle |s\rangle \tag{19.2}$$

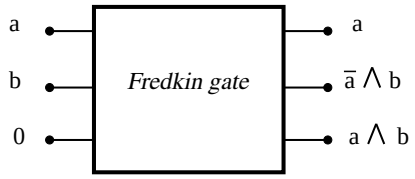
(sometimes this is also represented as $\sum_{s\in\{0,1\}^{n-1}} (\alpha_s |0\rangle + \beta_s |1\rangle) |s\rangle$, and we will use both representations). Now suppose we observe just the first bit of the register and find it to be 0. Then the new state is

$$\frac{1}{\sqrt{|\alpha_s|^2}} \sum_{s\in\{0,1\}^{n-1}} \alpha_s |0\rangle |s\rangle \tag{19.3}$$

where the first term is a rescaling term that ensures that probabilities in future observations sum to 1.

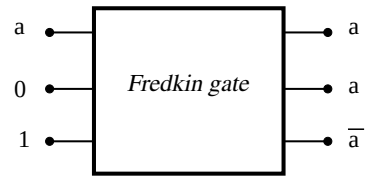
19.3 Classical computation using reversible gates

Motivated by the 2nd Law of Thermodynamics, researchers have tried to design computers that expend —at least in principle— zero energy. They have invented *reversible gates*, which can implement all classical computations in a reversible fashion. We will study reversible classical gates as a stepping stone to quantum gates; in fact, they are simple examples of quantum gates.



Implementing AND

Figure 19.2



Implementing NOT (and copy)

Figure 19.3: Implementing NOT and COPY with Fredkin Gate

The *Fredkin gate* is a popular reversible gate. It has 3 Boolean inputs and on input (a, b, c) it outputs (a, b, c) if $a = 1$ and (a, c, b) if $a = 0$. It is *reversible*, in the sense that $F(F(a, b, c)) = (a, b, c)$. Simple induction shows that if a circuit is made out of Fredkin gates alone and has m inputs then it must have m outputs as well. Furthermore, we can recover the inputs from the outputs by just applying the circuit in reverse. Hence a Fredkin gate circuit is *reversible*.

The Fredkin gate is *universal*, meaning that every circuit of size S that uses the familiar AND, OR, NOT gates (maximum fanin 2) has an equivalent Fredkin gate circuit of size $O(S)$. We prove this by showing that we can implement AND, OR, and NOT using a Fredkin gate some of whose inputs have been fixed 0 or 1 (these are "control inputs"); see Figure 19.2 for AND and Figure 19.3 for NOT; we leave OR as exercise. We also need to show how to copy a value with Fredkin gates, since in a normal circuit, gates can have fanout more than 1. To implement a COPY gate using Fredkin gates is easy and is the same as for the the NOT gate (see Figure 19.3).

Thus to transform a normal circuit into a reversible circuit, we replace each gate with its Fredkin implementation, with some additional "control" inputs arriving at each gate to make it compute as AND/OR/NOT. These inputs have to be initialized appropriately.

The transformation appears in Figure 19.4, where we see that the output contains some junk bits. With a little more work (see Exercises) we can do the transformation in such a way that the output has no junk bits, just the original control bits. The reversible circuit starts with some input bits that are initialized to 0 and these are transformed into output bits.

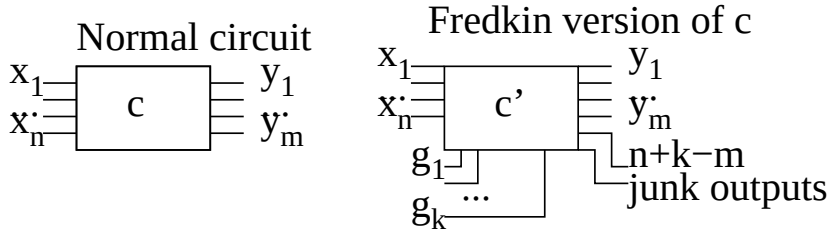


Figure 19.4: Converting a normal circuit c into an equivalent circuit c' of Fredkin gates. Note that we need additional control inputs

19.4 Quantum gates

A 1-input quantum gate is represented by a unitary 2×2 matrix $U = \left(\begin{smallmatrix} U_{00} & U_{01} \\ U_{10} & U_{11} \end{smallmatrix}\right)$. When its input bit is 0 the output is the superposition $U_{00}|0\rangle + U_{01}|1\rangle$ and when the input is 1 the output is the superposition $U_{10}|0\rangle + U_{11}|1\rangle$. When the input bit is in the superposition $\alpha_0|0\rangle + \beta_0|1\rangle$ the output bit is a superposition of the corresponding outputs

$$(\alpha_0 U_{00} + \beta_0 U_{10})|0\rangle + (\alpha_0 U_{01} + \beta_0 U_{11})|1\rangle. \tag{19.4}$$

More succinctly, if the input state vector is (α_0, β_0) then the output state vector is

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix} = U \begin{pmatrix} \alpha_0 \\ \beta_0 \end{pmatrix}$$

If $|\alpha|^2 + |\beta|^2 = 1$ then unitarity of U implies that $|\alpha'|^2 + |\beta'|^2 = 1$.

Similarly, a 2-input quantum gate is represented by a unitary 4×4 matrix R. When the input is the superposition $\alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$, the output is $\beta_{00}|00\rangle + \beta_{01}|01\rangle + \beta_{10}|10\rangle + \beta_{11}|11\rangle$ where

$$\begin{pmatrix} \beta_{00} \\ \beta_{01} \\ \beta_{10} \\ \beta_{11} \end{pmatrix} = R \begin{pmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{pmatrix}$$

In general, a quantum gate with k inputs is specified by a unitary $2^k \times 2^k$ matrix.

Example 28 A Fredkin gate is also a valid 3-input quantum gate. We represent it by an 8×8 matrix that gives its output on all 2^3 possible inputs. This matrix is a permutation matrix (i.e., obtainable from the identity matrix by applying a permutation on all the rows) since the output $F(a, b, c)$ is just a permutation of the input (a, b, c). Exercise 3 asks you to verify that this permutation matrix is unitary.

A quantum circuit on n inputs consists of (a) an n-bit quantum register (b) a sequence of gates $(g_j)_{j=1,2,\ldots}$. If g_j is a k-input gate, then the circuit specification has to also give a sequence of bit positions $(j, 1), (j, 2), \ldots, (j, k) \in [1, n]$ in the quantum register to which this gate is applied. The circuit computes by applying these gate operations to the quantum register one by one in the specified order. The register holds the state of the computation, and only one gate is applied at any given time.

EXAMPLE 29 Suppose we have an n-bit quantum register in the state $\sum_{S\in 0,1^n} \alpha_S |S\rangle$. If we apply a 1-input quantum gate U to the first wire, the new system state is computed as follows. First "factor" the initial state by expressing each n-bit configuration as a concatenation of the first bit with the remaining $n-1$ bits:

$$\sum_{S'\in\{0,1\}^{n-1}} \alpha_{0,S'} \left|0S'\right\rangle + \alpha_{1,S'} \left|1S'\right\rangle. \tag{19.5}$$

(Formally we could express everything we are doing in terms of tensor product of vector spaces but we will not do that.)

To obtain the final state apply U on the first bit in each configuration as explained in equation (19.4). This yields

$$\sum_{S'\in\{0,1\}^{n-1}} (\alpha_{0,S'}U_{00} + \alpha_{1,S'}U_{10}) \left|0S'\right\rangle + (\alpha_{0,S'}U_{01} + \alpha_{1,S'}U_{11}) \left|1S'\right\rangle \tag{19.6}$$

We can similarly analyze the effect of applying a k-input quantum gate on any given set of k bits of the quantum register, by first "factoring" the state vector as above.

19.4.1 Universal quantum gates

You may now be a little troubled by the fact that the set of possible 1-input quantum gates is the set of all unitary 2×2 matrices, an uncountable set. That seems like bad news for the Radio Shacks of the future, who may feel obliged to keep all possible quantum gates in their inventory, to allow their customers to build all possible quantum circuits.

Luckily, Radio Shack need not fear. Researchers have shown the existence of a small set of "universal" 2-input quantum gates such that every circuit composed of S arbitrary k-input quantum gates can be simulated using a circuit of size $2^{\text{poly}(k)} \cdot S\text{poly}(\log S)$ composed only of our universal gates. The simulation is not exact and the distribution on outputs is only approximately the same as the one of the original circuit. (All of this assumes the absence of any outside noise; simulation in presence of noise is a topic of research and currently seems possible under some noise models.)

In any case, we will not need any fancy quantum gates below; just the Fredkin gate and the following 1-input gate called the Hadamard gate:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

19.5 BQP

DEFINITION 44 *A language $L \subseteq \{0,1\}^*$, is in* **BQP** *iff there is a family of quantum circuits (C_n) of size n^c s.t. $\forall x \in \{0,1\}^*$:*

$$x \in L \Rightarrow \mathbf{Pr}[C(x)_1 = 1] \geq \frac{2}{3}$$
$$x \notin L \Rightarrow \mathbf{Pr}[C(x)_1 = 1] \leq \frac{1}{3}$$

Here $C(x)_1$ is the first output bit of circuit C, and the probability refers to the probability of observing that this bit is 1 when we "observe" the outputs at the end of the computation.

The circuit has to be uniform, that is, a deterministic polynomial time (classical) Turing machine must be able to write down its description.

At first the uniformity condition seems problematic because a classical Turing machine cannot write complex numbers needed to describe quantum gates. However, the machine can just express the circuit approximately using universal quantum gates, which is a finite family. The approximate circuit computes the same language because of the gap between the probabilities 2/3 and 1/3 used in the above definition.

THEOREM 52
P ⊆ BQP

PROOF: Every language in **P** has a uniform circuit family of polynomial size. We transform these circuits into reversible circuits using Fredkin gates, thus obtaining a quantum circuit family for the language. □

THEOREM 53
BPP ⊆ BQP

PROOF: Every language in **BPP** has a uniform circuit family (C_n) of polynomial size, where circuit C_n has n normal input bits and an additional n^k input wires that have to be initialized with random bits.

We transform the circuit into a reversible circuit C'_n using Fredkin gates. To produce "random" bits, we feed $O(n^c)$ zeros into an array of Hadamard gates and plug their outputs into C'_n; see Figure 19.5.

To see that this works, imagine fixing the first n bits to a string x in both circuits.

A simple induction shows that if we start with an N-bit quantum register in the state $\left|\vec{0}\right\rangle$ and apply the Hadamard gate one by one on all the bits, then we obtain the superposition

$$\sum_{S \in 0,1^N} \frac{1}{2^{N/2}} |S\rangle \tag{19.7}$$

Thus the "random" inputs of circuit C'_n get a uniform quantum superposition of all possible bit strings. Thus the output bit of C'_n is a uniform superposition of the result on each bit string and observing it at the end gives the probability that C_n accepts when fed a random input. □

19.6 Factoring integers using a quantum computer

This section proves the following famous result.

THEOREM 54 (SHOR 1994)
There is a polynomial size quantum circuit that factors integers.

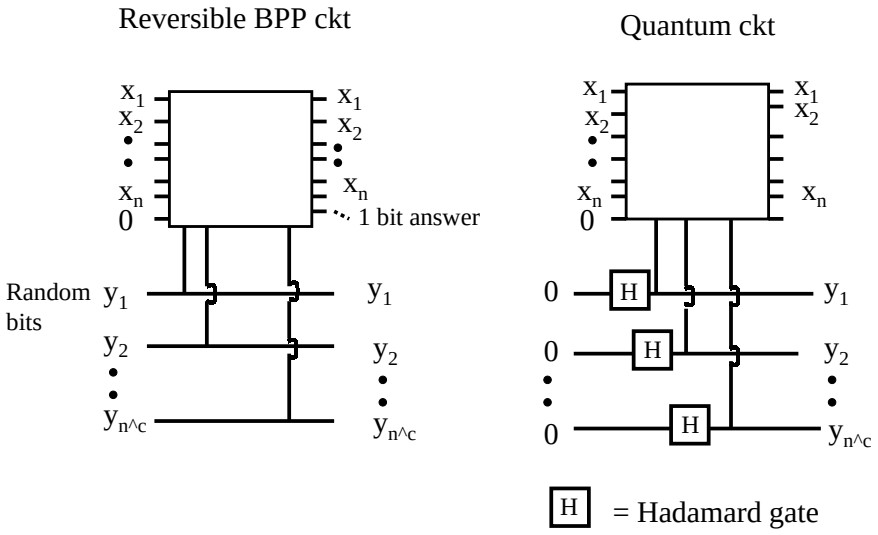


Figure 19.5: Turning a BPP circuit into an equivalent quantum circuit. An array of Hadamard gates produces random bits.

We describe Kitaev's proof of this result since it uses eigenvalues, a more familiar and intuitive concept than the Fourier transforms used in Shor's algorithm.

DEFINITION 45 (EIGENVALUE) *λ is an* eigenvalue *of matrix M if there is a vector e (called the* eigenvector*) , s.t.:*

$$M \cdot e = \lambda e$$

Fact: If M is unitary, then $|\lambda| = 1$. In other words there is a $\theta \in [0, 1)$ such that

$$\lambda = e^{2\pi i\theta} = \cos(2\pi\theta) + i\sin(2\pi\theta).$$

Fact: If $M \cdot e = \lambda e$ then $M^k \cdot e = \lambda^k e$. Hence e is still an eigenvector of M^k and λ^k is the corresponding eigenvalue.

Now we are ready to describe Kitaev's algorithm and prove Theorem 54.

PROOF: Let N be the number to be factored. As usual, Z_N^* is the set of numbers mod N that are co-prime to N. Simple number theory shows that for every $a \in Z_N^*$ there is a smallest integer r such that $a^r \equiv 1 \pmod{N}$; this r is called the *order* of a. The algorithm uses the well-known fact that if we can compute the order of random elements of Z_N^* then we can factor N with high probability. First, note that with probability at least $1/2$, the element a has even order. Now if $(a^r - 1) \equiv 0 \pmod{N}$, then $(a^{\frac{r}{2}} - 1)(a^{\frac{r}{2}} + 1) \equiv 0 \pmod{N}$. If a is random, with probability $\geq \frac{1}{2}$, $a^{\frac{r}{2}} \neq 1 \pmod{N}, a^{\frac{r}{2}} \neq -1 \pmod{N}$ (this is a simple exercise using the Chinese remainder theorm) and hence $gcd(N, a^{\frac{r}{2}} - 1) \neq N, 1$. Thus, knowing r we can compute $a^{r/2}$ and compute $gcd(N, a^{\frac{r}{2}} - 1)$. Thus with probability at least $1/4$ (over the choice of a) this method yields a factor of N.

The factoring algorithm is a mixture of a classical and a quantum algorithm. Using classical random bits it generates a random $a \in Z_N^*$ and then constructs a quantum circuit.

Observing the output of this quantum circuit a few times followed by some more classical computation allows it to obtain r, the order of a, with reasonable probability. (Of course, we could in principle describe the entire algorithm as a quantum algorithm instead of as a mixture of a classical and a quantum algorithm, but our description isolates exactly where quantum mechanics is crucial.)

Consider a classical reversible circuit that acts on numbers in Z_N^*, and is described by $U(x) = ax \pmod N$. Then we can view this circuit as a quantum circuit operating on a quantum register. If the quantum register is in the superposition[1]

$$\sum_{x \in Z_N^*} \alpha_x \, |x\rangle,$$

then applying U gives the superposition

$$\sum_{x \in Z_N^*} \alpha_x \, |\, ax \pmod N\rangle.$$

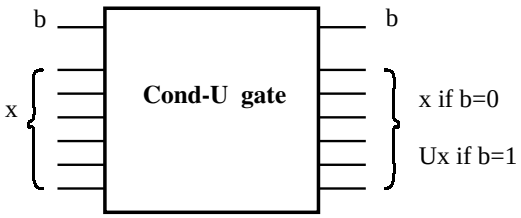
Interpret this quantum circuit as an $N \times N$ matrix —also denoted U—and consider its eigenvalues. Since $U^r = I$, we can easily check that each eigenvalue has the form $e^{2\pi i\theta}$ where $\theta = \frac{j}{r}$ for some $j \leq r$. The algorithm will try to obtain a random eigenvalue. It thus obtains —in binary expansion— a number of form $\frac{j}{r}$ where j is random. Chances are good that this j is coprime to r, which means that $\frac{j}{r}$ is an irreducible fraction. Even knowing only the first $2 \log N$ bits in the binary expansion of $\frac{j}{r}$, the algorithm can round off to the nearest fraction whose denominator is at most N (this can be done; easy number theory) and then it reads off r from the denominator.

Now we describe how to compute the first $2 \log N$ bits of a random eigenvalue of U. Assume for now that the algorithm has a quantum register whose state is a superposition, denoted $\vec{e}$, corresponding to a random eigenvector of U. (See below for details on how to put a register in such a state.) Then applying U gives the final state $\lambda\vec{e}$, where λ is the eigenvalue associated with $\vec{e}$. Thus the register's state has undergone a *phase shift* —i.e., multiplication by a scalar— although there is yet no direct way to measure λ.

Now define a conditional-U circuit (Figure 19.6), whose input is (b, x) where b is a bit, and cond-$U(b, x) = (b, x)$ if $b = 0$ and $(b, ax \pmod N)$ if $b = 1$. Again, notice that this can be implemented using classical Fredkin gates.

Using a cond-U circuit and two Hadamard gates, we can build a quantum circuit shown in Figure 19.7. When this is applied to a quantum register whose first bit is 0 and the remaining bits are in a state $\vec{e}$, then we can measure the corresponding eigenvalue λ by

[1] Aside: This isn't quite correct; the cardinality of Z_N^* is not a power of 2 so it is not immediately obvious how to construct a quantum register whose only possible configurations correspond to elements of Z_N^*. However, if we use the nearest power of 2, everything we are about to do will still be approximately correct. There are also other fixes to this problem.



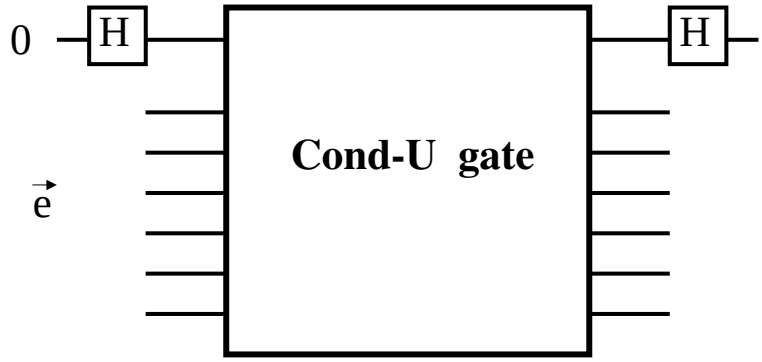


Figure 19.7: Basic block: Conditional-U gate and two Hadamard gates.

repeated measurement of the first output bit.

$$\begin{aligned} |0\rangle|e\rangle &\xrightarrow{H_1} \frac{1}{\sqrt{2}}|0\rangle|e\rangle + \frac{1}{\sqrt{2}}|1\rangle|e\rangle \\ &\xrightarrow{\text{cond-U}} \frac{1}{\sqrt{2}}|0\rangle|e\rangle + \frac{\lambda}{\sqrt{2}}|1\rangle|e\rangle \\ &\xrightarrow{H_2} \frac{1}{2}((1+\lambda)|0\rangle|e\rangle + (1-\lambda)|1\rangle|e\rangle) \end{aligned} \tag{19.8}$$

Thus the probability of measuring a 0 in the first bit is proportional to $|1+\lambda|$. We will refer to this bit as the *phase bit*, since repeatedly measuring it allows us to compute better and better estimates to λ. Actually, instead of measuring repeatedly we can just design a quantum circuit to do the repetitions by noticing that the output is just a scalar multiple of $\vec{e}$ again, so we can just feed it into another basic block with a fresh phase bit, and so on (see Figure 19.8). We measure phase bits all at once at the end.

Are we done? Unfortunately, no. Obtaining an estimate to the first m bits of λ would involve a circuit with 2^m phase bits (this is a simple exercise about probabilistic estimation), and when $m = 2\log N$, this number is about N, whereas we are hoping for a circuit size of

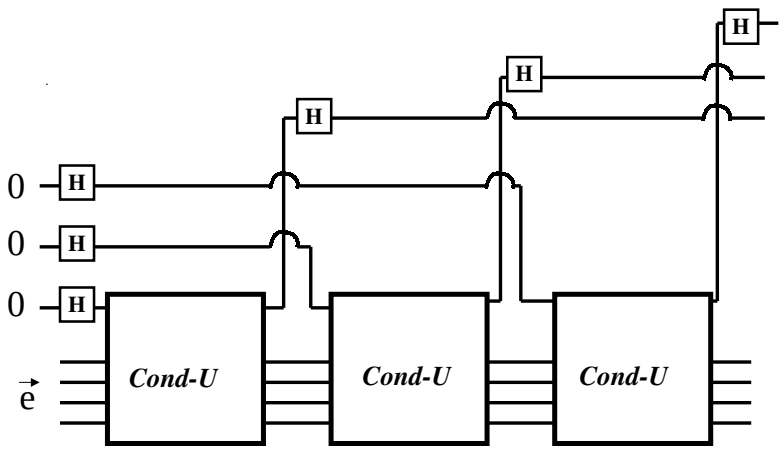


Figure 19.8: Repeating the basic experiment to get better estimate of λ.

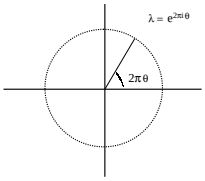


Figure 19.9: Eigenvalue $\lambda = e^{2\pi i\theta}$ in the complex plane.

poly($\log N$). Thus simple repetition is a very inefficient way to obtain accurate information about λ.

A more efficient technique involves the observation that U has a special property: powers of U also have small circuits. Specifically, $U^{2^k}(x) = a^{2^k}x \pmod N$, and a^{2^k} is computable by circuits of size poly($\log N + \log k$) using fast exponentiation.

Thus we can implement a conditional-U^{2^k} gate using a quantum circuit of size poly($\log N + k$). The eigenvalues of U^{2^k} are λ^{2^k}. If $\lambda = e^{2\pi i\theta}$ where $\theta \in [0, 1)$ (see Figure 19.9) then $\lambda^{2^k} = e^{2\pi i\theta 2^k}$. Of course, $e^{2\pi i\theta 2^k}$ is the same complex number as $e^{2\pi i\alpha}$ where $\alpha = 2^k\theta \pmod 1$. Thus measuring λ^{2^k} gives us $2^k\theta \pmod 1$ and in particular the most significant bit of $2^k\theta \pmod 1$ is nothing but the kth bit of θ. Using $k = 0, 1, 2, \ldots 2\log N$ we can obtain[2] the first $2\log N$ bits of θ.

As in Figure 19.8, we can bundle these steps into a single cascading circuit where the output of the conditional-$U^{2^{k-1}}$ circuit feeds into the conditional-U^{2^k} circuit. Each circuit has its own set of $O(\log N)$ phase bits; measuring the phase bits of the kth circuit gives an estimate of the kth bit of θ that is correct with probability at least $1 - 1/N$. All phase bits are measured in a single stroke at the end.

[2]There is a slight inaccuracy here, since we are blurring the distinction between measuring λ and θ. For example, when θ is very close to $1/2$, the complex number $1 + \lambda$ is close 0 and the phase bit almost always comes up 1. So instead we do overlapping observations, the first estimating the values of the 1st and 2nd bits, the second estimating the values of the 2nd and 3rd bits, etc.

To finish, we show how to put a quantum register into a state corresponding to a random eigenvector.

19.6.1 Uniform superpositions of eigenvectors of U

Actually, we show how to put the quantum register into a uniform superposition of eigenvectors of U. This suffices for our cascading circuit, as we will argue shortly.

First we need to understand what the eigenvectors look like. Recall that $\left\{1, a, a^2, \ldots, a^{r-1}\right\}$ is a subgroup of Z_N^*. Let B be a set of representatives of all cosets of this subgroup. In other words, for each $x \in Z_N^*$ there is a unique $b \in B$ and $l \in \{0, 1, \ldots, r-1\}$ such that $x = ba^l \pmod N$. Then the following is the complete set of eigenvectors, where $\omega = e^{\frac{2\pi i}{r}}$:

$$\forall j \in \{0, 1, \ldots, r-1\}, \forall b \in B \qquad \vec{e}_{j,b} = \sum_{l=0}^{r-1} \omega^{jl} \left| ba^l \pmod N \right\rangle \tag{19.9}$$

The eigenvalue associated with this eigenvector is $\omega^{-j} = e^{-\frac{2\pi ij}{r}}$.

Fix b and consider the uniform superposition:

$$\frac{1}{r}\sum_{j=0}^{r-1} \vec{e}_{j,b} = \frac{1}{r}\sum_{j=0}^{r-1}\sum_{l=0}^{r-1} \omega^{jl} \left| ba^l \pmod N \right\rangle \tag{19.10}$$

$$= \frac{1}{r}\sum_{l=0}^{r-1}\sum_{j=0}^{r-1} \omega^{jl} \left| ba^l \pmod N \right\rangle. \tag{19.11}$$

Separating out the terms for $l = 0$ and using the formula for sums of geometric series:

$$= \frac{1}{r}\left(\sum_{j=0}^{r-1} |b\rangle + \sum_{l=1}^{r-1} \frac{(\omega^l)^r - 1}{\omega^l} \left| ba^l \pmod N \right\rangle\right) \tag{19.12}$$

since $\omega^r = 1$ we obtain

$$= |b\rangle \tag{19.13}$$

□

Thus if we pick an arbitrary b and feed the state $|b\rangle$ into the quantum register, then that can also be viewed as a uniform superposition $\frac{1}{r}\sum_j \vec{e}_{j,b}$.

19.6.2 Uniform superposition suffices

Now we argue that in the above algorithm, a uniform superposition of eigenvectors is just as good as a single eigenvector.

Fixing b, the initial state of the quantum register is

$$\frac{1}{r}\sum_j \left|\vec{0}\right\rangle |\vec{e}_{j,b}\rangle,$$

where $\vec{0}$ denotes the vector of phase bits that is initialized to 0. After applying the quantum circuit, the final state is

$$\frac{1}{r}\sum_j |c_j\rangle\,|\vec{e}_{j,b}\rangle,$$

where $|c_j\rangle$ is a state vector for the phase bits that, when observed, gives the first $2\log N$ bits of j/r with probability at least $1 - 1/N$. Thus observing the phase bits gives us whp a random eigenvalue.

Exercises

§i Implement an OR gate using the Fredkin gate.

§ii Given any classical circuit computing a function f from n bits to n bits, describe how to compute the same function with a reversible circuit that is a constant factor bigger and has no "junk" output bits. (Hint: Use our transformation, then copy the output to a safe place, and then run the circuit in reverse to erase the junk outputs.)

§iii Verify that the Fredkin gate is a valid quantum gate.

§iv Verify all the number theoretic facts mentioned in the description of the factoring algorithm.

Chapter 20

Homeworks

PRINCETON UNIVERSITY COS 522: COMPUTATIONAL COMPLEXITY

Homework 1

Out: *February 14* Due: *February 26*

You can collaborate with your classmates, but be sure to list your collaborators with your answer. Also, limit your answers to one page or less —you just need to give enough detail to convince me. If you suspect a problem is open, just say so and give reasons for your suspicion.

§i Show that $\Sigma_2^p = \mathbf{NP}^{\mathrm{SAT}}$.

§ii Show that $\mathbf{SPACE}(n) \neq \mathbf{NP}$. (Hint: Use padding, mentioned in the notes for Lecture 1.)

§iii Can you give a definition of **NEXPTIME** in terms of certificates as we did for **NP**? If not, report your best attempt.

§iv 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 $\mathbf{DTIME}(n^{1.1})$ superior to $\mathbf{DTIME}(n)$?

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

§v Suppose we define the *logspace hierarchy* in analogy with the polynomial hierarchy using logspace machines that can use alternation. Does this hierarchy collapse by Immerman's theorem ($\mathbf{NL} = \mathbf{coNL}$)?

PRINCETON UNIVERSITY COS 522: COMPUTATIONAL COMPLEXITY

Homework 2

Out: *March 5* Due: *March 14*

You can collaborate with your classmates, but be sure to list your collaborators with your answer. Also, limit your answers to one handwritten page, preferably less —you just need to give enough detail to convince me. If some question doesn't seem correct or solvable, indicate why.

§i Show that for each $k > 0$ there there is a language in **PH** that is not decidable by circuit families of size n^k. (Hint: Diagonalization.)

§ii Let $\mathcal{H}$ be a family of 2-universal hash functions mapping $\{0,1\}^n$ to $\{0,1\}^m$ where $n > m$. Let $S \subseteq \{0,1\}^n$ have size at least 2^m. Show that

$$\Pr_{h \in \mathcal{H}}[h(S) \neq \{0,1\}^m] \leq \frac{2^{2m+1}}{|S|}.$$

§iii Let $f, g : \{0,1\}^* \to \mathbf{N}$ be functions and $c > 1$. We say that f *approximates* g *within a factor* c if for every string x, $g(x) \leq f(x) \leq c \cdot g(x)$. Show that for every $g \in \#\mathbf{P}$ and every $\epsilon > 0$, there is a function in $\mathbf{FBPP}^{\mathrm{SAT}}$ that approximates g within a factor $1 + \epsilon$. (Hint: Use the previous Problem.)

§iv The XOR casino offers the following game. A sequence of k cards all labelled with 0 or 1 is laid on the table face-down. The cards are chosen by the casino, with just one restriction (under state gambling laws): with probability at least p, all k cards must be 1. Now a card is picked uniformly at random and all the other cards are turned face up. You are asked to guess the number on the hidden card; if you guess correctly you receive a payoff of \$ 1.

This question explores what your expected payoff can be.

(a) Suppose the casino's strategy is the following: with probability p make all cards 1 and otherwise make each card 1 with probability 1/2 and 0 with probability 1/2. Show that your expected payoff then is at most $1/2 + p/2$, and that you have a strategy that achieves this payoff.

(b) Now suppose the casino strategy is unknown to you (except you know that it obeys state laws). Suppose you use the following guessing strategy: if all $k-1$ cards you saw had a 1, you guess that the k'th one is 1, otherwise you guess 0 or 1 with equal probability.

Show that then the casino has a strategy to make your expected payoff at most

$$\frac{1}{2} + \frac{p}{2} - \frac{1-p}{2k}.$$

(c) Now suppose you adopt the following strategy: if among the $k-1$ cards revealed to you, t cards have a 0 on them, then you guess with probability $(1+2^{-t})/2$ that the kth card has a 1 and with probability $(1-2^{-t})/2$ that it has a 0.

Show that with this strategy your expected payoff is at least $1/2 + p/2 - 2^{-k/3}$, irrespective of the casino's strategy (so long as it obeys state laws).

§v Use the previous question to prove the following version of the Yao XOR Lemma. Suppose $f : \{0,1\}^n \to 0,1$ is a function such that no circuit of size $S(n)$ can, given a random x, predict $f(x)$ with probability 3/4. Let $f^{\oplus k} : \{0,1\}^{(n+1)k} \to 0,1$ be the function that breaks its input into k parts $y_1, y_2, \ldots, y_k$ of n bits each and one part r of k bits. It computes the k-tuple $(f(y_1), f(y_2), \ldots, f(y_k))$ and outputs its inner product (mod 2) with r.

Then no circuit of size $t(nk)$ can predict $f^{\oplus k}$ for more than $1/2 + s(nk)$ fraction of inputs, where $poly(s(nk)t(nk)) \leq S(n)$. (Hint: The circuit could have hardwired answers to quite a few random inputs; think of these as the $k-1$ cards of the casino game.)

PRINCETON UNIVERSITY COS 522: COMPUTATIONAL COMPLEXITY

Homework 3

Out: *March 28* Due: *April 4*

You can collaborate with your classmates, but be sure to list your collaborators with your answer. Also, limit your answers to one handwritten page, preferably less —you just need to give enough detail to convince me. If some question doesn't seem correct or solvable, indicate why.

§i (a) Show that $\mathbf{AM}[2] \subseteq \mathbf{NP}/\text{poly}$. (b) Show that if $\overline{SAT} \in \mathbf{NP}/\text{poly}$ then $\mathbf{PH} = \mathbf{\Sigma}_3^p$. (c) Conclude that if Graph Isomorphism is **NP**-complete under polynomial time reductions, then $\mathbf{PH} = \mathbf{\Sigma}_3^p$

§ii A *program checker* for a computational problem π is a probabilistic algorithm C. Given any program P that supposedly computes π and an input x, the checker calls P at most poly($|x|$) times, and runs in poly($|x|$) time (this does not include the time required by P).

(a) If P correctly computes π on every input, then with probability at least 0.99, C outputs "CORRECT" on x.

(b) If $P(x) \neq \pi(x)$, then with probability at least 0.99, C outputs "BUGGY" on x.

Show that Graph Isomorphism and Discrete Log (for a specific prime p) have program checkers. What about SAT?

§iii A degree d polynomial is one whose degree in each variable is at most d. Let the *distance* between two functions f, g (denoted $\Delta(f, g)$) be the fraction of points on which f, g disagree. Show that if f, g are degree d polynomials in m variables, then $\Delta(f, g) \geq 1 - md/|F|$. (Hint: Use induction on m.)

§iv Let $\Delta_d(f)$ denote the distance of f to the nearest degree d polynomial.

Suppose we are given the table of values of a function $f : F^m \to F$. Supposedly this function is a degree d polynomial in m variables over field $F = Z_q$, but this needs to be checked. Let $q = \Omega(m^3 d^3)$. Consider the following tester:

Pick $i \in_R \{1, \ldots, m\}$ and $a_1, a_2, \ldots, a_m \in_R F$ randomly. For $s = 0, 1, \ldots, d$, read $f(a_1, \ldots, a_{i-1}, s, a_{i+1}, \ldots, a_m)$ from the table. Let these values be $b_0, \ldots, b_d$ respectively. Let $g(x)$ be a degree d univariate polynomial such that $g(i) = b_i$. ACCEPT iff $f(a_1, a_2, \ldots, a_m) = g(a_i)$.

Show that there is a constant $c > 0$ such that the probability this test accepts is at most $1 + \sqrt{\frac{md}{q}} - \min\left\{c\Delta_d(f), \frac{1}{10md}\right\}$. Report partial progress on this problem too, but keep it brief. (Hint: Use some kind of induction on m and also use Problem 3. Note that the test is implicitly defining m functions, one for each value of i. The ith function has the property that fixing all but the ith coordinate gives a polynomial of degree d in that variable. The induction should use properties of such functions.)

PRINCETON UNIVERSITY COS 522: COMPUTATIONAL COMPLEXITY

Homework 4

Out: *April 16* Due: *April 25*

You can collaborate with your classmates, but be sure to list your collaborators with your answer. Also, limit your answers to one handwritten page, preferably less —you just need to give enough detail to convince me. If some question doesn't seem correct or solvable, indicate why.

For this problem set, assume that circuits consist of $\vee$ *and* $\wedge$ *gates with fanin* 2 *and the inputs are* $x_1, x_2, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}$*. In other words, the* $\neg$ *gates have been pushed down to the inputs. The depth of the circuit is the number of edges (wires) in the longest path from output to an input.*

§i Let $\mathcal{A}$ be a class of deterministic algorithms for a problem and I be the set of possible inputs. Both sets are finite. Let $\text{cost}(A, x)$ denote the cost of running algorithm A on input x. Prove Yao's lemma:

$$\max_{\mathcal{D}} \min_{A \in \mathcal{A}} \mathbf{E}_{x \in \mathcal{D}}[\text{cost}(A, x)] \quad = \quad \min_{\mathcal{P}} \max_{x \in I} \mathbf{E}_{A \in \mathcal{P}}[\text{cost}(A, x)],$$

where $\mathcal{D}$ is a probability distribution on I and $\mathcal{P}$ is a probability distribution on $\mathcal{A}$.

Does this result hold if the class of algorithms and inputs is infinite?

§ii For any graph G with n vertices, consider the following communication problem: Player 1 receives a clique C in G, and Player 2 receives an independent set I. They have to communicate in order to determine $|C \cap I|$. (Note that this number is either 0 or 1.) Prove an $O(\log^2 n)$ upperbound on the communication complexity.

Can you improve your upperbound or prove a lower bound better than $\Omega(\log n)$?

§iii Associate the following communication problem with any function $f : \{0,1\}^n \to \{0,1\}$. Player 1 gets any input x such that $f(x) = 0$ and player 2 gets any input y such that $f(y) = 1$. They have to communicate in order to determine a bit position i such that $x_i \neq y_i$.

Show that the communication complexity of this problem is *exactly* the minimum depth of any circuit that computes f.

§iv Use the previous question to show that computing the parity of n bits requires depth at least $2 \log n$.

§v Show that in any circuit with m edges and depth d, there are $km/logd$ edges whose removal yields a circuit of depth less than $d/2^{k-1}$.

PRINCETON UNIVERSITY COS 522: COMPUTATIONAL COMPLEXITY

Homework 5

Out: *March 26* Due: *May 10*

For undergrads: Visit the library and look up a few journals on complexity theory. Pick a research paper (one not covered in class) and do a 2-page typewritten summary. The summary should indicate the main ideas of the paper. Keep in mind that you're writing for me, so you can omit many details.

For grads: You could do the undergrads' assignment for a maximum grade of B+. To get a better grade your assignment is to prove a new theorem. If this new theorem is publishable then you get an A+ on the assignment. You can collaborate with one other person in the class.

Some journals that carry articles on complexity theory: Journal of the Association for Computing Machinery (JACM), SIAM J. Computing, Computational Complexity, Journal of Computer and System Sciences, Computational Complexity, Theoretical Computer Science, Electronic Colloquium on Computational Complexity (ECCC).