These are scribed notes of my lectures on **Advanced Complexity Theory (MIT 6.841/18.405)** as taught in Spring 2002. Thanks to Mike Aleknovich, Ben Barden, Deniss Čebikins, Alice Chan, Austin Che, Johnny Chen, Eleni Drinea, Seth Gilbert, M. T. Hajiaghayi, Jason Hickey, Rebecca Hitchcock, Christos Kapoutsis, Dah-Yoh Lim, Emily Marcus, Vahab S. Mirrokni, Ashish Mishra, Claire Monteleoni, Benjamin Morse, Minh Nguyen, David Pritchard, April Rasala, George Savvides, Nitin Thaper, Emmanuele Viola, Grant Wang, and David Woodruff for scribing these lectures. Thanks to Dan Spielman for lecturing on March 13, 2002; and most significantly for providing the material for the course from prior semesters. Most of these notes are based on his notes, which can be found at `http://www-math.mit.edu/~spielman/AdvComplexity/2001/`.

These notes have been posted mostly unedited. There are probably many omissions, and also some redundancy (e.g. see Chapters 22–25). There is a possibility that these maybe revised in the future. Please check `http://theory.lcs.mit.edu/~madhu/ST02` for updates. Email me (`madhu@mit.edu`) if you have comments.

- Madhu Sudan
May 16, 2002

## Topics Covered

Lecture 01 (2/06): Introduction. Review of Complexity: Reductions, Time, Space, Nondeterminism.
Lecture 02 (2/11): Relativization. Baker-Gill-Solovay. Introduction to Alternation.
Lecture 03 (2/13): Alternation. ATIME vs. SPACE, ASPACE vs. TIME.
Lecture 04 (2/19): Power of Alternation: Fortnow's Time/Space lower bound. The Polynomial Hierarchy.
Lecture 05 (2/20): The PH Collapse hypothesis. Non-uniform computations. Karp-Lipton theorem.
Lecture 06 (2/25): Randomness. Randomized complexity classes. Sample problems in RP, RL.
Lecture 07 (2/27): Amplification of error in Randomness. BPP in P/poly. BPP in PH.
Lecture 08 (3/04): Circuit lower bounds: Parity is not in AC0.
Lecture 09 (3/06): SAT reduces probabilistically to Unique SAT. Counting classes.
Lecture 10 (3/11): Toda's theorem: PH is contained in #P.
Lecture 11 (3/13): Toda's theorem (contd.).
Lecture 12 (4/01): Proofs. Interactive Proofs, Arthur Merlin games.
Lecture 13 (4/03): IP in PSPACE; AM[poly] = IP[poly]; AM[k] = IP[k].
Lecture 14 (4/08): #P is in IP; Straightline programs of polynomials; IP = PSPACE.
Lecture 15 (4/10): Probabilistically checkable proofs (PCP).
Lecture 16 (4/17): NP in PCP(polylog,polylog).
Lecture 17 (4/22): NP in PCP(polylog,polylog).
Lecture 18 (4/24): Overview of the PCP Theorem. NP is in PCP[poly(n), O(1)].
Lecture 19 (4 / 29):Average case complexity.Hardness of the permanent on random instances. Distributed NP.
Lecture 20 (5/01): DNP. Avg-P. A problem complete for p-sampleable DNP problems.
Lecture 21 (5/06): A DNP-complete problem (contd.). Ajtai's worst-case to average case connection for lattice problems.
Lecture 22 (5/08): Quantum information, manipulation and computation. Quantum circuits and Turing machines.
Lecture 23 (5/13): Quantum Polynomial time (BQP). Simon's algorithm. Shor's algorithm.
Lecture 24 (5/15): Quantum computation wrap-up. Summary.

# Chapter 1

| | |
|---|---|
| **6.841/18.405J: Advanced Complexity Theory** | February 6, 2002 |
| Lecture 1 | |
| *Lecturer: Madhu Sudan* | *Scribe: April Rasala* |

## 1.1 Administrivia

The course web page is http://theory.lcs.mit.edu/~madhu/ST02/. If you have not received a class email from Professor Sudan, please email him at madhu@mit.edu to be added to the class email list. As mentioned on the course web page, grades for the course will be based on scribing 1-2 lectures, 4 problem sets and class participation. The only prerequisite for the course is 6.840 or an equivalent course at another institution.

## 1.2 Course Contents

Informally, complexity theory examines the effect of limiting resources on the ability to solve a computational problem. Typical resources to consider are time, space, and non-determinism. We'll also consider quantifier alternation and talk about the polynomial time hierarchy. In addition, we will look at randomness as a resource. In determining the power of randomization it is natural to consider related topics like derandomization and pseudorandomness. We will touch on these topics but anyone with a specific interest in derandomization and pseudorandomness might also want to check out a course at Harvard offered by Salil Vadhan (http://www.courses.fas.harvard.edu/~cs225/).

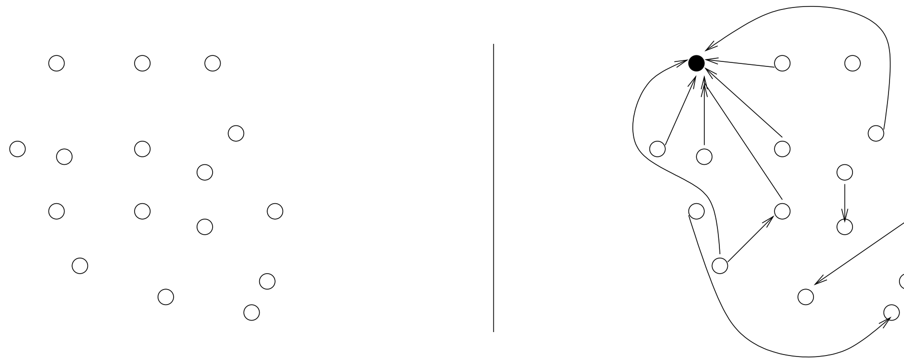
We will consider questions such as "What is a proof", "How does interaction affect computation?", "How does knowledge affect computation"? We will also briefly study topics related to circuit complexity and lower bounds for computation. Finally, if time permits, we will consider other models of computation such as quantum computing.

A more detailed description of possible topics and a lecture schedule is available on the course web page.

## 1.3 Classical Computational Complexity

Starting with work by Edmonds, Karp, Cook and Levin, complexity theory has traditionally looked at what sorts of things we can compute when we place a limit on a particular resource. For instance, we might restrict the amount of space used in the computation to be at most logarithmic in the input size. Other classic resources to consider are time and non-determinism.

Given a limit on some resource, such as time or space, we then try to collect a set of interesting problems. We attempt to link these problems together by understanding which problems are computationally no easier than other problems. We can imagine that we have a set of problems which are represented pictorially as a bunch of dots (figure on the left). If we can determine that problem $x$ is no harder than problem $y$, then we



draw an arrow from the dot representing $x$ to the dot representing $y$. Interesting problems, under the given resource restriction, are those problems with many arrows pointing toward them (for instance the dark dot in the picture on the right). We then define what aspects of this problem make it interesting and determine other problems that have a similar structure. In this way, we can show that this is not a unique interesting problem. This allows us to define a complexity class of problems.

Another question to ask is "what arrows can we rule out?" Hopefully we can eventually get a map of all computational problems and the complexities involved.

## 1.4 Reductions

A formal reduction allows us to turn this notion of the relative hardness of problems into a precise definition. As it turns out, there are already two different types of reductions and both are important for our purposes.

Let $O$ be an oracle (or subroutine) that solves problem $y$.

- Turing reduction: Given oracle $O$ solving $y$ with the given resource restrictions, there is an algorithm using $O$ that can solve problem $x$ with the same resource restrictions.
- Karp reduction: Given an oracle $O$ solving $y$, there is an algorithm that can solve problem $x$ by using $O$ once and simply returning the output.

Clearly both types of reductions would allow us to draw some connections between different problems. However, Karp reductions require that $x$ and $y$ be the same "type" of problem. In contrast, a Turing reduction can be used to show that a search problem (say finding a satisfying solution to an instance of SAT) is no harder than a decision problem (such as determining if a given SAT instance is satisfiable). Therefore Turing reductions allow us to focus on questions involving membership in a language.

Since Karp reductions are a restricted version of Turing reductions, the next question is why do we also need Karp reductions? One justification is that Turing reductions might be too powerful to really describe subtle relative difficulties of problems. For instance, suppose we want to determine if a 3-CNF formula $\phi$ is unsatisfiable. Using a Turing reduction we could show that this was "no harder" than determining if $\phi$ is satisfiable since we could simply call the oracle for 3SAT on $\phi$ and then negate the answer provided by the oracle. But this doesn't mesh well with our intuition that seems to say that there is something harder about showing that a boolean formula is unsatisfiable. If we use a Karp reduction then we are not allowed to negate the answer that the oracle produces.

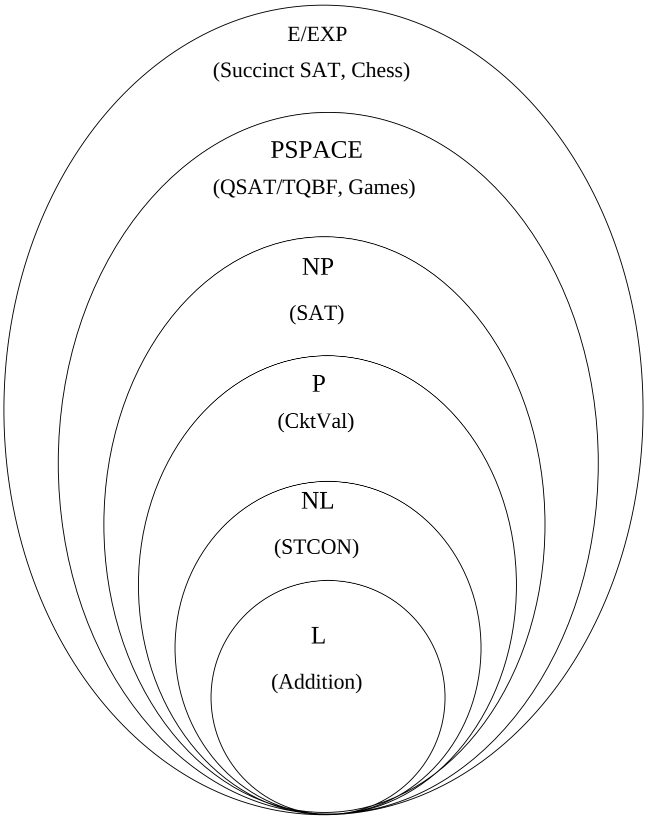
Thus the benefit of Turing reductions is that they allow us to restrict our attention to decision questions about membership in a language. Once we are concerned with only languages,Karp reductions provide more insight.

### 1.4.1 A Word on Notation

If there exists a polynomial time Turing machine $M$ with access to an oracle for problem $L_2$ that can solve problem $L_1$, then we write $L_1 \leq_T^p L_2$. To denote "algorithm $A$ using $O$" we use $A^O$.

## 1.5 Complexity Classes

The classical complexity classes are depicted in the following picture. Note that $\mathtt{E}$ refers to the class of




languages that can be decided in $\mathtt{TIME}(2^{O(n)})$ and $\mathtt{EXP}$ refers to the class of languages that can be decided in $\mathtt{TIME}(2^{n^{O(1)}})$.

What can we say about these classes? First, as shown in the picture, the set of problems that can be solved in $t(n)$ steps on a deterministic Turing machine is contained within the set of problems that can be solved in $t(n)$ time on a non-deterministic Turing machine. This set of problems is in turn contained within the set of problems that can be solved using only $t(n)$ space. Stated formally,

$$\mathtt{TIME}(t(n)) \subseteq \mathtt{NTIME}(t(n)) \subseteq \mathtt{SPACE}(t(n)).$$

Furthermore, $\mathtt{SPACE}(t(n)) \subseteq \mathtt{TIME}(2^{t(n)})$.

### 1.5.1 Hierarchy Theorems

Roughly the Time Hierarchy Theorem says that if $t_1(n) \ll t_2(n)$, then $\texttt{TIME}(t_1(n)) \subset \texttt{TIME}(t_2(n))$. In making this precise, we must consider two cases

**Theorem 1.1** *The set of languages that can be recognized by a Turing machine in* $\texttt{TIME}(t_1(n))$ *is strictly contained within the set of languages that can be recognized by a single-tape Turing machine in* $\texttt{TIME}(t_2(n))$ *if* $t_2(n) = \Omega(t_1^2(n))$.

**Theorem 1.2** *The set of languages that can be recognized by a Turing machine in* $\texttt{TIME}(t_1(n))$ *is strictly contained within the set of languages that can be recognized by a 2-tape Turing machine in* $\texttt{TIME}(t_2(n))$ *if* $t_2(n) = \Omega(t_1(n)\log t_1(n))$.

Like the Time Hierarchy Theorem, the Space Hierarchy Theorem allows us characterize some of the relationships between the complexity classes based on restricted space. However, it requires only that $t_2(n)$ be growing slightly faster than $t_1(n)$.

**Theorem 1.3** *The set of languages that can be recognized by a Turing machine in* $\texttt{SPACE}(t_1(n))$ *is strictly contained within the set of languages that can be recognized by a Turing machine in* $\texttt{SPACE}(t_2(n))$ *if* $t_1(n) = o(t_2(n))$. [1]

The above theorems apply to deterministic classes. The technique for proving them relies on a simple diagonalization argument. Roughly, suppose we want to show that there is some language $L$ that is in time $t_2(n)$ but not in time $t_1(n)$. We do this by specifically defining $L$ such that it is different from all languages that are decidable in time $t_1(n)$. In particular, suppose that Turing machine $M$ runs in time $t_1(n)$. If $M$ accepts its own encoding, then we define $L$ to reject the encoding of $M$. On the other hand, if $M$ rejects its own encoding, then we define $L$ to accept the encoding of $M$. Thus $L = \{\langle M\rangle \mid M \text{ runs in time } t_1(n) \text{ and } \langle M\rangle \notin L(M)\}$

Since every language decidable in time $t_1(n)$ has an associated Turing machine $M$ that recognizes it and since $L$ is designed to specifically differ from $M$ on the encoding of $M$ itself, $L$ must differ from every language decidable in time $t_1(n)$. The only trick is that we need to be sure that the Turing machine deciding $L$ has enough time to simulate $M$ on its own encoding. Thus the need for time $t_1^2(n)$ with a single tape Turing machine and time $t_1(n)\log t_1(n)$ if the machine deciding $L$ has two tapes.

Notice that the diagonalization argument required that we negate the decision of $M$ on its own encoding. This is only possible if $M$ is a deterministic Turing machine. However, a more difficult argument due to Cook shows that $\texttt{NTIME}(t_1(n)) \subset \texttt{NTIME}(t_2(n))$ as long as $t_2$ is slightly faster growing than $t_1$.

Finally, Blum's Speedup Theorem says that if we can solve a problem in time $cn$ for some constant $c$, then we can solve it in time $c'n$ for any constant $0 < c' < c$ by making the Turing machine "bigger" by a constant factor.

### 1.5.2 Open Questions

Probably the biggest open question is whether $P$ equals $NP$. We will generally assume $\texttt{P} \neq \texttt{NP}$. A stronger assertion would be that $\texttt{NP} \neq \texttt{CoNP}$. Weaker assumptions would be $\texttt{P} \neq \texttt{PSPACE}$, $\texttt{SAT} \notin \texttt{L}$ or $\texttt{NP} \neq \texttt{L}$, and $\texttt{SAT} \notin \texttt{TIME}(n \log^c n)$ for any constant $c$. In addition we can consider these relationships in pairs. For instance we know that either $\texttt{L} \neq \texttt{P}$ or $\texttt{P} \neq \texttt{PSPACE}$ by the Space Hierarchy Theorem.

### 1.5.3 Savitch's Theorem

We now consider the relationship between non-deterministic space and deterministic space. We will be proving Savitch's Theorem which says that something computable in non-deterministic space $t(n)$ is computable in deterministic space $t^2(n)$.

We begin by considering the space needed to compute the composition of two functions.

[1] Note: We assume that $t(n) = \Omega(\log n)$.

**Lemma 1.4** *If* $g : \{0,1\}^n \rightarrow \{0,1\}^n$*,* $f : \{0,1\}^n \rightarrow \{0,1\}^n$*,* $g \in \mathtt{SPACE}(s_1(n))$ *and* $f \in \mathtt{SPACE}(s_2(n))$*, then* $f \circ g \in \mathtt{SPACE}(s_1(n) + s_2(n))$

**Sketch of Proof** The obvious algorithm to compute $f(g(n))$ would be to compute $g(n)$ and then compute $f(g(n))$. However, we don't know if we can write down all of $g(n)$ in $\mathtt{SPACE}(s_2(n))$ (for example if $s_2(n) = \log n$). We get around this problem by computing the $i^{th}$ bit of $g(n)$ whenever we need it. Thus each time we need a bit of the input, we recompute $g(n)$ and only write down the bit that we need at that step of the computation. Thus we can reuse the same $s_1(n)$ space each time we compute another bit of $g(n)$. At the same time we can be using $s_2(n)$ space to compute $f(g(n))$. Thus the total space used is at most $s_1(n)+s_2(n)$ as claimed. ■

We are now ready to prove Savitch's Theorem.

**Theorem 1.5** *For any* $t(n) \geq \log n$*,* $\mathtt{SPACE}(t(n)) \subseteq \mathtt{SPACE}(t^2(n))$*.*

**Sketch of Proof** We first simplify the task by showing that we only need to consider $t(n) = \log n$. By a standard padding argument, one can see that if $t(n) = \Omega(\log n)$ then we can always pad the input with enough zeros so that the algorithm that ran in $t(n)$ space, now runs in $\log n'$ space where $n'$ is the length of the input after it has been padded. Thus proving this theorem for $t(n) = \log n$ proves it for all $t(n) \geq \log n$.

To further simplify things we consider an NL-complete problem, namely

$$\mathtt{STCON} = \{(G, s, t) \mid \text{there exists a path from } s \text{ to } t \text{ in directed graph } G.\}$$

(This problem is also known as PATH.) First, $G$ can be represented by the adjacency matrix $A$ where $A(i,i) = 1$ for all $0 \leq i \leq n-1$ and $A(i,j) = 1$ if and only if there is an edge from $i$ to $j$ in $G$. Given this representation of $G$, using boolean multiplication,

$$A^n = \underbrace{AAA\ldots A}_{n\ times}$$

will have the property that $A^n(i,j) = 1$ iff there is path of length at most $n$ from $i$ to $j$ in $G$ and $A^n(i,j) = 0$ otherwise. Thus if we can compute $A^n$ in $O(\log^2 n)$ space, then we will have shown that $\mathtt{STCON} \in \mathtt{SPACE}(\log^2 n)$ and hence all problems in $\mathtt{NL} \in \mathtt{SPACE}(\log^2 n)$.

Consider the space needed to compute $A^{2^L}$. If $L = 1$, then $A^{2^1} = A^2$ which can easily be done in $c \log n$ space for some constant $c$. Now we assume for the purpose of induction that we can compute $A^{2^{L-1}}$ in $c(L-1)\log n$ space. To show that $A^{2^L}$ can be computed in $cL\log n$ space we first compute $A^{2^{L-1}}$ in $c(L-1)\log n$ space. We can then square $A^{2^{L-1}}$ in $c\log n$ space. Since we used $c(L-1)\log n$ space to compute $A^{2^{L-1}}$ and $c\log n$ space to compute $A^{2^{L-1}} \cdot A^{2^{L-1}}$, by Lemma 1.4, we can compute $A^{2^L}$ in $c(L-1)\log n + c\log n = cL\log n$ space.

To complete our proof note that $A^n = A^{2^{\log n}}$ and therefore $A^n$ can be computed in $\log^2 n$ space. ■

# Chapter 2

| 6.841/18.405J: Advanced Complexity Theory | Feb. 11, 2002 |
|---|---|
| Lecture 2 | |
| *Lecturer: Madhu Sudan* | *Scribe: Vahab S. Mirrokni* |

In this lecture, we will see the diagonalization method as the most powerful method in complexity theory. We will discuss some results from this method e.g. Ladner's theorem; relativization, its relation with diagonalization method, and Baker-Gill-Solovay's theorem which implies that the diagonalization method does not help to prove NP $\neq$ P. Finally, we will have an introduction to alternation.

## 2.1 The Diagonalization Method

In order to find a relation between different complexity classes and to construct a better view of complexity, the only powerful known method is diagonalization. The main advantage of this method is to prove that some problems are not computable e.g. the undecidability of halting problem. Time and space hierarchy theorems are based on diagonalization. The other result from this method is the following: if P $\neq NP$, it is not possible to describe if a language is NPcomplete. Ladner's Theorem, which we will describe briefly, is another proof based on this method. It says that between any two different complexity classes there is an infinite hierarchy of complexity classes. For example, if P $\neq$ NP, it says that there are some problems that are niether in P nor in NPcomplete. Furthermore, there are infinite hierarchy of complexity classes between these P and NPcomplete. Ladner's theorem has an important effect on the philosophy of complexity theory. It means that in the case of P $\neq$ NP we can't classify all the problems in a finite number of complexity classes. Some examples of NP problems that are not known to be in P or NPcomplete are Primality testing, Graph isomorphism, and Factoring. Linear Programming was imagined to be in the middle of P and NPcomplete, but after some time it has been proved to be in P. Primality testing has a randomized polynomial time algorithm and it may be in P. On the other hand, it will be surprising if Factoring or Graph isomorphism is in P. More precisely, Ladner's theorem is the following:

**Theorem 2.1** *(Ladner, 1973) If* P $\neq$ NP*, then for all* $k \geq 1$*, there are languages* $L_1, \ldots, L_k \in$ NP *such that*

$$L <_{\mathrm{P}} L_1 <_{\mathrm{P}} \cdots <_{\mathrm{P}} L_k <_{\mathrm{P}} L'$$

*where* $L \in$ P *and* $L'$ *is* NPcomplete.

**Proof Idea** We just explain the intuition behind a weaker statement by taking $k = 1$: $\exists L \in$ NP such that $\mathrm{L} <_{\mathrm{P}} \mathrm{L}_1 <_{\mathrm{P}} \mathrm{L}'$. Let $n_1 = 1$ and $n_i = 2^{n_{i-1}}$, and let $L_1 = L$ for strings of length $[n_{i-1}, n_i)$ for odd $i$, and $L_1 = L'$ for strings of length $[n_{i-1}, n_i)$ for even $i$. Then probably $L_1 \notin$ P, because it has a large part of $L'$ which is NPcomplete. Also, probably $L_1$ is not NPcomplete, because otherwise if there is a polynomial reduction from $L'$ to $L$, for a string $s$ of length $n \in [n_{i-1}, n_i)$ for odd $i$ in $L'$, it can not be reduced to a string of greater length in $L_1$, because in that case the length of this string must be exponentially greater than $n$ and it contradicts the fact that the reduction is polynomial. On the other hand, if $s$ can be reduced to a string with smaller length, its length is exponentially smaller; thus, we can use an exponential time algorithm for the new problem, and It solves the first problem in $L'$ in polynomial time of the input size, and it contradicts the fact that there is no polynomial time algorithm for deciding $L'$. The flaw of this argument is that we are assuming that the hardness of the NPcomplete problem ($L'$) is uniformly distributed. Ladner's proof is based on picking a more careful choice of $n_i$'s so that the above statements are not probable but certain. He used diagonalization method to find these $n_i$'s. The idea is the following: list all polynomial reductions from $L'$ as $R_1, R_2, \ldots$ and all polynomial Turing machines $M_1, M_2, \ldots$, and then start from string of length 0, and include all strings in $L$ until it contradicts with $M_1$, then exclude all strings in $L$ until it contradicts the reduction $R_1$, and so on. Actually, this argument was not complete. ■

So far, we have seen power of diagonalization. The big question here is "can it resolve P vs. NP?" Baker-Gill-Solovay theorem answers this question negatively. To go through this theorem, first, we discuss relativization, i.e. stuff with oracles.

## 2.2 Relativization

**Definition 2.2** *Let $C$ be a complexity class of languages decidable with TM with some upper bounds on some resources, and let $A$ be an arbitrary language. Then $C^A$ is the set of languages accepted by oracle TM with access to oracle $A$ with the same or similar resource bound as a TM in $C$.*

However, in the above definition, the effect of $A$ on a TM in $C$ is not clear, and it is not an exact definition. It can be restated for the class P and NP more precisely.

**Definition 2.3** *$P^A$ is the set of all languages accepted by deterministic polynomial time oracle TM's with access to oracle for $A$.*

**Definition 2.4** $\mathrm{NP}^A$ *is the set of all languages accepted by non-deterministic polynomial time oracle TM's with access to oracle for $A$.*

By the Turing machine with access to oracle for $A$, we mean a TM which has the ordinary input, work, and output tapes and also the oracle tape. This new tape gets an input and gives the output 0 or 1 that indicates ifthe input is in $A$ or not.

**Proposition 2.5** *If diagonalization shows $C_1 \not\subset C_2$, then for every $A$, $C_1^A \not\subset C_2^A$. Roughly speaking, $C_1 \not\subset C_2$ relativizes.*

**Proof Idea** If we know $C_1 \not\subset C_2$ using diagonalization, then it means that we can find $m \in C_1$ as the universal TM for TM's in $C_2$ i.e. it can simulate every TM $N \in C_2$, then we simulate $N$ and negate the answer. Now, we can augment these machines into oracle machines and have similar results. In other words, we have TM $m^A \in C_1^A$ as the universal TM for TM's in $C_2^A$ i.e. it can simulate every TM $N^A \in C_2^A$; then we simulate $N^A$ and negate the answer. Thus, the similar argument shows that $C_1^A \not\subset C_2^A$. ■

Actually, there are many philosophies behind this proposition and these definitions. In the case of P and NP all above make sense, but somewhere else we can not use the same arguments.

**Theorem 2.6 (Baker-Gill-Solovay)** *There exist oracles $A$ and $B$ such that*

1. $\mathrm{P}^A = \mathrm{NP}^A$, *and*

*2.* $\mathrm{P}^B \neq \mathrm{NP}^B$.

**Proof** The first part is straightforward. The idea is to take some language that is sufficiently powerful e.g. PSPACE-complete. Let $A$ be TQBF. Clearly $\mathrm{P}^A \subseteq \mathrm{NP}^A$. For the other direction, as TQBF is PSPACE-complete, we have

$$\mathrm{NP}^A \subseteq \mathrm{NPSPACE} = \mathrm{PSPACE} \subseteq \mathrm{P}^A \subseteq \mathrm{NP}^A.$$

For the second part, the idea is that non-determinism gives us more powerful access to the oracle, allowing us to ask more questions than a deterministic TM can.

**Definition 2.7** *For any language $B$, let*

$$L(B) = \{x | \exists w : |x| = |w| \text{ and } B(w) = 1\}$$

One can easily observe that $\forall B : L(B) \in \mathrm{NP}^B$, because we can guess $w$ the same length as $x$, and see if $B(w) = 1$. We want to show that there is a $B$ for which $L(B) \notin \mathrm{P}^B$. In order to do this for all TM's, first we can examine how this is done for one. Let $M^? \in P^?$. We know that $M^?$ is a polynomail time oracle TM, its running time is a polynomial $p(n)$ and we can find $n$ such that $p(n) < 2^n$. We want to construct $B$ so that $M^B$ gives a wrong answer on some string $x$ of length $n$, say $x = 0^n$, provided $M^B$ runs in time $\leq 2^n - 1$. To do so, we start simulating $M^?$ for an input string $x$.

- The answer for any query of length $n$ from $M^?$ is 0.
- At the end, some $w \in \{0,1\}^n$ remians unasked, becuase $M^?$ runs in $p(n)$ and it can ask at most $2^n - 1$ strings of length $n$.
- Now, if $M^?$ says "yes, $x \in L^B$", then set $B(w) = 0$ for every $w \in \{0,1\}^n$.
- if $M^?$ says "no, $x \notin L^B$", then $B(w) = 1$.

In either case, $L(M^B) \neq L(B)$, because $M^B(0^n) \neq (0^n \in L(B))$ (This part of the proof is from [1]). Now we need an oracle $B$ such that for all polynomial time oracle TM's $M^?$, $L(M^B) \neq L(B)$. Suppose $M_1, M_2, \ldots$ are an enumeration of polynomial time oracle TM's. We construct a sequence $B_0 \subseteq B_1 \subseteq \ldots \subseteq B$, with lengths $n_1 < n_2 < \ldots$, such that

1. $M_i^{B_i}(0^{n_i}) \neq (0^{n_i} \in L(B_i))$.
2. $M_i^{B_i}(0^{n_i}) = M_i^B(0^{n_i})$.
3. $w \in L(B_i) : 0^{n_i} \in L(B_i) \Leftrightarrow 0^{n_i} \in L(B)$.

The construction algorithm is as follows:

1. First let $B_0 = \emptyset$.
2. For $i = 1, 2, 3, \ldots$ do
   (a) Choose $n_i$ such that $\forall j < i$, $M_j^{B_{i-1}}$ does not ask about any strings of length $n_i$ on input $0^{n_j}$. In addition, $M_i^{B_{i-1}}$ should run for fewer than $2^{n_i}$ steps on input $0^{n_i}$.
   (b) Simulate $M_i$ with oracle $B_{i-1}$ on input $0^{n_i}$. (At this point, $B_i$ has no strings of length $n_i$.) This will answer no to all queries of length $n_i$.
   (c) If $M_i^{B_{i-1}}$ accepts, set $B_i = B_{i-1}$. (So $M_i^{B_i}$ has accepted $0^{n_i}$, but $0^{n_i} \notin L_{B_i}$).
   If $M_i^{B_{i-1}}$ rejects, then find $x$ such that $|x| = n_i$ and $M_i^{B_{i-1}}$ on input $0^{n_i}$ did not ask $x$. Then set

$$B_i = B_{i-1} \cup \{x\}.$$

Set $B = \bigcup_{i=0}^{\infty} B_i$. ■

Again, notice that the above proof is only true for the special case (P vs. NP), and in general cases there are other issues that we won't discuss in this lecture.

One can similarly prove that there exists an oracle $B$ such that $\mathrm{NP}^B \neq \mathrm{coNP}^B$.

Now, consider this question: why do we use TQBF (a PSPACE-complete) problem for the first part of BGS theorem? Is an NP-complete problem sufficiently powerful? To answer this question, we should answer if $\mathrm{NP}^{\mathrm{NP}} = \mathrm{NP}$? The answer is No. Actually we get a new complexity class that is not known to be equal to NP. To argue about this question, first we define the following problem.

**Definition 2.8** MinDNF *is the language consisting of pairs $(\phi, k)$, where $\phi$ is a DNF formula and $k$ is an integer such that $\exists$ a DNF formula $\psi$ such that $|\psi| \leq k$ and $\psi$ is equivalent to $\phi$.*

**Proposition 2.9** MinDNF *is in* $\mathrm{NP}^{\mathrm{NP}}$.

**Proof** We can construct a non-deterministic oracle TM using SAT oracle to solve MinDNF.

- First guess $\psi$ of length $\leq k$.
- Ask SAT oracle if there exists an assignment $x$ such that $\psi(x) \neq \phi(x)$.
- Accept if oracle says No, otherwise reject.

This TM decides the MinDNF problem, and from the existence of this oracle TM, we can conclude $MinDNF \in \mathrm{NP}^{\mathrm{NP}}$. ■

Similarly, one can prove that co-NP$\subseteq \mathrm{NP}^{\mathrm{NP}}$. Notice that MinDNF has one $\exists$ quantifier. Let's investigate the case that there are more quantifiers.

## 2.3 Alternation

In the definition of non-deterministic TM, one way to think about it is a normal TM which has "existential states" ($\exists$). The converse of TM with Existential states is one with Universal ($\forall$) states. At each node, the machine can be thought of as spinning off two parallel actions and taking both paths at the same time. The machine accepts if all of its branches end in the accept state.

We can now consider machines with both Existential and Universal states. They are called Alternating Turing Machines (ATM).

**Definition 2.10** *The Alternating Turing Machine (ATM) is a TM with two special states $\exists$ and $\forall$. Computation accepts after entering*

- $\exists$*, if at least one of the outgoing edges lead to accept.*
- $\forall$*, if ALL of outgoing edges lead to accept.*

Figure 2.3 illustrates one ATM.

Similar to TM's, main resources for the ATM's are TIME and SPACE, but the other resource for them is Alternation i.e. # times we alternate from $\exists$ to $\forall$ and vice-versa. More precisely, for machine M and input $x$, TIME is the deepest path you can find in the tree. SPACE is maximum space on paths. Alternation is maximum # of alternations along the paths.

According to these resources, we can define general complexity classes as follows:

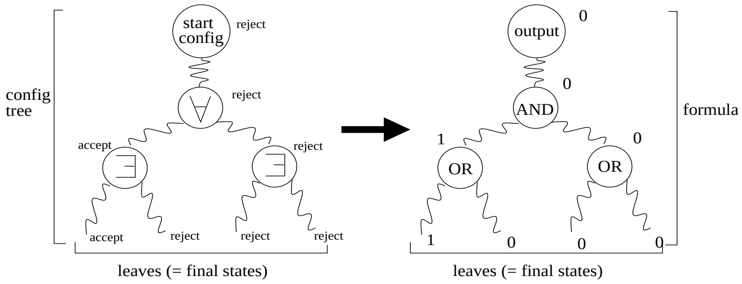



**Figure 2.1**: The computation tree of an ATM and corresponding circuit configuration

**Definition 2.11**

$$\mathrm{ATSP[a, t, s]} = \textit{class of all languages decided by ATM with } \mathrm{a(n)} \textit{ alternation, } \mathrm{t(n)} \textit{ time, and } \mathrm{s(n)} \textit{ space.}$$

*In particular, ATIME* $(t) = \cup_{\forall a,s} ATSP[a,t,s]$ *and ASPACE*$(s) = \cup_{\forall a,s} ATSP[a,t,s]$.

We have the following facts:

- ATIME(poly)=PSPACE
- ASPACE(poly)=EXPTIME

The interesting point is that alternation increases the computation power and also change TIME and SPACE in these cases.

**Definition 2.12**

$$\mathrm{NP} = \mathrm{ATSP[1, poly, poly]}, \textit{ with condition starting } \exists \textit{ state}$$

$$\mathrm{NP^{NP}} = \Sigma_2^{\mathrm{P}} = \mathrm{ATSP[2, poly, poly]}, \textit{ with condition starting } \exists \textit{ state}$$

$$\Sigma_i^{\mathrm{P}} = \mathrm{ATSP[i, poly, poly]}, \textit{ with condition starting } \exists \textit{ state}$$

*and similarly,*

$$\Pi_i^{\mathrm{P}} = \mathrm{ATSP[i, poly, poly]}, \textit{ with condition starting } \forall \textit{ state}$$

In the next lecture, we will study more about relations between these complexity classes.

# Bibliography

[1] Lecturer: Dan Spielman, Scribe: Edward Early, *6.841/18.405J Advanced Complexity Theory: Lecture 6*, **Feb. 27, 2001**.

# Chapter 3

| **6.841/18.405J: Advanced Complexity Theory** | February 13, 2002 |
|---|---|
| Lecture 3 | |
| *Lecturer: Madhu Sudan* | *Scribe: Emily Marcus* |

## 3.1 Alternation

Last time, we introduced the idea of an Alternating Turing Machine (ATM). This machine has two special "nodes" - the existential state ($\exists$) and the for-all state ($\forall$). If the machine entered a for-all node, it must accept on all of its outgoing branches of that node in order for the node to have accepted. If it enters an existential state, it must have at least one outgoing branch which has accepted.

The computation resources which are of interest in the ATM model are time, space, and total number of alternations. An alternation occurs when an ATM leaves the existential state and enters a for-all state or vice-versa.

Alternation is not actually an "implementable" or tangible property in the way that time and space are. In this respect it is more like non-determinism - you can't actually build a machine which uses alternation or non-determinism except by simulation. However, considering each of these resources gives rise to interesting classes of problems. In addition, analyzing alternation can give us more insight into the power of the more tangible properties of space and time.

The big question for today is how alternation relates to the resources we have already considered, namely time and space. Is an ATM more powerful than a TM?

In thinking about these questions, we will prove two main results, which are

- ATIME($t$) = SPACE(poly($t$))
- ASPACE($s$) = TIME($2^{O(s)}$)

## 3.2 Alternations vs Space

**Lemma 1:** ATIME(t) $\subseteq$ SPACE(O(t))

To see intuitively why this is the case, consider the tree representing the computation of some ATM A on some input. This tree will have nodes which are existential and nodes which are for-all. But in order to simulate this on a regular TM M, we only need to know where we are with respect to these nodes. The ordinary computations can be simulated by M trivially. Therefore, if M is augmented with an extra tape, which preserves a stack of the special nodes which have been encountered while simulating A, M can simulate A completely.

Because A runs in ATIME(t), no branch of its computation can be deeper than O(t). But M only needs to simulate M one branch at a time, so M only requires SPACE(O(t)).

Before proving the other half of the relationship between SPACE and ATIME, which requires constructing an alternating TM, it is useful to describe intuitively how programming such a machine works. Most of the programming is similar to programming an ordinary TM. However, the two special states are worth describing.

- The existential state effectively allows you to "guess" something. So, for example, if you "guess" subsequent bits, you can effectively pull out a number which satisfies some condition.
- The for-all state allows you to verify that something is true.

With this, we can prove the other half of the equality.

**Lemma 2:** SPACE($s$) $\subseteq$ ATIME($s^2$) (this should, by now, be unsurprising)

In order to see why this is true, we must first consider what a computation in SPACE($s$) looks like.

There are two assumptions we can make to simplify things slightly. First, if a computation has more than $2^s$ configurations, it has repeated at least one configuration and is therefore unlikely to terminate. Second, we can also assume without loss of generality that the final accepting configuration is unique.

Then, each configuration needs to contain

- the positions of all read/write heads
- the complete contents of the work tape
- the state that the machine is in

These somewhat general assumptions about the configurations are sufficient for this lemma. Later, when we are considering time instead of space, we will need to be much more precise about the configurations.

We also need to assume that s($|x|$) $\geq \log |x|$.

From here, the argument follows along lines similar to Savitch's theorem, with the $s^2$ term coming from the same type of argument.

So now the question we want to ask is whether our Turing Machine M can go from configuration $C_0$ to configuration $C_1$ in time $2^s$.

We'll define an alternating TM A which will, on input $< C_0, C_1, t >$, tell us whether M can go from $C_0$ to $C_1$ in time $t$.

1. Bit by bit, guess a configuration $C_3$ such that M can go from $C_0$ to $C_3$ in time $\frac{t}{2}$ and can go from $C_3$ to $C_1$ also in time $\frac{t}{2}$. This "guess" can be made by using the existential state, as described above.
2. Verify, using a for-all state, that M($C_0$, $C_3$, $\frac{t}{2}$) and M($C_3$, $C_1$, $\frac{t}{2}$) accept.

Inductively, this method takes space $s(\log t)$, and since initially $t$ is at most $2^s$, the whole thing requires at most $s^2$ space. Therefore SPACE($s$) $\subseteq$ TIME($s^2$).

In total now, we have shown that SPACE($s$) = TIME(poly($s$)).

## 3.3 Alternations vs Time

**Lemma 3:** ASPACE($s$) $\subseteq$ TIME($2^{O(s)}$)

This lemma is not easy to prove.

The basic idea is that we can create a machine which given a tree representing a computation, walks through the tree and figures out what happens at each node, and then propagates that information through the tree to figure out whether the entire computation accepts or not.

The problem with this simple idea, however, is that not all of the branches necessarily need to be resolved as being either accepting or rejecting in order for the whole computation to accept or reject.

So, in walking down the tree, if we are at a for-all node, we can check if that is accepting by checking every outgoing edge. If we are in an existential state, we need to check whether either of the edges leads to an accepting computation.

But what if there are repeated states on the way down?

The key observation is that if an existential node ends up accepting, it can't be accepting *because* of a branch below it which does not halt. So, if a configuration is ever repeated on a given branch, that branch can just be labelled as rejecting without any further inspection.

You can then work your way back up the tree, until the root node has a label; this label represents whether the entire computation accepts or rejects.

The tiem required to do all of this is $2^{O(s)}$ since examining each node requires a polynomial amount of time and there are $2^{O(s)}$ of them.

**Lemma 4:** TIME($2^{O(s)}$) $\subseteq$ ASPACE(O(s)).

Here we need to do a construction similar to that in Lemma 2, i.e. inspecting the sequence of configurations of a TIME($2^{O(s)}$) machine M. But in contrast to Lemma 2, the objective here is to use very little space in order to do this analysis. To allow this to be done in very little space, we'll set up the configurations so that they can be analyzed in a very localized manner.

For the $i^{th}$ entry of the configuration in the $t^{th}$ configuration of the computation, the following information will get stored.

- the contents of the $i^{th}$ cell of the tape
- whether or not the read-write head is in this position
- if the answer to the above is "yes" then the cell also contains the state of the machine.

Why is this careful encoding useful? Because then we can easily verify local consistency by comparing cell (t, i) to the three above it, namely (t-1, i-1), (t-1, i), and (t-1, i+1).

So how can we use this idea of local consistency to build an ATM which checks for global consistency based on this local consistency property? The idea of recursion, used in proving Lemma 2, doesn't help us here, because there is no longer enough available space to write down an entire configuration.

Instead, what we'll do is build an ATM M($t$, $i$, $\sigma$), which accepts its input if the original machine, M, on input x, could have $\sigma$ as the contents of cell $i$ in the configuration which arises after time $t$.

M($t$, $i$, $\sigma$) behaves as follows

1. Guess $r_1$, $r_2$, and $r_3$ as the contents of the three cells above cell ($t$, $i$).
2. Check that $r_1$, $r_2$, and $r_3$ are locally consistent with $\sigma$.
3. Verify, using a for-all, that M($t-1$, $i-1$, $r_1$), M($t-1$, $i$, $r_2$), and M($t-1$, $i+1$, $r_3$) all accept.

In doing this, there isn't any need for any extra space in order to keep track of things. You only need to write down enough information to spawn a subroutine call, and it isn't necessary to preserve the input. So while this is extremely slow, it requires very little space.

It is also necessary to check some of the boundary cases (for example, confirming that the final state is an acceptance state).

In total now, we have shown that

- ATIME($t$) = SPACE(poly($t$))
- ASPACE($s$) = TIME($2^{O(s)}$)

## 3.4 Philosophy of ATIME as a game

Philisophically, what is ATIME?

Fundamentally, ATIME problems represent a game. There are two players, one of them called "there exists" (referred to here as E), and the other one called "for all" (called A here).

Given a language L and an input x, E is trying to prove that x $\in$ L and A is trying to prove that x $\notin$ L.

So, as the game proceeds, the computation is left to follow its own course. But every time the computation enters an existential state, it stops and E is allowed to decide in what direction the computation should proceed. Similarly, A gets to choose every time a for-all state is entered.

At the end of the game, you are able to tell whether A or E has won, since watching this game will convince you that one or the other of them is correct. However, before the game finishes, poly-time computation is not enough to look into the future and and predict which of the players will win.

So, philisophically speaking, ATIME(poly) can simulate two-player games and the computation itself can be viewed as a two-player game.

But since PSPACE = ATIME(poly), this implies that there are a number of games which are PSPACE-complete. Examples include Checkers, Go, and Generalized Geography [1].

Why isn't chess a PSPACE-complete game? It has a somewhat obscure rule that if a board configuration is ever repeated, the game is automatically a draw. So you can't tell just by inspecting the current state of the computation whether the game is over or not. This makes it fundamentally different from the other games named above.

**Question:** Is it possible to maintain a counter which is incremented for every move, and then to declare the game to be a draw if the counter gets high enough that a configuration was necessarily repeated?

No, this isn't quite sufficient, because a repeated configuration can have different outcomes if one of the two players decides to act differently the second time around.

**Question:** Is this still an issue if the two players are playing optimally?

We're only looking at things locally at this point. This is an intricate technical issue which we'll try to cover in more detail later.

**Question:** (A third question which the scribe couldn't hear; sorry)

So in general, if a game allows you to verify local validity (i.e. watching the moves and knowing that each one is legal) and you are able to inspect only the current state and know whether someone has won, the game is likely to be PSPACE-complete.

What about a game like Mah-jong? This can be played as a one-player game. But because the intial board layout is randomized, the randomness allows it to have the same complexity as a game like Go or Checkers. The details of this will be explored later in the semester.

[1] A game in which players take turns naming geographical locations. Each location may be named at most once, and the first letter of the name of the location must be the last letter of the name of the previous location. A player wins when the other player cannot think of a location to name.

## 3.5 Philosophy of ATIME/ASPACE as conveying information

Supoose you are trying to decide whether to vote for candidate A or B in an election. There are varying levels to which the candidates might be allowed to try to convince you to vote for them. A few of these scenarios are

1. There is no campaigning allowed. You decide which candidate to vote for based on what you already know about.
2. Each candidate is allowed a single advertisement of fixed length, which they prepare independently.
3. The candidates are allowed to have a full-fledged debate, which voters can watch all of.

We can't really say which of these is "better", but we can try to understand the implications of each by figuring out what they allow you to compute.

Here a voter is considered to be a polynomial-time computation. So a voter can reach conclusions in P on his or her own. The candidates in tis scenario are analagous to the game players of before. The protocol for advertising is giving the voters more "knowledge" if it gives proofs that the voters are able to verify of a more complex language L.

In this context, we can analyze how much "knowledge" each of the above scenarios would convey to the voters.

1. In the case of no advertising, the voters can only recognize L $\in$ P. This is pretty much the trivial case.
2. In the case of a single advertisement, you can recognize L $\in$ NP $\bigcup$ coNP. More generally, if the candidates are allowed to produce $n$ ads and are allowed to see the opponents previous ads before making theirs, you get the additional resource of $n$ alternations.
3. Full-fledged debate with as much time as the candidates need gives all of PSPACE. The candidates are able to convince the voters that one of them is correct because they will present a complete argument which the voter has enough computational power to verify, even though the voter on his/her own would not have been able to come up with the argument.

So, from a computational point of view, debates are far more powerful than either of the other two methods of advertising because they are able to convince voters of much more complex languages.

## 3.6 TQBF : A PSPACE-complete problem

TQBF = totally quantified boolean formulae

If you have $n^2$ boolean variables, $x_{1,1} \cdots x_{n,n}$ and define $X_1 = x_{1,1}x_{1,2} \cdots x_{1,n}$ etc, then TQBF = $\{\phi \mid \phi(X_1, X_2, \cdots X_n)$ is a 3-cnf boolean formula such that $\exists X_1 \forall X_2 \exists X_3 \cdots Q_n X_n =$ true $\}$.

Note that because all such formulae are *totally* quantified (meaning they have no free variables), they are always either true or false.

Intuitively, there is a clear correspondence between TQBF and an alternating TM. The only difference is that TQBF formulae effectively defer all of the computation until the alternation is done. And, in fact, this whole class of problems can be further reduced to 3-SAT.

In summary, it is clear that TQBF $\in$ ATIME(poly). The fact that it is also a complete problem is not hard to verify.

# Chapter 4

| **6.841/18.405J: Advanced Complexity Theory** | February 19, 2002 |
|---|---|
| Lecture 4 | |
| *Lecturer: Madhu Sudan* | *Scribe: Ben Barden* |

In this lecture, we will discuss Fortnow's time/space lower bound for SAT, and see why alternation is considered a powerful tool for proof. We will also have an introduction to the polynomial hierarchy

## 4.1 Fortnow's Theorem

### 4.1.1 groundwork

Before we begin, we will be using a new class for this lecture, and pretty much for this lecture only: LIN, or near-linear time.

**Definition 4.1** *$LIN = \bigcup_c TIME(n \log^c n)$ for $c > n$*

Now, let us consider SAT. Specifically, let us consider an aspect of SAT that we know very little about, the lower bounds on its deterministic time/space requirements. We really don't know much. We do, however, have some pretty strongly held beliefs and intuitions about them.

**belief 1** *We believe that SAT is not solvable in deterministic polynomial time.*

**belief 2** *We believe that SAT is not solvable in deterministic logarithmic space.*

**belief 3** *We believe that SAT is not solvable in nearly linear time.*

Note that if belief number 1 is true, then it follows that the other two are true as well. So, beliefs 2 and 3 are, as such things go, fairly weak statements. Why, then, are they interesting? Well, in the case of 3, we note that non-deterministically, SAT is solvable in near-linear time, and, in fact, is complete for non-deterministic near-linear time. Therefore, if we could actually prove that it required more than near-linear time on a deterministic machine, that would be a clear and provable case of the power of non-determinism. We know, though, that alternation is powerful for small space computation, as. The other interesting thing is that, in spite of their weakness, neither 2 nor 3 has ever been proven. They are not entirely without support, though. In 1997, Fortnow managed to prove that at least one of the two had to be true, even if he could not be certain of which one.

**Theorem 4.2** *either SAT cannot be solved deterministically in logarithmic space or it cannot be solved deterministically in near-linear time.*

The proof depends heavily on the following fact

**Fact 4.3** *if $a' \leq a - 1$ and $t' \leq \frac{t}{logt}$ then ATIME$[a, t] \nsubseteq$ ATIME$[a', t']$*

which can be displayed through relatively simple diagonalization. Fortnow's shows us that if SAT is solvable in both deterministic log space and deterministic linear time, it is possible to show that ATIME$[a, t] \nsubseteq$ ATIME$[a', t']$ for those relative definitions of a, a', t, and t', thus causing a contradiction, and proving that at least one of the assumptions must be untrue. Now, we will not be giving the full proof. For simplicity's sake, we will only be showing that we can reach a contradiction if SAT $\in$ Time$(n \log n)$ and SAT $\in L$, and we are only giving the main steps, leaving many of the details of the proof as exercises

### 4.1.2 step 1

**Fact 4.4** *If a language $L$ is in NTIME(t), and $x$ is a string of length n, then $\exists$ an SAT instance $\phi$ of size $t(n) \log t(n)$ such that $x \in L$ iff $\phi \in$ SAT*

The proof here can either be taken as an exercise by the reader, or referenced in Cook's 1971 paper on the topic.

### 4.1.3 step 2

**Fact 4.5** *Given our assumptions, ATIME$[a, t]$ is contained in NTIME$[t(\log t)^{2a}]$*

proof as follows: To begin, we make the first of our two assumptions.

**Assumption 4.6** *SAT can be solved in deterministic, $n \log n$ time.*

One of the oddly more useful aspects of this statement for our purposes is the fact that if SAT can be solved deterministically in $n \log n$ time, then coSAT can also be solved deterministically in $n \log n$ time. Additionally, as this implies that SAT is in P, P=NP=coNP, and thus that coNP qualifies as a language $L$ in NTIME$(n \log n)$. From this we can know by Fact #4, that it can be simulated by a SAT instance of size $n \log n \log n \log n$, or $n(\log n)^2$.

### 4.1.4 step 3

**Fact 4.7** *Given our assumptions, NTime$[t(\log t)^{2a}]$ is in SPACE$[logt + a \log \log t]$*

and now, our second assumption.

**Assumption 4.8** *SAT can be solved in deterministic, logarithmic space.*

given this assumption, then we can know that NTime$[t(\log t)^{2a}]$ is in SPACE$[logt + a \log \log t]$.

### 4.1.5 step 4

**Fact 4.9** *SPACE$[s]$ is contained within ATISP$[b, 2^{\frac{s}{b}}, bs]$, which in turn is contained in ATIME$[b, 2^{\frac{s}{b}}]$*

### 4.1.6 results and considerations

Finally, we end, after a roundabout path and a bit of variable replacement, with the following conclusion

**Fact 4.10** *Given our assumptions, ATIME$[a, t] \subset$ NTIME$[t(\log t)^2 a] \subset$ SPACE$[\log t + a \log \log t] \subset$ ATISP$[b, 2^{\frac{\log t + a \log \log t}{b}}$ $ab \log \log t] \subset$ ATIME$[b, 2^{\frac{\log t + a \log \log t}{b}}]$*

note that this is for any b. If we choose $b = a - 1$, and clean it up a bit, we come up with

**Fact 4.11** *Given our assumptions, ATIME[a, t]* $\subset$ *ATIME[b,* $2^{\frac{\log t + a \log\log t}{a-1}}$*]*

now, $a \geq b$, and $2^{\log t - \log\log t}$ (another form of $\frac{t}{logt}$) is obviously bigger than $2^{\frac{\log t + a \log\log t}{a-1}}$ for any a greater than 2. Our very first fact said that ATIME$[a, t]$ cannot be contained within ATIME$[b, 2^{\frac{\log t + a \log\log t}{a-1}}]$, and now we discover that if our assumptions are both true, it must be. Therefore, at least one of our assumptions must be false. Note that here is a sterling example of the power of alternation as a tool for proof. The statement to be proven said nothing about alternation. The proof used it heavily and integrally.

## 4.2 The Polynomial Hierarchy

### 4.2.1 definitions and preliminary notes

**Definition 4.12** $\Sigma_i^p$ *is that class of languages that can be accepted by an ATM that starts in an existential state, has up to i alterations, and runs in polynomial time.*

**Definition 4.13** $\Pi_i^p$ *is that class of languages that can be accepted by an ATM that starts in a universal state, has up to i alterations, and runs in polynomial time.*

**Definition 4.14** *The Polynomial Hierarchy is the union over all constant i of all languages in* $\Sigma_i^p$*.*

Now, every language in any $\Sigma_i^p$ is contained easily within $\Pi_{i+1}^p$, as they can be simulated merely by ignoring the first universal state and alternating immediately to the first existential state. $\Pi_i^p$ is contained in $\Sigma_{i+1}^p$ for similar reasons. Thus, the polynomial hierarchy also contains all $\Pi_i^p$ for constant i. Also note that every language in $\Pi_i^p$ has its complement in $\Sigma_i^p$ and vice versa. By general notation, P is both $\Pi_0^p$ and $\Sigma_0^p$, thus making $\Sigma_1^p$ and $\Pi_1^p$ NP and coNP respectively.

### 4.2.2 iTQBF

As it turns out, the polynomial hierarchy has a number of complete problems, for every level. The canonical complete problem is iTQBF, the limited-alternation version of the PSPACE-complete problem TQBF. iTQBF is the collection of languages that take problems either of the form $\forall\{a_1, a_2, a_3..., a_i\}\exists b_1, b_2, b_3...b_i\forall...$ and a boolean formula, such that there are no more than i groups of no more than O(i) variables each, or of a nearly identical form, differing only in that the first quantifier is universal rather than existential. In either case, each language in the collection accepts if the formula can be true under the restrictions, and rejects otherwise. The proof that it is complete is fairly easy for polynomial time reductions, as each member of the collection is essentially the definition of its particular place in the hierarchy in language form. Let us consider a random language $A$ in $\Sigma_i^p$. Now, For each such $A$, there exists a language $B$ in $\Pi_{i-1}^p$, and some constant $c$ less than infinity, such that $x \in A$ iff $\exists y$ such that $\|y\| < \|x * c\|$ and $(x, y) \in B$. All of the existential decisions required before the first universal decision can easily be encoded on such a string. Similarly, for all languages $A'$ in $\Pi_i^p$ there exists a language $B'$, and some constant $c$ less than infinity, such that $x \in A'$ iff $\forall y$ such that $\|y\| < \|x * c\|$, $(x, y) \in B$. Again, any universal decisions could be easily coded on such a string. Given these two, we can derive that for any random language $A$ in, for example, $\Sigma_{i+1}^p$, that there exists a language $Q$ in either $\Sigma_1^p$ or $\Pi_1^p$ such that $x \in A$ iff $\exists y_1\forall y_2\exists y_3\forall y_4...y_i$ such that $\{x, y_1, y_2, y_3...y_i\} \in Q$ Note that Now, if we examine iTQBF again, we discover that the version for $\Sigma_1^p$ is equivalent to SAT, and that for $\Pi_1^p$ equivalent to coSAT. We can thus reduce any Q in either $\Sigma_1^p$ or $\Pi_1^p$ to the appropriate iTQBF, so long as the information on the input strings are available. The information on the input strings, in turn, can be described exactly as the universal and existential input groups on the higher levels of the proof.

### 4.2.3 minDNF

Some of the PH-complete problems, though, are not so easily proven. minDNF is a good example. MinDNF is a language that takes ($\phi$, k), where $\phi$ is a TM and $k$ is a number, and returns true if there exists a TM

$\psi$ such that $\|\psi\| < k$ (using some appropriate TM size rubric) and $\psi$ returns the same output as $\phi$ on all inputs. This language was so difficult to prove complete that, even though it was first described during the 70s in Meyer and Stockmeyer's original paper on PH, it wasn't until 2000 that it was proven $\Sigma_2^p$-complete buy Umans. We will not be sketching the proof here.

## 4.3 Coming Attractions

In our next lecture, we will examine the PH Collapse Hypothesis, a rather strong assumption that states that $\forall i, \Sigma_i^p \neq \Pi_i^p$. Among a number of other things, it implies both that P $\neq$ NP, and that NP $\neq$ coNP.

## 4.4 References

S. Cook. The complexity of theorem-proving procedures. In Proceedings of the 3rd ACM Symposium on computing, pages 151-158. ACM, New York, 1971

# Chapter 5


| **6.841/18.405J: Advanced Complexity Theory** | February 20, 2002 |
|---|---|
| Lecture 05 | |
| *Lecturer: Madhu Sudan* | *Scribe: Seth Gilbert* |


## 5.1 Collapse of the Polynomial Hierarchy

### What does the polynomial hierarchy look like?

It is an infinite tower of increasing complexity, with each level of the hierarchy nested inside the next level of the hierarchy.

- That is, $\Sigma_i^p \subseteq \Sigma_{i+1}^p$ and $\Sigma_i^p \subseteq \Pi_{i+1}^p$
- And similarly, $\Pi_i^p \subseteq \Sigma_{i+1}^p$ and $\Pi_i^p \subseteq \Pi_{i+1}^p$

### Each level of the hierarchy contains a complete problem.

In particular, for the $i^{th}$ level of the hierarchy, $i$-TQBF is complete. The subset of $i$-TQBF whose first quantifier is $\exists$, is complete for $\Sigma_i^p$, and the subset whose first quantifier is $\forall$ is complete for $\Pi_i^p$.

### Collapse

By *collapse of the polynomial hierarchy*, we mean that the entire hierarchy collapses to a finite level. That is, $\forall i > i_c$ $\Sigma_{i_c}^p = \Sigma_i^p$. (The same is true for $\Pi_i^p$.) Note that this does not imply anything about levels of the hierarchy below the collapse point.

**Theorem 5.1** *The polynomial hierarchy collapses if and only if the existential and the forall hierarchies are equal. That is:*

$$\Sigma_i^p = \Pi_i^p \iff \Sigma_i^p = \Sigma_j^p, \forall j > i$$

These versions of collapse may seem slightly different, as one is a horizontal collapse, and the other is a vertical collapse. But in fact they are equivalent.

**Proof** Once the incremental case comparing $i$ and $i+1$ is proved, it is clear by induction that it holds for the entire hierarchy (above that point) collapsing. That is, we only prove here that if $\Sigma_i^p = \Pi_i^p$, then $\Sigma_{i+1}^p$ (and the converse), and the rest follows by induction.

**Lemma 5.2** $\Pi_i^p \subseteq \Sigma_i^p \longrightarrow \Pi_i^p = \Sigma_i^p$

**Proof** If we can show inclusion in one direction, then the two classes are equal. This is true because the two classes are complementary.

$$\begin{array}{rcl} L \in \Pi_i^p & \iff & \bar{L} \in \Sigma_i^p \text{ (because they are complements)} \\ \forall L \in \Sigma_i^p & \longrightarrow & \bar{L} \in \Sigma_i^p \text{ (by the inclusion hypothesis)} \\ & \longrightarrow & \Sigma_i^p \subseteq \Pi_i^p \end{array}$$

■

First, assume that the polynomial hierarchy collapses at level $i$. That is, $\Sigma_i^p = \Sigma_{i+1}^p$. Since $\Pi_i^p = \Sigma_{i+1}^p$, this clearly implies that $\Sigma_i^p = \Pi_i^p$.

Next, assume $\Sigma_i^p = \Pi_i^p$. Start with $(i+1)$-TQBF, a complete problem for $\Sigma_{i+1}^p$. We will show how to solve this in $\Sigma_i^p$. Define:

$$L' = \{(\phi, x_1) | \forall x_2 \exists x_3 \cdots \mathrm{Q}_{i+1} x_{i+1}, \phi(x_1, x_2, x_3, \ldots, x_{i+1}) = 1\}$$

That is, $L'$ is the language of (i+1)-TQBF problems and $x_1$'s that allow the TQBF to be true. It is clear that $L' \subseteq \Pi_i^p \subseteq \Sigma_i^p$, as the first quantifier is a $\forall$, and there are only $i$ alternations.. Therefore, we can construct another language (say $L''$) that is in $\Sigma_i^p$. In particular:

$$L' \leq \{\psi | \exists y_1 \forall y_2 \cdots \mathrm{Q}_i y_i, \psi(y_1, y_2, \ldots, y_i)\}$$

This $\psi_{(\phi,x_1)}$ is constructed so that $(\phi, x) \in L' \iff \psi \in i-\mathrm{TQBF}$. The following is the procedure to decide the $(i+1)$-TQBF in $\Sigma_i^p$.

1. Guess $x_1$
2. Compute $\psi_{(\phi,x_1)}$
3. Guess $y_1$
4. ForAll $y_2$, Guess $y_3$ ...
5. Verify $\psi(y_1, \ldots, y_i) = 1$

Note that steps (1–3) involve only a single $\exists$ quantifier. That is, only a single alternation. And the rest of the $y_i$ require only $i-1$ alternations. Therefore we can solve $(i+1)$-TQBF with only $i$ alternations, that is, in $\Sigma_i^p$, which proves that $\Sigma_i^p = \Sigma_{i+1}^p$. ■

Note that this only holds for $i > 0$ This proof generally indicates that the ability to switch qualifiers is powerful and carries up the hierarchy, once equivalence is found at any level.

It hasn't been proven that the polynomial hierarchy does not collapse, but we generally believe that it does not collapse. We often therefore assume that the polynomial hierarchy does not collapse, and use this to prove other theorems. This is a strong assumption, however it allows us to reduce many of our conjectures to a single assumption. We will see some conjectures supported by this assumption later.

## 5.2 Circuit Complexity (Non-Uniform Complexity)

Circuit complexity is slightly different, but essentially the same technically as non-uniform complexity. It provides a new approach to the problem of $P \neq NP$.

## Circuits

Circuits are a model of computation with a fixed number of input bits. A Turing Machine works as follows:

1. Design algorithm.
2. Decide input to algorithm.

A circuit works as follows:

1. Decide length of input, $n$
2. Design algorithm to solve problems of length $n$
3. Decide input to algorithm (of length $n$)

## Formally

- A circuit is a directed, acyclic graph.
- 3 types of nodes:
  1. input: $x_1, \ldots, x_n$
  2. output: $o_1, \ldots, o_n$
  3. computation: OR, AND, NOT
- Each set of nodes is disjoint (for now)
- Fan-in: the number of edges going into a node

| Node Type | Fan-in |
|---|---|
| Input | 0 |
| Output | 1 |
| OR, AND | 2 |
| NOT | 1 |

- Fan-out: the number of edges leaving a node

| Node Type | Fan-out |
|---|---|
| Output | 0 |
| All Others | $\infty$ |

- To determine output:
  1. Topological sort (linear time)
  2. Calculate each level of the tree (linear time)
- Circuit computes some function $f : \{0,1\}^n \rightarrow \{0,1\}^n$.
- What resource to measure? Size of circuit = # of gates/nodes
  - Sometimes count # of wires/edges, but one measure is quadratic in the other
  - If you assume maximum fan-out of two, then at most twice as many wires as nodes
- Can compute function in (deterministic) time linear on # of nodes/edges

- Therefore, size is a good measure of how efficiently a circuit can compute.

What is interesting, however, is not determining how big a circuit is for a particular constant size. Instead, we want to define a set of functions, parameterized on the input size, $n$. We want to examine the size of the smallest circuit that can compute the function for that input. This gives us something interesting to measure as $n$ gets large. That is, we can study size as a function of $n$.

Consider a family of boolean functions $\{f : \{0,1\}^n \rightarrow \{0,1\}\}_{n=1}^{\infty}$. How does the smallest circuit computing $f_n$ grow with $n$?

This pins down the function once we set $n$. There is an absolute minimum for each $n$. This therefore provides a more concrete and combinatorial approach to examining complexity.

A common example of family circuits: $\phi_n$, the SAT family.

- $f_n$ takes as input CNF formula $\phi$ of length $n$
- $f_n = 1$ if and only if $\phi \in SAT$

If P=NP, the SAT family of circuits will be polynomial sized in $n$

- Draw TM tableau
- Create boolean formula to implement

So if we prove that $f_n$ has no polynomial sized circuit , then $NP \neq P$. Unfortunately, we don't know how to do this.

## Non-Uniform Complexity

- Modify/enhance TM so that we only worry about length $n$ inputs
- TM may work differently on different length inputs
- TM works as follows:
  1. Fix machine M
  2. Given length $n$
  3. Determine advice $a1, \ldots, a_n$
     - $a_i$ used to decide language
     - binary strings on $\{0,1\}^{p(n)}$
     - length of advice polynomial in $n$
  4. Given input $x \in \{0,1\}^n$
  5. Run $M(x, a_n)$ to decide if $x \in L$ or not.

**Definition 5.3** *P/Poly* $= \{L|$

- $\exists M \in P$
- *polynomial* $p(.)$
- $\{a_1, \ldots, a_n\}$ *s.t.* $|a_i| = p(i)$

*such that* $\forall x \in \{0,1\}^*, x \in L \iff M(x, a_{|x|})$ *accepts*$\}$

This provides a concrete way of asking how to use input length without defining a particular model of computation (i.e. circuits).

**Lemma 5.4** *P/Poly = class of functions that have polynomial sized circuits*

**Proof** If there is a known polynomial circuit for a given problem, then design $M$ to calculate the value of a circuit, and have the advice strings be the circuits for various sized inputs. $M$ using the advice can then clearly decide the output. Conversely, if we have a TM that can use advice to solve the problem, create a circuit to simulate the computation on the TM and build the advice into the circuit.

**Lemma 5.5** *Any problem in P has polynomial-sized circuits*

**Proof** Take language $L \in P$ decided by algorithm $A$. The goal is to find $C_n = A(x)$, where $|x| = n$. Construct a tableau for the TM's computation. It is of size $p(n) \times p(n)$. For every cell, for every $\sigma$ contents of the cell, let $x_{c,\sigma} = 1$ if (and only if) the cell has a $\sigma$ on it.

Compute $x_{c,\sigma}$ based on $x_{c_{i-1},\sigma_1}, x_{c_i,\sigma_2}, x_{c_{i+1},\sigma_3} \forall \sigma$. There is clearly a small circuit that will encode this piece of the TM computation. In this way all the variables can be determined by the circuit. If $x_{c_{final},\sigma_{accept}} = 1$, accept, else reject.

This circuit is at most a polynomial size larger than the TM; it blows up no more than $p^2$. (And, as before, Cooke was able to reduce this to something like $n \log n$. ■

Using this, we can easily see that it is possible to build a polynomial sized circuit that solves the problem using advice. ■

### Question: Is P/Poly more powerful than P?

- $BPP \subseteq P/Poly$
- Every unary language (where all strings are of the form $1^n$) is in P/Poly
  - Even undecidable unary languages in P/Poly
  - There is only one string of a given length, so hardwire the advice string
  - There is one advice string per input string, so the advice string indicates whether the appropriate unary string is in the language
- Every unary language is *not* in P. (Some unary language are undecidable.)
- P/Poly is not contained in P, NP, PSPACE
- P/Poly includes undecidable languages
- Conclusion: P/Poly is more powerful than P??

### Question: Is $NP \subseteq$ P/Poly?

Can you give polynomial advice to solve SAT?
Belief: $NP \nsubseteq P/Poly$ We want to relate all our beliefs to a single belief: the polynomial hierarchy collapse.

## 5.3 Karp-Lipton Theorem

**Theorem 5.6** *If NP $\subseteq$ P/Poly, then the Polynomial Hierarchy collapses.*

**Proof** For the sake of contradiction, suppose NP $\subseteq$ P/Poly. This implies that that we have $M, P, (a_1, \ldots, a_n)$ such that:

$$M(\phi, a_{|\sigma|}) = 1 \iff \phi \in SAT$$

Goal: Show $\Sigma_{i+1}^p \subseteq \Sigma_i^p$. We need a Turing Machine with $i$ alternations. We can compress $M$ and $P$, but we cannot compress all $a_n$ since there are infinitely many advice strings. So we need to figure out how to generate/encode an infinite sequence of advice strings.
Given $\phi \in (i+1)-$TQBF, need to compute right $a_n$ for that $\phi$.

**Definition 5.7** *$a_n$ is* GOOD *if it functions as right advice.*

As long as it causes $M$ to compute correctly, it is good, regardless of whether it is the same as the original $a_n$. That is, $M(\phi, a_n) = 1 \iff \phi \in SAT$, for all $\phi$ of length $n$. We can decide this property efficiently: in the 2nd level of the polynomial hierarchy.

**Lemma 5.8** *$a_n \in$ GOOD can be decided in $\Pi_2^p$.*

**Proof** We know how to check all $\phi$ of length $n$ using a *forall* quantifier (that is, it is in co-NP).

$$\forall \phi \text{ of length } n, M(\phi, a_n) = \text{“}\phi \in SAT\text{”}$$

But we know how to check if $\phi \in SAT$ using a single existential quantifier. We just ask an NP oracle. And we already know that co-NP$^{NP} = \Pi_2^p$. ■

Further, we can then find $a_n$ in $\Sigma_3^P$, since we can use the first existential state to guess the advice string, and then use the remaining two levels to verify.

**Lemma 5.9** *If $a_n \in$ GOOD is in $\Pi_{i-1}^p$, then $\Sigma_{i+1}^p \subseteq \Sigma_i^p$.*

**Proof** Let $i$ be even. Take an example of $(i+1)-$TQBF. We want to decide:

$$\{\psi | \exists x_1 \forall x_2 \cdots Q_{i+1} x_{i+1}, \psi(x_1, x_2, \ldots, x_{i+1}\}$$

Given a set of $x_1, \ldots, x_n$, define $\phi$ as a function of the last variable of $\psi$. That is:

$$\phi(y) = \psi(x_1, x_2, \ldots, x_i, y)$$

In order to decide whether $\phi \in SAT$, we can use the P/Poly machine M, along with the advice $a_{|\phi|}$. Given $a_n \in$ GOOD, we can determine whether $\phi \in SAT$ in deterministic polynomial time.

How do we determine $a_n$? By the lemma above, we can guess and verify $a_n$ is three alternations. However, in parallel, we also need to guess and verify $x_1, \ldots, x_n$ using $n$ alternations.

The following is a $\Sigma_i^p$ machine for deciding if $\psi \in (i+1)-$TQBF.

1. Guess $a_n$
2. Guess $x_1$
3. ForAll: $a_n$ is GOOD
4. ForAll: $x_2$
5. Guess $x_3$
6. Repeat alternating ForAll and Guess
7. ForAll: $x_i$

8. Calculate $\phi$, given $x_1, \dots, x_i$ guessed above

9. Calculate in deterministic polynomial time $M(\phi, a_n)$ and accept if and only if $\phi \in SAT$.

Notice that this machine uses only $i$ alternations. Steps 1–2 use a single $\exists$, and steps 3–4 use a single $\forall$. The guessing and verifying of $a_n$ happen in parallel with guessing and verifying $x_1, \dots, x_i$. By using the P/Poly deterministic machine to solve the last SAT problem, we are avoiding one level of the hierarchy. As a result, this $\Sigma_i^p$ machine can solve $(i+1)-$TQBF, showing that the polynomial hierarchy collapses. ■

■

# Chapter 6

| **6.841/18.405J: Advanced Complexity Theory** | February 25, 2001 |
|---|---|
| Lecture 06 | |
| *Lecturer: Madhu Sudan* | *Scribe: Mike Alekhnovich* |

**PS submission suggestions:**

- avoid if possible using PDF files. Tex or PS are just fine...

- try to choose a name of the file that can identify yourself. Certainly names like "ps1.ps" is not a great idea at all...

**Brief content:**

Today we will learn what randomized computation is, will define eight new complexity classes, and see several interesting randomized algorithms.

## 6.1 Randomized Computation

Why do we care about randomized complexity? Well, mostly physisists. It is their beleif that nature is random and the system always evaluates to increase the entropy. How does this assumption alter our notion of "physically realizable computing"? As usual, in order to study this resource formally we define a new complexity class(es) and then try to relate them to earlier defined ones. It doesn't really matter if they are physically realizable, as long as it is mathematically definable. However, it appears that this particular one is realizable, which is very exciting.

### 6.1.1 How to define?

In parallel to non-determinism there are two ways to introduce the resource formally:

- As a computation on Turing machine with a special state $R$ which picks randomly between two other states, and jumps to one of them.
- As a computation on two input turing machine $M(x, y)$, where $x$ is our "real" input and $y$ is a "random" string.

We will give the second variant of the definition. Let $L$ be a language and and $M$ be two input turing machine. Following the terminology of logical proof theory we can define *completeness* and *soundness* of $M$ recognizing $L$:

- **Completeness**. This is the minimal probability of $M$ accepting a "true" word that belongs to the language. Formally
$$c = \inf_{x \in L} \Pr_y[M(x,y) \text{ accepts}]$$
- **Soundness**. This is the maximal probability that $M$ accepts a "wrong" word that is not in the language. Formally
$$s = \sup_{x \notin L} \Pr_y[M(x,y) \text{ accepts}]$$

The case $s = 0$ is reffered as perfect soundness (no false positives), $c = 1$ as perfect completeness (no false negatives). Intuitively one can suggest 4 essentially different patterns of $c$ and $s$ that make the language $L$ randomly tractable:

- $c > s$,

- $c > 0$, $s = 0$

- $c = 1$, $s < 1$

- $c = 1$, $s = 0$

They exactly correspond to the four randomized complexity classes (more precisely to the eight classes because one can bound the time and the space of the computaion).

### 6.1.2 Time bounded computation

In this case we require our machine $M(\cdot,\cdot)$ run in time polynomial in $x$ (clearly this implies that we need only polynomial number of random bits, so we can assume that $|y| < |x|^{O(1)}$). According to the four patterns of $s$ and $c$ the following classes can be defined:

- $BPP$ ($c = \frac{2}{3}$, $s = \frac{1}{3}$): $BPP$ (stands for bounded error probability polynomial) is a most general randomized class. We admit our machine do both types of error with bounded probability.
- $RP$ ($c = \frac{2}{3}$, $s = 0$): we admit only one-sided error of rejecting the "true" world.
- *coRP* ($c = 1$, $s = \frac{2}{3}$): the class dual to $RP$.
- $ZPP$ ($c = 1$, $s = \frac{2}{3}$): $ZPP$ (stands for zero-equal probability) is a class of randomized computations with expected polynomial time.

1). These classes are robust with respect to $c$ and $s$. As we (probably) will show in the sequel, the particular choice of these constants doesn't play any role, we can choose them arbitrarily (as long as we preserve conditions $c = 1$, $s = 0$).
2). It is believed that classes $P$ and $BPP$ are very close to each other, it is even conjectured (although we are still far from proving it) that $P = BPP$. Sadly, we don't have a complete problem for $BPP$ (in particular the natural idea with universal random machine doesn't seem to work). However this class contains a lot of other interesting problems.
3). How does $ZPP$ work? Notice that in this case we don't admit any error, all we require of the machine is to run on average in polynomial time. It is an exercise to show that $ZPP = RP \cap coRP$. A natural example of a problem in $ZPP$ is primality testing. It is not hard to show that $PRIMES \in coRP$. By a more sophisticated algorithm due to Adleman and Huang, $PRIMES \in RP$ as well. If we run both routins in parallel then we get a $ZPP$ primality testing algorithm.

### 6.1.3 Space bounded computation

So far we have seen polynomially bounded randomized computation. What is our machine $M(\cdot,\cdot)$ is restricted to run in $LOG-SPACE$ instead? Well, there are four classes: $BPL$, $RL$, $coRL$ and $ZPL$. They are defined analogously to time bounded classes, there are two catches though.

**Catch 1:** we should require $M$ run in polynomial time.

**Catch 2:** in two input model we should provide only one-way access to the second input (the machine cannot return to previously given random bits).

### 6.1.4 Relationships with other classes

The following inclusions that relates randomized complexity with other classes are known:

- $RP \subseteq NP$
- $coRP \subseteq coNP$
- $BPP \subseteq PH$
- $BPP \subseteq P/poly$

It is open if $BPP \subseteq NP$, however $BPP$ is contained in the higher levels of polynomial hierarchy. Later we will prove some of these inclusions. Let us now turn to some interesting problems in the defined classes.

### 6.1.5 $RP$ and Polynomial Identity Testing

*Polynomial* over integers is a finite sum

$$P = \sum_{c_{i_1 i_2 \ldots i_n}} c_{i_1 i_2 \ldots i_n} \cdot x_1^{i_1} \cdot x_2^{i_2} \cdot \ldots \cdot x_n^{i_n}.$$

We can define *degree* of a single variable $\deg_{x_k}(P)$ as a maximal $i$ for which $P$ contains a term $(x_k^i \cdot \ldots)$. The total degree $\deg(P)$ of a polynomial $P$ is the maximal sum $i_1 + i_2 + \ldots + i_n$ over all non-zero coefficients $c_{i_1 i_2 \ldots i_n}$.

**Example.** $P(x_1, x_2) = x_1^2 x_2^3 + 3x_1^3 + 4x_2^4$.

For this polynomial, $\deg_{x_1}(P) = 3$, $\deg_{x_2}(P) = 4$ and $\deg(P) = 5$.

Now we define *Polynomial Identity (to zero) Testing* problem:
**Instance:** *polynomial* $A : \mathbf{Z}^n \to \mathbf{Z}$ *represented as an oracle, an integer* $d = \deg(A)$.
**Question:** $\exists a_1, \ldots, a_n \; A(a_1, \ldots, a_n) \neq 0$?

Before we will give a randomized algorithm for this problem let us discuss some applications and consequences. As posed, the problem naturally contains in $NP^A$. In fact, one can show that it doesn't belong to $P^A$, thus this oracle separates $P^A$ and $RP^A$.

**Example of an application.** *Let* $M_{ij} = \sum c_k x_k$ *be a matrix that contains polynomials as elements. Let* $det(M)$ *be its determinant calculated by standard rules (thus* $det(M)$ *is a polynomial over* $x$*). The question is* $det(M) = 0$*?*

We can consider $det(M)$ as an oracle, indeed for any specific value of $x$ we can calculate $det(M)(x)$ by the Gaussian method. Together with an algorithm for polynomial identity this gives us a randomized algorithm that solves the problem.

**Theorem.** *Polynomial identity testing $\in RP$.*
**Proof.** We run the following algorithm:

**1).** *Set $m = 3d$*

**2).** *Pick $a_1, ..., a_n$ independently from $\{1, ..., m\}$.*

**3).** *If $A(a_1, ..., a_n) \neq 0$ then accept*
*else reject*

The question is why does this algorithm work? The soundness is trivially 0 (if there is no point for which $P \neq 0$, we will never accept.) The completeness is less trivial and appeals to the following useful lemma:

**Lemma (Schwartz).** *Let $P(x_1, ..., x_n)$ be a non-zero polynomial over an arbitrary field and $S$ be a finite subset of domain of $P$. Then*

$$\Pr_{a_1,...,a_n \in S}[P(a_1, ..., a_n) = 0] \leq \frac{d}{|S|}.$$

**Proof** by induction. If $n = 1$ then the statement of the lemma is just the classical theorem that degree $d$ non-zero polynomial cannot have more than $d$ roots. Assume that $n > 1$ and $P(x_1, ..., x_n)$ is a polynomial of degree $d$. Consider a variable $x_n$, let $d_n = \deg_{x_n}(P)$. Let us represent $P$ in the form:

$$P(x_1, ..., x_n) = Q(x_1, ..., x_{n-1})x_n^{d_n} + R(x_1, ..., x_n)$$

so that $\deg(Q) \leq d - d_n$ and $\deg_{x_n}(R) < d_n$.

Let $a_1, ..., a_n$ be a randomly chosen subset of $S$. Let $g(x_n) = P(a_1, a_2, ..., a_{n_1}, x_n)$ be the univariate polynomial that results after we replace each $x_1, ..., x_{n-1}$ with $a_1, ..., a_{n-1}$ in $P$. Clearly $\deg g \leq d_n$. Define two "bad" events:

$$E_1 : \; Q(a_1, ..., a_{n-1}) = 0$$
$$E_2 : \; \bar{E}_1 \wedge g(x_n) = 0$$

Obviously if neither of them happens then $P(a_1, ..., a_n) \neq 0$. Thus we need to estimate the probability of the event $E_1 \vee E_2$. By the induction hypothesis, $\Pr[E_1] \leq \frac{d-d_n}{|S|}$. If the event $E_1$ does not occure then $g$ can have at most $d_n$ roots, thus we have $\Pr[E_2] \leq \frac{d_n}{|S|}$. Finally

$$\Pr[E_1 \vee E_2] \leq \Pr[E_1] + \Pr[E_2] \leq \frac{d}{|S|},$$

which finishes the proof.

Now, applying the lemma we get

$$c = \Pr_{a_1,...,a_n \in [1..m]}[A(a_1, ..., a_n) \neq 0] \geq \frac{2}{3},$$

and the theorem follows.

### 6.1.6 $RL$ and Undirected S-T Connectivity

At the end we will briefly describe an example of a language in $RL$. Consider the following

*Undirected S-T connectivity* (USTCON) problem:
**Instance:** *Undirected graph $G$.*
**Question:** *Is there a path connecting $s$ to $t$?*

We present an algorithm based on the random walk:

*For* $n^4$ *steps do*

*Pick a random neighbor* $v$ *of* $u$

$u := v$

*If* $u = t$ *then stop and accept*

The algorithm runs random walk over the matrix of $G$ for a while. If it reaches $t$, it is clearly connected. If after a great deal of time it has not succeeded, it rejects.

Despite the innocent simplicity of the algorithm, the analysis is by far not that simple. In fact, for a long time people had been thinking that USTCON is as complex as STCON, till the following beautiful result was proved by Aleliunas, Karp, Lipton, Lovász and Rackoff: For every connected graph with high probability the random walk will visit each node in $O(n^3)$ steps.

# Chapter 7

| **6.841/18.405J: Advanced Complexity Theory** | February 27, 2002 |
|---|---|
| Lecture 7 | |
| *Lecturer: Madhu Sudan* | *Scribe: Deniss Čebikins* |

### 1. Review of definitions

Recall the definitions of BPP and RP.

A language $L$ is in BPP if and only if there exists a polynomial time Turing machine $M$ and a polynomial $p$ such that

$$\Pr_y[M(x,y) \text{ accepts}] \geq c = \frac{2}{3} \text{ for all } x \in \mathrm{L}$$

$$\Pr_y[M(x,y) \text{ accepts}] \leq s = \frac{1}{3} \text{ for all } x \notin L$$

where $|y| = p(|x|)$.

A language $L$ is in RP if and only if there exists a polynomial time Turing machine $M$ and a polynomial $p$ such that

$$\Pr_y[M(x,y) \text{ accepts}] \geq c = \frac{2}{3} \text{ for all } x \in L$$

$$\Pr_y[M(x,y) \text{ accepts}] = 0 \text{ for all } x \notin L$$

where $|y| = p(|x|)$.

Our goal is to show that the above definitions do not really depend on the constants $\frac{1}{3}$ and $\frac{2}{3}$. In fact, in the definition of BPP these constants can be replaced by any pair of constants $(c, s)$ provided that $0 < s < c < 1$. Furthermore, we will show that instead of constants we can use functions $c(n) = 1 - e^{-q(n)}$ and $s(n) = e^{-q(n)}$, where $n = |x|$, and $q$ is a polynomial.

### 2. Example: amplification of RP

We start with an example of amplification that shows that a language with the following properties is in RP.

Let $L$ be a language for which there exists a polynomial time Turing machine $M$ and a polynomial $p$ such that

$$\Pr_y[M(x,y) \text{ accepts}] \geq \frac{1}{n^2} \text{ for all strings } x \in L \text{ of length } n$$

$$\Pr_y[M(x,y) \text{ accepts}] = 0 \text{ for all } x \notin L$$

where $|y| = p(|x|)$. To show that $L \in \text{RP}$ we will construct another Turing machine $M'$ that satisfies the conditions of the definition of RP stated above.

Lemma. There exists a polynomial $m(n)$ such that $\left(1 - \frac{1}{n^2}\right)^{m(n)} \leq \frac{1}{3}$.

*Proof.* Put $m(n) = 2n^2$. Then $\left(1 - \frac{1}{n^2}\right)^{m(n)} \approx e^{-\frac{m(n)}{n^2}} = e^{-2} = 0.1353... < \frac{1}{3}$.

Let $M'(\cdot,\cdot)$ be a Turing machine with the following algorithm:

- $M'$ takes as input a string $x$ of length $n$ and a sequence $\bar{y} = \{y_1, y_2, \ldots, y_{m(n)}\}$ of strings, where $|y_i| = p(n)$ and $m(n)$ is a polynomial such that $\left(1 - \frac{1}{n^2}\right)^{m(n)} \leq \frac{1}{3}$.
- For each $i$ between 1 and $m(n)$ machine $M'$ simulates $M$ on input $(x, y_i)$.
- $M'$ accepts if and only if $M(x, y_i)$ has accepted for some $i$.

Clearly, $M'$ is a polynomial time Turing machine, and $\bar{y}$ has polynomial length. Let us analyze the probability of "success" of $M'$. First of all, it is obvious that

$$\Pr_{\bar{y}}[M'(x,\bar{y}) \text{ accepts}] = 0 \text{ for all } x \notin L$$

since $M(x, y_i)$ will never accept if $x \notin L$. If $x \in L$, and $y_1$, $y_2$, ..., are chosen independently at random, then for each $i$ the probability that $M(x, y_i)$ rejects is at most $1 - \frac{1}{n^2}$, therefore the probability that $M(x, y_i)$ rejects for all $i$ is at most $\left(1 - \frac{1}{n^2}\right)^{m(n)}$, hence

$$\Pr_{\bar{y}}[M'(x,\bar{y}) \text{ accepts}] \geq 1 - \left(1 - \frac{1}{n^2}\right)^{m(n)} \geq \frac{2}{3}$$

It follows that $L \in \text{RP}$.

### 3. Amplification of BPP

In this section we will show that for every language $L$ in BPP and every polynomial $q$ there exists a polynomial time Turing machine $M'(\cdot,\cdot)$ and a polynomial $u$ such that

$$\Pr_{\bar{y}}[M'(x,\bar{y}) \text{ accepts}] \geq 1 - e^{-q(n)} \text{ for all strings } x \in L \text{ of length } n$$

$$\Pr_{\bar{y}}[M'(x,\bar{y}) \text{ accepts}] \leq e^{-q(n)} \text{ for all strings } x \notin L \text{ of length } n$$

where $|\bar{y}| = u(n)$.

Since $L \in \text{BPP}$, there exist a polynomial time Turing machine $M$ and a polynomial $p$ such that

$$\Pr_y[M(x,y) \text{ accepts}] \geq c \text{ for all } x \in L$$

$$\Pr_y[M(x,y) \text{ accepts}] \leq s \text{ for all } x \notin L$$

where $|y| = p(|x|)$. To make our result stronger we will assume that $c$ and $s$ are arbitrary constants satisfying $0 < s < c < 1$ rather than $\frac{2}{3}$ and $\frac{1}{3}$.

As in the previous section, the input to the new machine $M'$ will be a string $x$ of length $n$ and a sequence $\bar{y} = \{y_1, y_2, \ldots, y_{m(n)}\}$ of independently selected random strings such that $|y_i| = p(n)$ for all $i$. We will see how to choose the polynomial $m(n)$ later.

The algorithm of $M'$ will be simple: for each $i$, simulate $M$ on input $(x, y_i)$ and accept if the fraction of $i$'s in $\{1, 2, \ldots, m(n)\}$ for which $M(x, y_i)$ has accepted is at least $\frac{c+s}{2}$. (Thus, for example, if $s = 0.86$ and $c = 0.88$ then $M'$ will accept if $M(x, y_i)$ accepts for at least 87% of indices $i$ in the set $\{1, 2, \ldots, m(n)\}$.)

In our analysis of the probability of correctness of this algorithm we will use the following lemma.

Lemma (Chernoff bound). Let $D$ be a distribution on $\{0, 1\}$. Suppose that $x_1, x_2, \ldots, x_N$ are chosen independently from $D$. Let $\mu = \mathrm{E}_{x \in D}[x]$. Then for any $\lambda$ the following inequality holds:

$$\Pr\left[\left|\frac{\sum_{i=1}^{N} x_i}{N} - \mu\right| \geq \lambda\right] \leq e^{-\lambda^2 \cdot \frac{N}{2}}$$

We apply the lemma in the following way. For $1 \leq i \leq m(n)$, define

$$X_i = \begin{cases} 1 & \text{if } M(x, y_i) \text{ accepts} \\ 0 & \text{if } M(x, y_i) \text{ rejects} \end{cases}$$

First let us consider the case $x \in L$. Then $\mu = \frac{\mathrm{E}[X_i]}{m(n)} \geq c$. Put $\lambda = \frac{c-s}{2}$. Then

$$\Pr\left[\text{ the fraction of } i \text{ in } \{1, 2, \ldots, m(n)\} \text{ for which } M(x, y_i) \text{ accepts is smaller than } \tfrac{c+s}{2}\ \right] \leq$$

$$\leq \Pr\left[X_1 + X_2 + \cdots + X_{m(n)} \leq \left(\frac{c+s}{2}\right) \cdot m(n)\right]$$

$$\leq \Pr\left[\left|\frac{\sum_{i=1}^{m(n)} X_i}{m(n)} - \mu\right| \geq \frac{c-s}{2}\right] \leq e^{-\left(\frac{c-s}{2}\right)^2 \cdot \frac{m(n)}{2}}$$

since the distance from $\mu$ to $\left(\frac{c+s}{2}\right)$ is at least $\left(\frac{c-s}{2}\right)$. Therefore

$$\Pr_{\bar{y}}[M'(x, \bar{y}) \text{ accepts}] \geq 1 - e^{-\left(\frac{c-s}{2}\right)^2 \cdot \frac{m(n)}{2}}$$

Similarly one can show that if $x \notin L$ then

$$\Pr_{\bar{y}}[M'(x, \bar{y}) \text{ accepts}] \leq e^{-\left(\frac{c-s}{2}\right)^2 \cdot \frac{m(n)}{2}}$$

In order to finish the argument it remains to set $m(n) = \frac{2q(n)}{(c-s)^2}$.

Here is an application of this result:

Proposition (Adelman). $\mathrm{BPP} \subseteq \mathrm{P}/_{\mathrm{poly}}$.

*Proof.* Suppose that $L$ is in BPP. Then there exists a polynomial time randomized Turing machine $M'$ such that

$$x \in L \quad \Longrightarrow \quad M' \text{ accepts } x \text{ with probability } 1 - 2^{-(n+1)} \text{ or more}$$
$$x \notin L \quad \Longrightarrow \quad M' \text{ accepts } x \text{ with probability } 2^{-(n+1)} \text{ or less}$$

We claim that given $n$, there exists a string $y$ such that $M(x, y)$ accepts if and only if $x \in L$ for all $x \in \{0, 1\}^n$. (Hence $y$ is the advice to $M$ corresponding to inputs of length $n$.)

Let us call a string $y$ "bad" for $x \in \{0, 1\}^n$ if $x$ is in $L$ and $M(x, y)$ rejects, or else if $x$ is not in $L$ and $M(x, y)$ accepts. We will also say that $y$ is "good" for $x$ if it is not "bad" for $x$. For any fixed $x$ we have

$$\Pr_y[\ y \text{ is "bad" for } x\ ] \leq 2^{-(n+1)}$$

therefore

$$\Pr_y[\exists x \in \{0,1\}^n \mid y \text{ is ``bad'' for } x] \le \sum_{x \in \{0,1\}^n} \Pr_y[\ y \text{ is ``bad'' for } x\ ] \le 2^n \cdot 2^{-(n+1)} = \frac{1}{2}$$

so $\Pr_y[\ y$ is "good" for all $x\ ] \ge \frac{1}{2}$. The claim and hence the proposition follow.

### 4. BPP and the Polynomial Hierarchy

We will show in this section that BPP $\subseteq$ PH. In fact, we will prove that BPP $\subseteq \Sigma_2^{\mathrm{P}}$.

Let $L$ be a language in BPP. To show that $L \in \Sigma_2^{\mathrm{P}}$ we can represent the process of deciding whether $x$ is in $L$ as a two round debate, in which Player 1 tries to prove that $x \in L$, and Player 2 tries to prove that $x \notin L$. Player 1 passes some information to Player 2, Player 2 then replies to Player 1, and after the discussion an independent "judge" decides the winner.

Since $L \in$ BPP, there exist a polynomial time Turing machine $M$ and a polynomial $p$ such that $M$ takes as input a string $x$ of length $n$ and a random string $y \in \{0,1\}^{p(n)}$, and

$$\begin{aligned} x \in L \quad &\Longrightarrow \quad M \text{ accepts with probability of more than } \tfrac{1}{2} \\ x \notin L \quad &\Longrightarrow \quad M \text{ accepts with probability of less than } \tfrac{1}{2p(n)} \end{aligned}$$

Let us fix $x \in \{0,1\}^n$. In the two round debate Player 1 will try to find a bijection $\pi : \{0,1\}^{p(n)} \to \{0,1\}^{p(n)}$ such that for every $y \in \{0,1\}^{p(n)}$ at least one of $y$, $\pi(y)$ is "good" for $x$ (recall that $y$ is "good" for $x$ if $x \in L \Leftrightarrow M(x,y)$ accepts). Player 2 will attempt to prove by counterexample that the bijection specified by Player 1 does not satisfy the condition. In other words, Player 2 will try to find a string $y$ such that both $y$ and $\pi(y)$ are "bad" for $x$.

Notice that if $x \notin L$ then the fraction of "good" strings for $x$ is too small for a satisfactory bijection $\pi$ to exist.

Define $\pi_r(x) = x \oplus r$. (Here "$\oplus$" means XOR: for example, $01101 \oplus 10001 = 11100$.)

Since the description of a bijection between $\{0,1\}^{p(n)}$ and $\{0,1\}^{p(n)}$ is too long to be transmitted in polynomial time, let us consider the following debate scheme. Player 1 chooses $p(n)$ strings $r_1, r_2, \ldots, r_{p(n)}$ of length $p(n)$, Player 2 chooses a string $y \in \{0,1\}^{p(n)}$, and the "judge" decides the winner as follows: Player 1 wins (i.e. $x \in L$) if at least one of $M(x, \pi_y(r_i))$ accepts, otherwise Player 2 wins (i.e. $x \notin L$).

First let us show that if $x \in L$ then Player 1 can always choose $r_1, r_2, \ldots, r_{p(n)}$ such that the "judge" concludes that $x \in L$ no matter what $y$ is produced by Player 2. We write

$$\begin{aligned} & \Pr_{r_1,\ldots,r_{p(n)}}[M(x,\pi_y(r_i)) \text{ rejects}] < \frac{1}{2} \\ \Longrightarrow \quad & \Pr_{r_1,\ldots,r_{p(n)}}[M(x,\pi_y(r_i)) \text{ rejects for all } 1 \le i \le p(n)] < 2^{-p(n)} \\ \Longrightarrow \quad & \Pr_{r_1,\ldots,r_{p(n)}}[\exists y \in \{0,1\}^{p(n)} \text{ such that } \forall i \ M(x,\pi_y(r_i)) \text{ rejects}] < 1 \\ \Longrightarrow \quad & \Pr_{r_1,\ldots,r_{p(n)}}[\forall y \in \{0,1\}^{p(n)} \ \exists i \ M(x,\pi_y(r_i)) \text{ accepts}] > 0 \end{aligned}$$

The last inequality means that there exists a sequence $r_1, r_2, \ldots, r_{p(n)}$ such that for any string $y$ the "judge" algorithm will conclude that $x \in L$.

Now suppose that $x \notin L$. In this case for any sequence $r_1, r_2, \ldots, r_{p(n)}$ generated by Player 1 there must exist a string $y$ such that the "judge" algorithm concludes that $x \notin L$. We have

$$\Pr_y[M(x,\pi_y(r_i)) \text{ accepts}] \le \frac{1}{2p(n)} \qquad \Longrightarrow \qquad \Pr_y[\exists i \ M(x,\pi_y(r_i)) \text{ accepts}] \le p(n) \cdot \frac{1}{p(n)} = \frac{1}{2}$$

hence for any sequence $r_1, r_2, \ldots, r_{p(n)}$ there exists $y \in \{0,1\}^{p(n)}$ such that $M(x, \pi_y(r_i))$ rejects for all $i$, as desired.

The following algorithm with two alternations decides $L$: first nondeterministically select $r_1, r_2, \ldots, r_{p(n)}$, then verify that for all $y \in \{0,1\}^{p(n)}$ the "judge" algorithm determines that $x \in L$. Since it takes polynomial time to run the "judge" algorithm for a particular choice of $r_1, r_2, \ldots, r_{p(n)}$ and $y$, it follows that $L \in \Sigma_2^{\mathrm{P}}$.

# Chapter 8

**6.841/18.405J: Advanced Complexity Theory** March 4, 2002

## Lecture 8

*Lecturer: Madhu Sudan* *Scribe: Minh Nguyen*

## 8.1 Introduction

When we introduced the notion of circuits, one of our hopes was to use combinatorial techniques and circuits to prove that $P \neq NP$. We would like to show exponential lower bounds on circuit size for functions in NP, but the best we have been able to show is exponential lower bounds for constant depth circuits.

Today, we introduce the class $AC_0$ and prove a lower bound for the parity function. We do not introduce the class $AC_0$ because of its power but because of the powerful techniques used in the proof (algebraic techniques and randomization).

## 8.2 Circuit depth

We consider circuits with

- NOT gates (unary function),
- OR & AND gates with unbounded fan-in (the gates have an unlimited number of inputs and we take the OR/AND of all the input bits).

The depth of a circuit is defined as the longest path from input to output. (A circuit is an acyclic graph so "longest path" is well-defined and efficiently computable)

The size of a circuit is defined as the number of wires; if we are not interested in polynomial factors, it is also the number of gates in the circuit. We have seen that the circuit size represents non-uniform time complexity. Circuit depth represents parallel time, i.e. how fast a parallel algorithm can solve a problem. The unbounded fan-in simulates concurrent reading and writing on shared memory cells.

We define $AC_0$ as the class of constant depth, poly-size circuits with unbounded fan-in OR and AND gates.

## 8.3 Parity function

For every $n$, the parity function is defined as $\bigoplus_n : \{0,1\}^n \to \{0,1\}$, $\bigoplus(x_1 \ldots x_n) = \sum x_i \pmod 2$.

Since the OR and AND gates have unbounded fan-in, the OR and AND functions can be computed in constant time. We will show that this is not the case for the parity function.

**Theorem 8.1** *If $C$ is a circuit of depth $d$ computing the parity of $n$ bits, then it must have size at least $2^{n^{\Omega(1/d)}}$.*

Note that we are not proving an impossibility result for constant depth circuits. For instance, there exists an exponential size circuit of depth 2 which computes the parity function by writing $\bigoplus(x_1 \ldots x_n)$ in DNF (disjunctive normal form= an OR of ANDs) form, $\bigvee_{S \in \mathcal{F}}(\bigwedge_{i \in S} x_i \cdots)$. There is also a circuit of depth $\log n$ and size $n$ which computes the parity of $n$ bits. Hence we want to show that we cannot have simultaneously small size and small depth: the proof will have to consider these two quantities together.

History of this lower bound:

1. Furst, Saxe, Sipser (83) introduce the method of random restrictions. Their theorem is weaker, the circuit size is superpolynomial in $n$.
2. Yao (85) strengthens the theorem with an exponential lower bound using the same technique.
3. Hastad (87) obtains a simpler proof and a stronger exponential lower bound on the size of the circuit.
4. Smolensky (87) proves the theorem by algebraic methods, this is the proof we'll see today.

## 8.4 Polynomials over $\mathbb{Z}_3$

### 8.4.1 $\mathbb{Z}_3$

We consider $\mathbb{Z}_3 = \{-1, 0, 1\}$ with arithmetic mod 3, where we think of 2 as $-1$ since $-1 \equiv 2 \pmod 3$.

There are two ways to represent the Boolean world in $\mathbb{Z}_3$:

- The obvious one is $\{0,1\} \subseteq \mathbb{Z}_3$,
- The other one is to use the map $\rho : \{0,1\} \to \{1,-1\}$, where $\rho(0) = 1, \rho(1) = -1$.

The map $\rho$ is linear: $\forall x \in \{0,1\}, \rho(x) = 1 - 2x$ and $\forall y \in \{1,-1\}, \rho^{-1}(y) = \frac{1-y}{2}$. We can switch from one representation to the other by a linear transformation over the inputs and think of a function $f : \{0,1\}^n \to \{0,1\}$ or $f : \{1,-1\}^n \to \{1,-1\}$ as functions mapping $\mathbb{Z}_3^n \to \mathbb{Z}_3$.

### 8.4.2 Polynomials over $\mathbb{Z}_3$

$\mathbb{Z}_3$ is a field: polynomials over $\mathbb{Z}_3$ are well-behaved. In particular, the Schwartz lemma from the previous lecture applies: if $p : \mathbb{Z}_3^n \to \mathbb{Z}_3$ is a non-zero polynomial of degree $d$, then

$$\Pr_{\bar{a} \xleftarrow{R} \mathbb{Z}_3^n} [p(\bar{a}) = 0] \leq \frac{d}{3}$$

Note that the Schwartz lemma does not appear to be all that interesting in $\mathbb{Z}_3$: for instance, the polynomial $x_1^3 - x_1$ is zero for any element of the field. However, we will find it useful since we will encounter polynomials of total degree one and for such polynomials the lemma guarantees that the polynomial evaluates to non-zero values on at least 2/3rds of the inputs.

### 8.4.3 Examples

The function $AND(x_1 \dots x_n) : \{0,1\}^n \to \{0,1\}$ can be computed by the polynomial $x_1 \cdot x_2 \dots \cdot x_n$. Similarly, $OR(x_1 \dots x_n) = 1 - \prod_{i=1}^{n}(1 - x_i)$. In each case, the function is computed by a polynomial over $\mathbb{Z}_3$ of degree 1 in each variable. This comes from the following fact:

**Fact 8.2** *For every $f : \{0,1\}^n \to \{0,1\}$, we can find a polynomial $q : \mathbb{Z}_3^n \to \mathbb{Z}_3$ such that $q$ has degree 1 in each variable and agrees with $f$ on $\{0,1\}^n$.*

(proof using AND, OR functions or using interpolation)

We have a similar fact for $g : \{1,-1\}^n \to \{1,-1\}$: $g$ is computed by some polynomial of degree 1 in each variable. Suppose that $g(\rho(x)) = \rho(f(x))$ and the boolean function $f$ is represented by the polynomial $p$. Then $g(y)$ is represented by the polynomial $1 - 2[p\left(\frac{1-y_1}{2}, \frac{1-y_2}{2}, \cdots, \frac{1-y_n}{2}\right)]$.

The parity function has a nice formulation in the $\{1,-1\}$ representation:

$$\bigoplus_n : \{1,-1\}^n \to \{1,-1\}, \bigoplus(x_1 \dots x_n) = \prod_{i=1}^{n} x_i$$

## 8.5 Proof of the theorem

**Proof Idea** The main intuition behind the proof are the following insights.

- The degree of a function is a measure of its complexity.
- Parity has high degree since $\bigoplus(x_1 \dots x_n) = \prod_{i=1}^{n} x_i$.
- Circuits in $AC_0$ compute low degree functions.

There is a caveat in this reasoning: the functions AND and OR have also degree $n$ and belong to $AC_0$! But if we delete part of the input to the function, we can represent the rest with a small depth circuit and a small degree polynomial. We will show that $AC_0$ "essentially" computes only small degree polynomials. These ideas are formalized in the next three lemmas. ■

**Lemma 8.3** *If $f : \{0,1\}^n \to \{0,1\}$ is computed by a depth $d$ circuit of size $s = 2^{n^{o(1/d)}}$, then there exist a set $S \subseteq \{0,1\}^n$ of size $|S| > \frac{3}{4} \cdot 2^n$ and a polynomial $p : \mathbb{Z}_3^n \to \mathbb{Z}_3$ of total degree $\leq (\log s)^{O(d)}$ such that $p(x) = f(x), \forall x \in S$.*

**Lemma 8.4** *If there exists a degree $D$ polynomial $p : \mathbb{Z}_3^n \to \mathbb{Z}_3$ such that $p(x) = \bigoplus(x)$ for all $x \in S \subseteq \{0,1\}^n$, then every Boolean function $f : S \to \{0,1\}$ is represented by a polynomial of total degree $\frac{n}{2} + D$ and of degree 1 in each variable.*

**Lemma 8.5** *If $h_1, \cdots, h_N : S \to \{0,1\}$ are such that for every $f : S \to \{0,1\}, \exists \alpha_1, \cdots, \alpha_N$ such that $f = \sum_{i=1}^{N} \alpha_i h_i$, then $N \geq |S|$.*

**Proof** of Theorem 1: Proof by contradiction. Suppose there exists a circuit $C$ of depth $d$ and size $s = 2^{n^{o(1/d)}}$ computing the parity function.

- By Lemma 3, the parity function is computed on $S$ of size $|S| \geq \frac{3}{4} \cdot 2^n$ by a polynomial of degree $D \leq (\log s)^{O(d)}$.
- By Lemma 4, every Boolean function on $S$ is computed by a polynomial of degree $\frac{n}{2} + D \leq \frac{n}{2} + (\log s)^{O(d)}$.

- Let $h_1, \cdots, h_N$ be all the monomials $x_1^{i_1} \cdots x_n^{i_n}$ where $i_j \in \{0,1\}$ and $\sum i_j \leq \frac{n}{2} + D$. Note that $h_1, \cdots, h_N$ generate all Boolean functions on $S$, hence by Lemma 5, we must have that $N \geq |S| \geq \frac{3}{4} \cdot 2^n$.

  How many such monomials are there?

$$
\begin{aligned}
N &\leq \sum_{i=0}^{\frac{n}{2}+D} \binom{n}{i} && \text{choose how to distribute up to } n/2+D \text{ ones in a vector of size } n \\
N &\leq \sum_{i=0}^{\frac{n}{2}} \binom{n}{i} + \sum_{i=\frac{n}{2}+1}^{\frac{n}{2}+D} \binom{n}{i} \\
N &\leq 2^{n-1} + D \cdot \binom{2^n}{\sqrt{n}} && \text{each term } \binom{n}{i} \text{ for } n/2 < i \leq n/2+D \text{ is smaller than } \binom{2^n}{\sqrt{n}}
\end{aligned}
$$

  If we assume that $s < 2^{n^{o(1/d)}}$, then $\frac{D}{\sqrt{n}} \leq \frac{1}{4}$ and $N < \frac{3}{4} \cdot 2^n$. This contradicts Lemma 5, hence there is no such circuit $C$.

■

## 8.6 Proofs of lemmas

**Proof** of Lemma 5:
Note that the set of functions $f : S \to \{0,1\}$ are members of the vector space $\mathbb{Z}_3^{|S|}$ that contain the "unit" functions $\{\delta_x\}_{x \in S}$ where $\delta_x(y) = 1$ if $y = x$ and 0 otherwise. Furthermore the $\delta_x$ functions are linearly independent of each other. Thus any collection of functions $h_1, \ldots, h_N$ that generate all the functions $f$ must have size $N \geq |S|$. The following paragraph elaborates further on this proof.

Every function $f : S \to \{0,1\}$ can be viewed as a vector of size $|S|$ over $\{0,1\}$. We will ignore most of these functions and just focus on the $\delta_x$ functions defined above. Consider the $|S| \times |S|$ matrix $F$ whose $x$th row is the vector corresponding to the function $\delta_x$. Note this matrix is simply the identity matrix. Now suppose there exists functions $h_1, \cdots, h_N : S \to \mathbb{Z}_3$ such that for every function $f : S \to \{0,1\}$, there exists $\alpha_1, \cdots, \alpha_N \in \mathbb{Z}_3$. such that $f = \sum_{i=1}^N \alpha_i f_i$. In particular, let $\alpha_{x,1}, \ldots, \alpha_{x,N}$ be the multipliers needed to get the function $\delta_x$. Let us represent the functions $h_1, \ldots, h_N$ as $|S|$ dimensional vectors over $\mathbb{Z}_3$ as well, and consider the $N$ by $|S|$ matrix $H$ whose rows are the vectors $h_1, \ldots, h_N$. Now let $A$ be the $|S|$ by $N$ matrix whose rows are indexed by $x \in S$ and columns by index $i \in \{1, \ldots, N\}$ and where the entry $A_{x,i} = \alpha_{x,i}$. By construction, $A \cdot H = F$! Now we get $|S| = \text{rank}(F) = \text{rank}(A \cdot H) \leq \min\{\text{rank}(A), \text{rank}(H)\} \leq N$. ■

**Proof** of Lemma 4: Let $S$ be a subset of $\{0,1\}^n$. Assume that $\bigoplus : \{0,1\}^n \to \{0,1\}$ is represented by a polynomial $p : \mathbb{Z}_3^n \to \mathbb{Z}_3$ of degree $D$. Let $T \subseteq \{1,-1\}^n$ be the associated set, $T = \{\rho(x) | x \in S\}$. Then the map $\rho_{\bigoplus} : T \to \{1,-1\}, \rho_{\bigoplus}(y_1 \ldots y_n) = \prod_{i=1}^n y_i$ agrees with the polynomial $r(\mathbf{y}) = 1 - 2p\left(\frac{1-\mathbf{y}}{2}\right)$ on the set $T$. $r$ is a polynomial of total degree $D$: the hypothesis about the parity function holds in the $\{1,-1\}$ representation as well.

Consider a Boolean function $f : S \to \{0,1\}$. Let $g : T \to \{1,-1\}$ be the associated function which is represented by a polynomial, i.e. by a summation of monomials. Let $\{A_i\}$ be monomials of total degree less or equal to $n/2$ and $\{B_i\}$ be monomials of total degree more than $n/2$.

$$Q_g = \sum_i \alpha_i A_i + \sum_i \beta_i B_i$$

Let $C_j = \frac{\prod_{i=1}^{n} x_i}{B_j}$ (complement of the variables appearing in $B_j$). Then $B_j \cdot C_j = \rho_{\bigoplus}(x_1 \ldots x_n)$ and in the $\{1, -1\}$ representation, $B_j = C_j \cdot \rho_{\bigoplus}(x_1 \ldots x_n)$.

$$Q_g = \sum_i \alpha_i A_i + \rho_{\bigoplus} \sum_i \beta_i C_i$$

$g$ is represented on $T$ by the polynomial $Q_g$ of total degree at most $\frac{n}{2} + D$ and of degree at most 1 in each variable (by substituting $x_i^2 = 1$). Switching back to the $\{0, 1\}$ representation, we have that the function $f$ is represented by the polynomial $Q_f(x) = \frac{1 - Q_g(1-2x)}{2}$ of degree $\frac{n}{2} + D$ on $S$.
■

# Chapter 9

| **6.841/18.405J: Advanced Complexity Theory** | March 6, 2002 |
|---|---|
| Lecture 9 | |
| *Lecturer: Madhu Sudan* | *Scribe: George Savvides* |

Topics for today:

1. Proof of Lemma 1 from last lecture
2. Qualitative overview of the proof that $PARITY \notin \mathcal{AC}_0$
3. "Promise" problems, complexity of Unique SAT

## 9.1 Proving Lemma 1

Recall Lemma 1 from last lecture:

**Lemma 9.1** *If $\mathbb{C}$ is a circuit of size $s$ and depth $d$ and computes function $f$ then there exists a polynomial $p : \mathbb{Z}_3^n \longrightarrow \mathbb{Z}_3$ of degree $D \leq (\log s)^{o(d)}$ and a set $S \subseteq \{0,1\}^n$ with cardinality $|S| \geq \frac{3}{4}2^n$ such that $\forall x \in S, \;\; p(x) = f(x)$*

### 9.1.1 Proof sketch

- WLOG we will assume that the circuit $\mathbb{C}$ consists of NOT and OR gates only (i.e we don't allow AND gates). This restriction can only increase in the size and the depth of the circuit by a constant factor since, using DeMorgan's law, we can replace AND gates with an equivalent configuration that uses only OR and NOT gates (the number of OR and NOT gates will be proportional to the fanin of the AND gate which is a constant).
- We then replace each gate by a polynomial as follows:

  **NOT gates** We will use the polynomial $p = 1 - x$. This polynomial does the right thing when $x \in \{0,1\}$. We won't bother with what happens when $x = -1$ (recall that we are working in $\mathbb{Z}_3$)

  **OR gates** Likewise, we would like to replace each OR gate with a *low degree* polynomial. In order to do this, we will replace each OR gate with a randomly chosen polynomial of degree $\log s$ (the polynomial will be chosen *independently* for each gate).

- We will then show that for a fixed input to the circuit, any of these polynomials (replacing the gates) will correctly compute the gate's output with probability at least $1 - \frac{1}{4s}$ [trivial for NOT gates, more interesting for OR gates].
- For any fixed input, the error probability for each (replaced) gate is at most $\frac{1}{4s}$. Thus, according to the union bound, the probability that there exists at least one polynomial in the entire circuit which computed the wrong result is at most $\frac{1}{4s} \cdot s = 1/4$ and so circuit produces the right result with probability at least $1 - 1/4 = 3/4$.
- We conclude that there must exist polynomials of degree $\log s$ replacing the gates of a circuit $\mathbb{C}$ so that the entire circuit correctly computes the output for $3/4$ of all possible inputs.
- Finally, we conclude that the degree of the output function must be $(\log s)^{O(d)}$.

**Questions and answers**

**Q:** Do we use the same polynomial to replace all the OR gates?

**A:** No. Each polynomial is chosen independently

**Q:** What's our gain?

**A:** In the original circuit, an OR gate may have up to $s$ inputs. Instead of replacing them with degree-$s$ polynomials (which compute the right result 100% of the time), we will randomly select polynomials of much lower degree that will still end up computing the right result most of the time.

### 9.1.2 How to replace gates with low-degree polynomials

**The polynomial representing the entire circuit**

In order to see what the function computed by the "poly-replaced" circuit looks like, we notice that throughout the circuit we are replacing OR gates (with a fanin of size $k$) with polynomials of degree $\log s$ (in $x_1, x_2, \dots, x_k$). At the lowest level of the circuits, where all the the inputs to the gates (polynomials) are constants in $\{0, 1\}$, we have polynomials of degree $\log s$. At the second level, where the inputs to the polynomials are themselves polynomials of degree $\log s$, we will have polynomials of degree $\log^2 s$. In general, if we have a polynomial of degree $d_1$ and all its inputs are polynomials of degree $d_2$, the output will be a polynomial of degree $d_1 d_2$ (can be proved using induction). So continuing all the way to depth $d$, we see that the resulting polynomial (which computes the output of the circuit) has degree $\leq (\log s)^d$.

This polynomial will not be computing the correct value for all inputs. However, for a fixed input and an *independent*, random choice of the polynomials to replace the OR gates, we will show that the probability that a (replaced) gate computes the wrong result is at most $1/4s$. By the union bound, we get the right values throughout the circuit with probability $\geq 3/4$. Notice that this requires that the choice of polynomials be made *independently* of the choice of inputs (i.e. the polynomials were not chosen to suit the input, or the other way around).

Since for a particular input we have a $3/4$ probability of getting the right result when the replacing polynomials are chosen randomly, we conclude that there must exist a particular choice of polynomials (not necessarily the same for all gates) which produces the correct result for at least $3/4$ of all possible inputs to the circuit.

### 9.1.3 Finding suitable $\log s$-degree polynomials

In order to get some intuition about what those $\log s$-degree polynomials might look like, it helps to recall that $\mathrm{OR}(y_1, y_2, \dots, y_k)$ is 1 if at least one of these variables is 1 and 0 otherwise. The polynomial $1 - \prod_{i=1}^{k}(1 - y_i)$

always produces the right result. Unfortunately, it has degree $k$. If we allow for a non-zero error probability, we can bring the degree down to $\log s$.

We will see how we can construct such polynomials in steps: We will first start with a degree-1 polynomial, then go to degree-two polynomials, and finally degree-$s$ polynomials which compute the right result with probability $1 - \frac{1}{4s}$.

**Degree 1** In order to construct a degree-1 polynomial which approximates $\mathrm{OR}(y_1, y_2, \ldots, y_n)$, we simply pick $a_1, a_2, \ldots, a_n \in_R \{1, 0, -1\}$ ($\in_R \equiv$ "randomly selected from"). We then construct the polynomial

$$p_{\bar{a}}(\bar{y}) = \sum_{i=1}^{k} a_i y_i$$

which is of degree 1 and whose output is *always* 0 when all the $y$'s are 0 (as it should be). But what happens when its input is non-zero? It turns out that we can use the Schwarz lemma to show that

$$\Pr_{\bar{a}}\left[p_{\bar{a}}(\bar{y}) = 0 \text{ if } \bar{y} \neq 0\right] \leq \frac{1}{3}$$

To see why the Schwarz lemma applies in this case, consider fixing $\bar{y} \neq 0$ and defining $Q(\bar{z}) = \sum_{i=1}^{k} z_i y_i$. Then the probability that $Q(\bar{z}) = 0$ if $Q \not\equiv 0$ is at most $\frac{1}{3}$. If we let $\bar{z}$ be the random choice of $\bar{a}$ we used above, this probability remains the same.

So, on a non-zero input, our polynomial is non-zero with probability $\geq 2/3$. But then it could be either 1 or $-1$!

**Degree 2** We can fix this $-1$ issue by simply squaring everything. Then:

$$p^2_{\bar{a}}(\bar{y}) = \left[p_{\bar{a}}(\bar{y})\right]^2 = \begin{cases} 0 & \text{with prob. 1 if } \bar{y} = 0 \\ 1 & \text{with prob} \geq 2/3 \text{ if } \bar{y} \neq 0 \end{cases}$$

Therefore, $\forall \bar{y},\ p^2_{\bar{a}}(\bar{y})$ produces the right result with probability $\geq 2/3$.

**Higher degree polynomials** In order to boost this result, we notice that our degree-2 polynomial always produces the right result when $\bar{y} = 0$, but when $\bar{y} = 1$, it will not necessarily produce a 1. This motivates the following approach: pick $\bar{a}_1, \bar{a}_2, \ldots, \bar{a}_\ell$ at random and come up with a polynomial that computes (with 100% accuracy)

$$\mathrm{OR}\left(p^2_{\bar{a}_1}, p^2_{\bar{a}_2}, \ldots, p^2_{\bar{a}_\ell},\right)$$

A polynomial that computes the OR function of $\ell$ variables, each of which is a polynomial of degree 2, will have degree $2\ell$. As for the correctness of the output, the probability that all $\ell$ polynomials inside the OR function fail to produce the correct result is $\leq (\frac{1}{3})^\ell$, so the probability that its result is correct is $\geq 1 - (\frac{1}{3})^\ell$. If we pick $\ell = \log s$ we get the result we wanted: a polynomial of degree $\log s$ computing the OR function with accuracy $1 - \frac{1}{4s}$.

We then repeat the process for all the gates independently.

## 9.2 Qualitative overview of the lower bound proof for PARITY

Here's a summary of the proof of the main theorem we set out to prove last time (PARITY $\notin \mathcal{AC}_0$).

1. If PARITY has a small depth, small size circuit $\mathbb{C}$ computing it, then, since $\mathbb{C}$ can be approximated by a polynomial, PARITY will have a low-degree polynomial computing it on *most* inputs.

2. By changing notations[1], we have seen that $\prod_{i=1}^{n} x_i$ has a "small degree" (say $D$) polynomial computing it correctly on most inputs in $\{-1,+1\}^n$.

3. This allows us to say that every boolean function has a low degree[2] polynomial computing it on most inputs. Therefore, every boolean function is in the linear span of a small number[3] of monomial functions.

4. This leads to a contradiction: the boolean functions (and in particular the $\delta_x$ functions given by $\delta_x(y) = 1$ iff $x = y$) require large bases on large domains. You cannot have *every* boolean function in the span of a small set of monomials. (see Lecture 08 for details).

**Alternative Proof:** There's an alternative proof in Spielman's notes from 2000 (lecture 13?) `http://www-math.mit.edu/ spielman/AdvComplexity/2000/`

## 9.3 Complexity of Unique SAT

### 9.3.1 Motivation

The complexity of Unique SAT was initially studied in the context of cryptography. In particular, Diffie and Hellman had proposed a list of candidate one-way functions and processes which contained, among others, the following examples:

1. Take a piece of code written in a high-level language and compile it. The process is demonstrably easy in one direction, but conjectured to be"hard" in the other

2. Take a formula $\phi$ and one of its satisfying assignments $\mathbf{a}$, and delete $\mathbf{a}$. That is: $(\phi, \mathbf{a}) \longrightarrow \phi$

   While the forward direction is easy, the reverse is also conjectured to be hard (clarification: it is not necessary to retrieve the same $\mathbf{a}$: any satisfying assignment will do).

3. Take two prime numbers $p, q$ and multiply them to get $p \cdot q$. That is: $p,\ q \to p \cdot q$

   Again, one direction (multiplication) is easy, while the other (factoring) seems harder.

But how does the difficulty of factoring relate to the difficulty of finding a satisfying assignment $\mathbf{a}$ of the previous example? Several similar questions were being asked at the time:

1. How did "changing the problem" affect its hardness?

2. What if $(\phi, \mathbf{a})$ were chosen at random?

3. What if $\phi$ were an instance with a known satisfying assignment $\mathbf{a}$?

These were clearly major gaps in our understanding of hardness. However, what bothered the cryptographers of the time most was another important question: going from $(\phi, \mathbf{a})$ to $\phi$ clearly results in *loss of information*, while going from $p$ and $q$ to $pq$ doesn't. In other words,

- $(\phi, \mathbf{a}) \longrightarrow \phi$ is an information-hiding map
- $p,\ q \to p \cdot q$ is an information-preserving map (and actually a one-to-one function)

[1] Notice that we can change the representation from $\{0,1\}$ to $\{+1,-1\}$ "for free" without changing the degree of the polynomials involved.

[2] $n/2 + D$ to be precise

[3] $\sum_{i=0}^{n/2+D} \binom{n}{i}$ to be precise

**Question:** Could hardness be a property of information-hiding maps only?

The hardness of unique-SAT provided a good starting point towards answering this question. Obviously, if $\phi$ is known to have only one satisfying assignment, there is no information loss when going from $(\phi, \mathbf{a})$ to $\phi$. However, does this make USAT any easier than SAT?

### 9.3.2 Formalizing the problem

**Promise Problems**

A lot of questions in Complexity are phrased in the following way: "Can you solve the problem on instances I care about?"

That's where promise problems comes into play. A promise problem $\Pi$ consists of two sets, $\Pi_{\text{YES}}$ and $\Pi_{\text{NO}}$, both of which are subsets of $\{0,1\}^*$ and whose intersection is the empty set. That is:

- $\Pi = (\Pi_{\text{YES}}, \Pi_{\text{NO}})$
- $\Pi_{\text{YES}}, \Pi_{\text{NO}} \subseteq \{0,1\}^*$
- $\Pi_{\text{YES}} \cap \Pi_{\text{NO}} = \emptyset$

Given a problem in $\Pi$, our goal is to come up with an algorithm $A$ which satisfies the following two conditions:

**1. Completeness** $x \in \Pi_{\text{YES}} \implies A(x)$ accepts

**2. Soundness** $x \in \Pi_{\text{NO}} \implies A(x)$ rejects

If algorithm $A$ satisfies these two conditions, then we say that it solves our problem. Note that other variants of this definition are also available. In particular, the definition can be naturally extended to the probabilistic world.

**Strings we don't care about:** Notice that if $\Pi_{\text{YES}} \cup \Pi_{\text{NO}} = \{0,1\}^*$ then solving the promise problem $\Pi$ is the same as deciding the language $\Pi_{\text{YES}}$. Loosely, we say that in this case the promise problem is just a language. The novelty of of promise problems is that it allows us to to specify strings we don't care about, namely the strings in $\{0,1\}^* - \Pi_{\text{YES}} - \Pi_{\text{NO}}$. Our algorithms can feel free to do whatever they want when given an input from that set. (However, they do need to halt, a condition that can easily be enforced by using a counter.)

There is a broad collection of problems that fall into the category of promise problems, including approximation algorithms and randomized algorithms.

### 9.3.3 USAT as a promise problem

Consider the following definition of USAT as a promise problem:

- $\text{USAT} = (\text{USAT}_{\text{YES}}, \text{USAT}_{\text{NO}})$
- $\text{USAT}_{\text{YES}} = \{\phi \mid \phi \text{ has exactly one satisfying assignment}\}$
- $\text{USAT}_{\text{NO}} = \{\phi \mid \phi \text{ is not satisfiable}\}$

### 9.3.4 Hardness of USAT

Moving back to the cryptographic motivation, suppose we consider the map $(\phi, \mathbf{a}) \longrightarrow \phi$, but now restrict $\phi$ to be a member of $\text{USAT}_{\text{YES}}$. Since $\mathbf{a}$ is unique, this map is information preserving. However, now it is not clear if this map is hard to invert. If we could show that it is, then we would answer, negatively, the question we raised earlier about complexity: I.e., "is intractability a consequence of information-hiding?" To be more formal, we will explore the question of whether USAT $\in \mathcal{P}$.

It turns out that it isn't, as a direct consequence of the following theorem due to Valiant and Vaziram:

**Theorem 9.2** *SAT reduces to USAT probabilistically.*

If USAT $\in \mathcal{P}$ then there exists an algorithm running in polynomial time which solves this particular problem with probability $\frac{1}{\text{poly}(n)}$. More precisely, if USAT $\in \mathcal{P}$ then $\mathcal{NP} = \mathcal{RP}$.

The probabilistic reduction takes a formula $\phi$ and creates another formula $\psi$ with the following properties:

1. $\phi \in \text{SAT} \implies \psi \in \text{USAT}_{\text{YES}}$ with probability $\frac{1}{\text{poly}(n)}$
2. $\phi \notin \text{SAT} \implies \psi \in \text{USAT}_{\text{NO}}$ with probability 1

Notice that we have no idea how to boost the probability from $\frac{1}{\text{poly}(n)}$ to something better in the first case.

**Related question:** Suppose that the reduction is given and that USAT $\in \mathcal{P}$. Can we use it to compute a satisfying assignment to the SAT problem?

**Answer:** Yes. Use the self-reducibility of USAT.

### 9.3.5 Main ideas for reduction from SAT to USAT

Say we have the universe $\{0,1\}^n$ and a formula $\phi$. Let $S = \{a \mid \phi(a) = 1\}$. Suppose further that we know $M = |S|$. We will do the following:

We pick a random (hash) function $h : \{0,1\}^n \longrightarrow \{0, \ldots, M-1\}$. Then, with probability at least some constant $c$, there exists an $x \in S$ such that $h(x) = 0$. Define $\psi(x)$ as follows:

$$\psi(x) = \phi(x) \wedge [h(x) = 0]$$

Notice that this is not really a SAT formula, but the completeness of SAT for NP indicates we can convert the above expression into a SAT formula, specifically by solving the following exercise.

Exercise: Given a circuit $C : \{0,1\}^n \to \{0,1\}$ show that it is possible to construct, in time poly$(|C|)$, a 3CNF formula $\phi(\mathbf{x}, \mathbf{y})$, where $\mathbf{x}$ is a vector of $n$ variables and $\mathbf{y}$ is a vector of at most $|C|$ variables such that for every $\mathbf{x} \in \{0,1\}^n$

$$C(\mathbf{x}) = 1 \Rightarrow \exists! \mathbf{y} \text{ s.t. } \phi(\mathbf{x}, \mathbf{y}) \text{ and } C(\mathbf{x}) = 0 \Rightarrow \forall \mathbf{y}, \phi(\mathbf{x}, \mathbf{y}) = 0.$$

A simple probability calculation (omitted from today's lecture) shows that if $\phi$ is satisfiable, then with some constant probability we also have $h(x) = 0$ for a *unique* $x \in S$. If $h$ could be computed efficiently, then this reduction would also take only polynomial time and we would be done.

**Problems with this reduction:**

1. If $h$ is random: we can not be able to specify it succinctly or compute it efficiently.
2. We don't really know $M$.

**Solution:**

1. We will not require $h$ to be completely random. Instead, we will require only *pairwise independence*(defined below).

2. We will guess M. It turns out that if we are within a factor of 2, the probability calculations will still work out (so we only need to make logarithmically many guesses).

**Pairwise independence**

**Definition 9.3** *$\mathcal{H} \subseteq \{f : \{0,1\}^n \longrightarrow \{0,1\}^m\}$ is a pairwise independent family of hash functions if for all $a \neq b \in \{0,1\}^n$ and $c, d \in \{0,1\}^m$ the following holds:*

$$\Pr_{h \in \mathcal{H}}[h(a) = c \wedge h(b) = d] = \left(\frac{1}{2^m}\right)^2$$

In the definition above, notice two things:

1. We only required that $a \neq b$. We do want the condition to hold for all pairs $(c, d)$ including when $c = d$.

2. $2^m$ is the size of the range of $\mathcal{H}$.

**Definition 9.4** *$\mathcal{H}$ is* nice *if $h \in \mathcal{H}$ can be (1) efficiently sampled and (2) efficiently computed*

**Example** The following family of hash functions is both pairwise independent and nice:

- $\mathcal{H} = \{h_{A,b}\}$ where
- $A$ is an $m \times n$, $\{0,1\}$ matrix
- $b$ is an $m$-dimensional $\{0,1\}$ vector
- $h_{A,b}(x) = Ax + b$ (all operations done $\mod 2$)

The proof is left as an exercise.

In the next lecture we will complete the reduction from SAT to USAT using the two ideas above (pairwise independent hash $h$, and guessing $M$ approximately).

# Chapter 10

| **6.841/18.405J: Advanced Complexity Theory** | March 10, 2002 |
|---|---|
| Lecture 10 | |
| *Lecturer: Madhu Sudan* | *Scribe: David Pritchard* |

The topics that will be covered today are:

1. Completing the proof of the Valiant-Vazirani Theorem
2. Introduction to the "Counting Problem" class #P
3. Begin the proof of Toda's Theorem, $\mathsf{PH} \subseteq \mathrm{P}^{\#\mathsf{P}}$

## 10.1 SAT and USAT

Last class we introduced the problem of unique satisfiability, USAT. Unlike the other problems we have so far discussed, USAT is not a language but instead a "promise problem": we want a TM that will decide if a given formula $\phi$ is satisfiable or not *under the condition that we promise the formula is either unsatisfiable or uniquely satisfiable* (that is, $\{x|\phi(x) \text{ is true}\}$ is $\emptyset$ or has exactly one element). It is evident that USAT is, in a sense, easier than SAT since anything that decides SAT will also decide USAT. However, if USAT is easy enough then another surprising conclusion is true:

**Theorem 10.1 (Valiant-Vazirani)** *$USAT \in \mathrm{P}$ implies $\mathsf{NP} = \mathsf{RP}$.*

We prove this by means of the following lemma, which effectively states that there is a randomized reduction from SAT to USAT:

**Lemma 10.2** *In polynomial time we can probabilistically reduce a formula $\phi \in SAT$ to another formula $\psi$ such that $\phi \notin SAT \Rightarrow \psi \notin SAT$, and $\phi \in SAT \Rightarrow$ with probability $1/\mathrm{poly}(n), \psi$ is uniquely satisfiable.*

(Here $n$ in the number of variables in *phi*). Lemma 2 implies Theorem 1 because, if USAT $\in$ P, then our $\mathsf{RP}$-machine for solving SAT could just perform this reduction polynomially many times (as many as is needed to amplify the probability of finding at least one correct reduction to a constant) and accept iff our USAT solver accepts any of these reductions[1]. We proceed, then, to prove the Lemma, as outlined in the previous lecture.

[1] Since $\psi$ may be multiply-satisfiable it is possible that we are giving the USAT solver problems outside of $\mathrm{USAT}_{\mathrm{NO}} \bigcup \mathrm{USAT}_{\mathrm{YES}}$; however the one-way error of our reduction means that it could only possibly accept formulae reduced from a satisfiable $\phi$.

### 10.1.1 Proving SAT reduces to USAT

**Proof Strategy**: Let $\psi(x) = \phi(x) \wedge h(x) = 0$ for a suitably chosen $h$.

First of all, we want to pick $h$ from a pairwise independent, nice (as defined in the last lecture) family of functions: so we will pick our $h$ from the family $\{h_{A,b} : x \mapsto Ax + b\}$ where all operations are done in $\mathbb{Z}_2$, $A$ is an $m \times n$ matrix, $b$ is an $m$-element vector, and all elements of $A$ and $b$ are chosen uniformly from $\{0, 1\}$. (Note that these functions take elements of $\mathbb{Z}_2^n$ to elements of $\mathbb{Z}_2^m$, so the phrase "$h(x) = 0$" is a vector equality.) Note that if $\phi$ is unsatisfiable, it is immediate that $\phi$ is unsatisfiable also: so henceforth, we will concern ourselves only with the case that *phi* has one or more satisfying assignments.

The question remains, what should $m$ be? Denote the set of all satisfying assignments of $\phi$ by $S$, and let $M = |S|$. If we know the value of $M$, it turns out (as we will see shortly) that taking $m$ such that $2^{m-2} \leq M \leq 2^{m-1}$ is a good choice. However, there is no cheap way to even approximate $M$, so what we do is the following:

**Choice of $m$**: Choose $m$ randomly (and uniformly) from $\{2, 3, \ldots, n + 1\}$.

Since $\phi$ has between 0 and $2^n$ satisfying assignments (its $n$ boolean variables can only take on $2^n$ distinct values), we have a $1/n$ chance of picking the correct $m$.

### 10.1.2 How Well Does This Work?

In the $1/n$ chance that we have picked the correct $m$, we would like to know how likely it is that our new formula has only one satisfying assignment.

**Lemma 10.3** *If $2^{m-2} \leq M \leq 2^{m-1}$, then $\psi$ is uniquely satisfiable with probability $\geq 1/8$.*

**Proof**

For a given boolean $n$-vector $x$ in $S$, Let $G_x$ be the event $(h(x) = 0 \wedge \forall y \in S \backslash \{x\}, h(y) \neq 0)$. Note that the $G_x$ are mutually exclusive. Now, call $y \in S$ "bad for $x$" if $h(x) = h(y) = 0$.

We see that Pr[$y$ is bad for $x$] is $1/2^m \cdot 1/2^m$, since Pr[$x$=0]=$1/2^m$, Pr[$y$=0]=$1/2^m$, and $h$ was selected from a pairwise independent family of hash functions. Thus, by using the union-bound, Pr[$\exists y$ such that $y$ is bad for $x$] $\leq |S - \{x\}|/2^{2m} < M/2^{2m}$. Hence

$$\begin{aligned}
\Pr[G_x] &= \Pr[h(x) = 0 \wedge \forall y \in S \backslash \{x\}, h(y) \neq 0] && (10.1)\\
&= \Pr[h(x) = 0 \wedge \neg(\exists y : y \text{ is bad for } x)] && (10.2)\\
&= \Pr[h(x) = 0] - \Pr[\exists y : y \text{ is bad for } x] && (10.3)\\
&> \frac{1}{2^m} - \frac{M}{2^{2m}} && (10.4)\\
&= \frac{(2^m - M)}{2^{2m}} && (10.5)
\end{aligned}$$

Which in turn gives us

$$\begin{aligned}
\Pr[\psi \text{ is uniquely satisfiable}] &= \Pr[\exists x : G_x] && (10.6)\\
&> \frac{M \cdot (2^m - M)}{2^{2m}} && (10.7)\\
&= \frac{M}{2^m} \cdot \frac{2^m - M}{2^m} && (10.8)\\
&\geq \frac{2^{m-2}}{2^m} \cdot \frac{2^{m-1}}{2^m} && (10.9)\\
&= 1/8 && (10.10)
\end{aligned}$$

(where step 9 comes from the inequality $2^{m-2} \leq M \leq 2^{m-1}$), which completes the proof. ■

Now, to complete the proof of the lemma, we need only observe that this implies Pr[$\psi$ is uniquely satisfiable] $\geq$ Pr[$m$ was chosen correctly]·Pr[$\psi$ is uniquely satisfiable $|m$ is chosen correctly] $> 1/8n$.

### 10.1.3 Finishing Up

We still need to show that $h(x)$ can actually be expressed as a boolean formula. By using a process similar to that described in the proof of the Cook-Levin theorem, we can rewrite $h(x)$ as $\exists y : \eta(x,y)$. Roughly speaking, $\eta(x,y)$ means "$y$ represents a computation of a TM calculating $h(x)$, and $h(x) = 0$". Furthermore it is evident from this process that at most one such $y$ exists, so our new, uniquely satisfiable (or unsatisfiable) formula would be $\psi(x,y) = h(x) \wedge \eta(x,y)$.

### 10.1.4 Remarks

This reduction is pretty remarkable, but there are stronger related reductions that remain open questions:

- Is there a non-probabilistic reduction from SAT to USAT?
- Is there a high-probability reduction from SAT to USAT?
- Is there a reduction from SAT to USAT whose accuracy we can check in polynomial time?

All of these questions have applications in cryptography, since USAT can be used to define a certain class of one-way functions (as mentioned in the last lecture).

## 10.2 The Classes #P and $\mathrm{P}^{\#\mathsf{P}}$

Consider the problem of counting how many satisfying assignments a particular boolean formula has. How difficult is this problem? How can we relate it to other problems? These questions motivate the definition of the class #P :

**Definition 10.4** *#P is the collection of all functions $f : \{0,1\}^* \to \mathbb{Z}_{\geq 0}$ defined by $f(x) = |\{y | M(x,y) \text{ halts}\}|$, where $M$ is any polynomial-time (in terms of the first argument) TM.*

An equivalent definition would be to express $f(x)$ as the number of accepting paths for a poly-time NTM on input $x$. Note that this is a class of functions, not of languages; but we can change a function $f$ into the language $L_f = \{< x, f(x) > | x \in \mathbb{Z}\}$ and a language into a function to $\{0,1\}$, so we mix the two freely.

The class $\mathrm{P}^{\#\mathsf{P}}$ is the class of polynomial-time computable functions on TMs that have oracle access to all #P functions (its queries are of the form $< M, x >$ where $x$ is any string and $M$ is a machine of the type described above).

The reductions of Cook and Karp are useful in proving completeness in this class, since they can be made preserve the number of second-arguments for any particular input; because of this, some of the complete functions for $\mathrm{P}^{\#\mathsf{P}}$ are:

- #SAT: How many satisfying assignments does a formula $\phi$ have?
- #HAMCYC: How many hamiltonian cycles does a directed graph have?
- #CYC: How many (simple?) cycles does a directed graph have? (Can be obtained from #HAMCYC).
- How many matchings does a given bipartite graph have?
- What is the *permanent* of a given matrix? (See below for definition).

The fact that #CYC is complete for this class is quite surprising, since determining whether a graph on $n$ nodes has a cycle can be done in $O(n^4)$ time. The *permanent* of an $n$-by-$n$ matrix $A = \{a_{i,j}\}$ is given by the formula

$$\mathrm{perm}A = \sum_{\pi \in S_n} \prod_{i=1}^{n} a_{i,\pi(i)}$$

where $\pi \in S_n$ means that we take the sum over all permutations $\pi$ of $\{1, 2, \ldots, n\}$. It is used in physics, for example to compute the energies of certain systems. The determinant can be defined by an almost-identical formula:

$$\det A = \sum_{\pi \in S_n} (-1)^{l(\pi)} \prod_{i=1}^{n} a_{i,\pi(i)}$$

where $l(\pi)$ is the "length", or number of inversions in $\pi$. However, despite the formulas' similarities, the permanent is $\mathrm{P}^{\#\mathsf{P}}$-complete and the determinant can be computed in $O(n^3)$ time (by using Gaussian elimination, or LU-decomposition)!

We can see the upper bound $\mathrm{P}^{\#\mathsf{P}} \subseteq \mathsf{PSPACE}$ by a straightforward simulation argument; whether this inequality is an equality is an open question. It is clear that $\mathsf{NP}, \mathsf{coNP} \subseteq \mathrm{P}^{\#\mathsf{P}}$ since we can just query the oracle with a SAT TM and return yes if at least one (respectively, all) of the computation paths accept. $\mathsf{BPP}, \mathsf{RP}, \mathsf{co\text{-}RP} \subseteq \mathrm{P}^{\#\mathsf{P}}$ since we can just compute how many of the possible branches accept and take the majority response. What about $\Sigma_2^P$? Toda's theorem answers this question:

**Theorem 10.5 (Toda)** $\forall i, \Sigma_i^P \subseteq \mathrm{P}^{\#P}$.

We will (start to) prove this theorem after we introduce some notation.

## 10.3 Complexity Class Operators and Toda's Theorem

An "operator" can be thought of a higher-order function: its input and output are complexity classes (sets of languages). We write the result of applying operator $\mathcal{O}$ to the class $\mathcal{C}$ as $\mathcal{O} \cdot \mathcal{C}$. A particularly simple operator is $\neg$, defined by $\neg \cdot \mathcal{C} = \{\overline{L} | L \in \mathcal{C}\}$. In words, $\neg \cdot \mathcal{C}$ is the class of complements of languages in $\mathcal{C}$. Several other operators include:

- $\exists : \mathcal{C} \mapsto \{\{x | \exists y, (x, y) \in L\} | L \in \mathcal{C}\}$
- $\forall : \mathcal{C} \mapsto \{\{x | \forall y, (x, y) \in L\} | L \in \mathcal{C}\}$
- $\bigoplus : \mathcal{C} \mapsto \{\{x |$ for an odd number of $y, (x, y) \in L\} | L \in \mathcal{C}\}$
- $\mathsf{BP} : \mathcal{C} \mapsto \{L' | \exists L \in \mathcal{C} : (x \in L' \to L$ contains at least a fraction $c(n)$ of $(x, y), x \notin L' \to L$ contains at most a fraction $s(n)$ of $(x, y), c(n) - s(n) \geq 1/\mathrm{poly}(n)\}$

Although these definitions are quite technical, they are most easily understood by a few examples:

- $\exists \cdot \mathrm{P} = \mathsf{NP}$
- $\forall \cdot \mathrm{P} = \mathsf{coNP}$
- $\mathsf{BP} \cdot \mathrm{P} = \mathsf{BPP}$
- $\neg \cdot \mathsf{NP} = \mathsf{coNP}$
- $\exists \cdot \Sigma_3^P = \Sigma_3^P$
- $\forall \cdot \Sigma_3^P = \Pi_4^P$

### 10.3.1 Overview of Toda's Theorem

In order to show that all $\Sigma_i^P \subseteq \mathrm{P}^{\#\mathsf{P}}$, we proceed in several steps:

1. $\Sigma_i^P \subseteq \mathsf{BP} \cdot \bigoplus \cdot \Pi_{i-1}^P$, and $\Pi_i^P \subseteq \mathsf{BP} \cdot \bigoplus \cdot \Pi_{i-1}^P$ (Extends Valiant-Vazirani.)
2. $\mathsf{BP} \cdot \bigoplus \cdot \mathrm{P}$ amplifies error (Subtle.)
3. $\bigoplus \cdot \mathsf{BP} \cdot \bigoplus \cdot \mathrm{P} \subseteq \mathsf{BP} \cdot \bigoplus \cdot \bigoplus \cdot \mathrm{P} \subseteq \mathsf{BP} \cdot \bigoplus \cdot \mathrm{P}$ (Surprising, but straightforward.)
4. $\mathsf{BP} \cdot \mathsf{BP} \cdot \bigoplus \cdot \mathrm{P} \subseteq \mathsf{BP} \cdot \bigoplus \cdot \mathrm{P}$ (Not surprising, straightforward.)
5. $\mathsf{BP} \cdot \bigoplus \cdot \mathrm{P} \subseteq \mathrm{P}^{\#\mathsf{P}}$ (Completely separate theorem.)

Once we have shown all of this, an easy induction proof shows that each of the classes $\Sigma_i^P$ can be collapsed into $\mathsf{BP} \cdot \bigoplus \cdot \mathrm{P}$, which means that they are all subseteqs of $\mathrm{P}^{\#\mathsf{P}}$.

### 10.3.2 Proof of Step 1

We need to prove: $\Sigma_i^P \subseteq \mathsf{BP} \cdot \bigoplus \cdot \Pi_{i-1}^P$

We will show that $i$-TQBF is in $\mathsf{BP} \cdot \bigoplus \cdot \Pi_{i-1}^P$, which is sufficient since this is a complete problem for the class. Consider the formula $\exists x_1 \forall x_2 \ldots \mathsf{Q}_i x_i \phi(x_1, x_2, \ldots x_n)$. We pick a pairwise-independent, nice hash function $h$ and consider the number of solutions to $\forall x_2 \ldots \mathsf{Q}_i x_i \phi(x_1, x_2, \ldots x_n) \wedge h(x_1) = 0$. With inverse polynomial probability, there will be exactly zero or one solutions: so applying $\bigoplus$ to this problem gives enough information to solve the original $i$-TQBF problem, and we are done.

We also need to prove: $\Pi_i^P \subseteq \mathsf{BP} \cdot \bigoplus \cdot \Pi_{i-1}^P$. Consider

$$\begin{aligned}
\Pi_i^P &= \neg \cdot \Sigma_i^P && (10.11)\\
&\subseteq \neg \cdot \mathsf{BP} \cdot \bigoplus \cdot \Pi_{k-1}^P && (10.12)\\
&= \mathsf{BP} \cdot \neg \cdot \bigoplus \cdot \Pi_{k-1}^P && (10.13)\\
&= \mathsf{BP} \cdot \bigoplus \cdot \Pi_{k-1}^P && (10.14)
\end{aligned}$$

We just proved the first assertion; the second assertion follows, roughly, from the fact that $BP$ is symmetric in allowing errors on both sides (ie both falsely accepting and falsely rejecting strings). The third assertion can be thought of in two steps: first, that we can create a complement of a class being operated on by $\bigoplus$ by simply accepting one additional second-element for each first-element in the class (in other words, adding one to an even number makes an odd number and vice-versa); secondly, that the class $\Pi_{k-1}^P$ is closed under this operation.

### 10.3.3 To Be Continued next class...

# Chapter 11

| **6.841/18.405J: Advanced Complexity Theory** | March 13, 2002 |
|---|---|
| Lecture 11 | |
| *Lecturer: Dan Spielman* | *Scribe: M.T. Hajiaghayi* |

In this lecture, we continue the proof of Toda's theorem, by proving some lemmas and theorems whose proofs were missed in the previous lecture. First we remind Toda's theorem.

**Theorem 11.1 (Toda 1988)** $PH \subseteq P^{\#P}$.

In the previous lecture, we introduced some interesting operators such as $\exists, \forall, BP$, and $\oplus$. Also, we introduced the steps of the proof. For example, we showed that $\sum_i^P \subseteq BP \cdot \oplus \cdot BP \cdot \oplus \cdots BP \cdot \oplus \cdot P$ (Step 1) and showed very briefly how we can simplify the sequence of operations (Steps 3 and 4). Today first we prove Steps 3 and 4 more precisely. Then we discuss amplifying $BP \cdot \oplus \cdot P$ (Step 2) and finally finish the proof by showing $BP \cdot \oplus \cdot P \subseteq P^{\#P}$ (Step 5).

**Claim 11.2** *For any class $C$ such as $P$ that we can amplify $BP \cdot C$,*

$$\begin{aligned}\oplus \cdot \oplus \cdot C &= \oplus \cdot C \\ BP \cdot BP \cdot C &= BP \cdot C \\ \oplus \cdot BP \cdot C &\subseteq BP \cdot \oplus \cdot C.\end{aligned}$$

**Proof**

**1)** Let $L$ be a language in $C$. Then $x \in \oplus_y \cdot \oplus_z \cdot L(x,y,z)$ means there is an odd number of $y$'s for which there is an odd number of $z$'s such that $(x,y,z) \in L$. It is equivalent to that there is an odd number of $(y,z)$ pairs for which $(x,(y,z)) \in L$.

**2)** First we amplify class $BP \cdot C$. Let $L$ be a language in $C$. Then $x \in BP_y \cdot BP_z \cdot L$ means for at least $c(n)$ fraction of $y$'s, where $c(n) \geq s(n) + \frac{1}{poly(n)}$, for at least $1 - \frac{1}{exp(n)}$ fraction of $z$'s, we have $(x,y,z) \in L$. Thus our new $c'(n) \geq (s(n) + \frac{1}{poly(n)})(1 - \frac{1}{exp(n)})$. On the other hand, for $x \notin L$ for at least $1 - s(n)$ fraction of $y$'s, for at most $\frac{1}{exp(n)}$ fractions of $z$'s we accept $x$. Thus for $s'(n) \leq s(n) + \frac{1}{exp(n)}$ fraction of $(y,z)$ pairs, we accept $x$. Here we can observe that still $c'(n) \geq s'(n) + \frac{1}{poly'(n)}$.

**3)** Let $L$ be a language in $C$. Then $x \in \oplus_y \cdot BP_z \cdot L$ iff there is a polynomial $p_1(n)$ and a language $L' = BP_z \cdot L \in BP \cdot C$ such that $(x,y) \in L'$ for an odd number of $y$'s with length $p_1(|x|)$.

Now, we amplify the error probabilities of the $BP$ operator such that the error is less than $2^{-2p_1(|x|)}$. Then there is a polynomial $p_2(n)$ such that

1. $(x,y) \in L' \rightarrow Pr_{|z|=p_2(|(x,y)|)}[(x,y,z) \in L] > 1 - 2^{-2p_1(|x|)}$

2. $(x, y) \notin L' \rightarrow Pr_{|z|=p_2(|(x,y)|)}[(x, y, z) \in L] < 2^{-2p_1(|x|)}$

Using above facts, we observe

$$x \in \oplus_y \cdot BP_z \cdot L \Rightarrow Pr_z[(x, y, z) \in L] > 1 - 2^{-2p_1(|x|)} \text{ for an odd number of } y.$$

$$x \notin \oplus_y \cdot BP_z \cdot L \Rightarrow Pr_z[(x, y, z) \in L] > 1 - 2^{-2p_1(|x|)} \text{ for an even number of } y.$$

In other words for any y, $Pr_z[(x, y, z) \in L$ disagrees with $(x, y) \in L'] < 2^{-2p_1(|x|)}$.
Thus, $Pr_z[(x, y, z) \in L$ disagrees with $(x, y) \in L'$ for all $y] < 2^{-2p_1(|x|)} \cdot 2^{p_1(|x|)} = 2^{-p_1(|x|)}$.
Therefore,

$$x \in \oplus_y \cdot BP_z \cdot L \Rightarrow Pr_z[(x, y, z) \in L \text{ for an odd number of } y] > 1 - 2^{-p_1(|x|)}$$

and

$$x \notin \oplus_y \cdot BP_z \cdot L \Rightarrow Pr_z[(x, y, z) \in L \text{ for an odd number of } y] < 2^{-p_1(|x|)}$$

, as desired. ■

Now, we discuss Step 2 of Toda's proof. To this end, first we need to introduce some machinery, called *arithmetic on NTM.* Let $N_1$ and $N_2$ be two NTM's. Let $n_1(x)$ and $n_2(x)$ be the number of accept paths of $N_1$ and $N_2$ on an input $x$. We define two new NTM's $N_+(x, y)$ and $N_*(x, y)$ such that $n_+(x, y) = n_1(x) + n_2(y)$ and $n_*(x) = n_1(x) * n_2(y)$. We can define $N_+$ on input $(x, y)$ as follows:

1. non-deterministically choose 1 or 2.
2. if 1 then run $N_1(x)$
3. if 2 then run $N_2(y)$

Machine $N_*$ is defined as follows:

1. run $N_1(x)$
2. if **accept**
   (a) run $N_2(y)$
   (b) if **accept** then **accept** else **reject**.
3. else **reject**

Here $TIME(N_+(x, y)) = \max\{TIME(N_1(x)), TIME(N_2(y))\} + 1$ and $TIME(N_*(x, y)) = TIME(N_1(x)) + TIME(N_2(y))$. Now we can observe that using the constructions of $N_+$ and $N_*$, for any polynomial family $P_n(a)$ of degree $poly(n)$ with positive coefficients at most $2^{poly(n)}$, we can take any machine $N$ that has $n(x)$ accept states and conform it to a machine $N_P(x)$ that has $P_{|x|}(n(x))$ accept states and has polynomial running time (we can construct the monomials $X^i$ by $N_*$ and the coefficients by $N_+$).

We can consider NTM's by circuits. Assume we have two circuits $C_1, C_2(C_i(\cdot) = M_i(w_i, \cdot))$ taking $n$-bit inputs and accepting $n_1$ and $n_2$ inputs respectively. We can observe that circuit $C_+$ given by $C_+(x, y) = C_1(x) \wedge C_2(x)$ accepts $n_1 \cdot n_2$ inputs and circuit $C_*$ given by $C_*(x, y, b) = (b \wedge C_1(x)) \vee (\bar{b} \wedge C_2(x))$ has $n_1 + n_2$ accepting inputs. In the rest of the lecture, we use the circuit model.

**Lemma 11.3** *We can amplify $BP \cdot \oplus \cdot P$.*

**Proof** For simplicity, we assume the error is one-sided. Let $L \in BP_y \cdot \oplus_z \cdot P$.

- If $x \in L$ then for all $y$'s, there is an odd number of $z$'s for which $C(x, y, z) = 1$.
- If $x \notin L$ then for at most $1 - \frac{1}{poly(n)}$ fraction of $y$'s there is an odd number of $z$'s for which $C(x, y, z) = 1$.

Now for amplification, choose $y_1, y_2, \cdots, y_m$ at random where $m$ is polynomial in $n$. Now we can observe that $\Pi_{i=1}^m(\#_{z_i}C(x, y_i, z_i) = 1)$ is odd iff $\forall i$, *the number of* $z_i$*'s for which* $C(x, y_i, z_i) = 1$ is odd. Here we can construct such a polynomial using the concept of arithmetic on NTM introduced above. Here if $x \in L$ then the probability that for all $y_i$'s, we get an odd number of $z$'s is 1. On the other hand, if $x \notin L$ with probability at most $(1 - \frac{1}{poly(n)})^m$ we get an odd number of $z$'s for all $y_i$'s. Now if $m = n \cdot poly(n)$ then the probability is exponentially small in $n$.

Now we consider a slightly harder case. Again let $L \in BP_y \cdot \oplus_z \cdot P$ such that,

- If $x \in L$ then for at most $1 - \frac{1}{poly(n)}$ fraction of $y$'s, there is an odd number of $z$'s for which $C(x, y, z) = 1$.
- If $x \notin L$ then for every $y$, there is an even number of $z$'s for which $C(x, y, z) = 1$.

The main idea here is that we complement parities, take product and complement the result. More precisely, we choose $y_1, y_2, \cdots, y_m$ at random where $m$ is polynomial in $n$. Now we observe $1 + \Pi_{i=1}^m(1 + \#_{z_i}C(x, y_i, z_i) = 1)$ is odd iff $\forall i$, *the number of* $z_i$*'s for which* $C(x, y_i, z_i) = 1$ is even. We can observe that if $x \notin L$ then our error probability is zero and if $x \in L$ the error probability is at most $(1 - \frac{1}{poly(n)})^m$ which is exponentially small in $n$ when $m = n\ poly(n)$.

Strictly speaking, in our above arguments, we need to consider the case where error is *almost one-sided* (e.g. accept with probability $1 - exp(-n)$ vs. $1 - 1/poly(n)$.) However almost nothing changes in the proof. ■

Finally, we prove Step 5 of Toda's proof.

**Theorem 11.4** $BP \cdot \oplus \cdot P \subseteq P^{\#P}$.

**Proof** Let $L$ be a language in $BP_y \cdot \oplus_z \cdot P$ where $y \in \{0,1\}^m$. First we introduce $P_n(a)$, a family of polynomials, whose degree is $poly(n)$ and whose coefficients are at most $2^{poly(n)}$ satisfying the following properties:

1. $P_n(a) = 0 \mod 2^{2^m}$ if $a = 0 \mod 2$.
2. $P_n(a) = -1 \mod 2^{2^m}$ if $a = 1 \mod 2$.

In fact, $P_n$ can be constructed as follows. Let $h(x) = 3x^4 + 4x^3$. We can easily check that $x = 0 \mod 2^m \rightarrow h(x) = 0 \mod 2^{2m}$ and $x = -1 \mod 2^m \rightarrow h(x) = -1 \mod 2^{2m}$ (just plug in $x = 0$ and $x = -1 + a2^m$ in $h(x)$). Now, we define

$$h^1(x) = h(x)$$
$$h^c(x) = h^{c-1}(h(x))$$

and let $P_n(a) = h^{\lceil \log 2m \rceil}(a)$. We can check that $P_n(a)$ has all aforementioned properties and its degree is polynomial in $m$, which is also polynomial in $n$ ($|y|$ is polynomial in $n$). We turn back to the statement of the theorem. Let $L = BP_y \cdot \oplus_z \cdot L'$. Using amplification mentioned in previous lemma, we know

1. if $x \in L$, then $Pr_y[|\{z : L'(x, y, z) = 1\}|\ \textit{is odd}] > 3/4$; and
2. if $x \notin L$, then $Pr_y[|\{z : L(x, y, z) = 1\}|\ \textit{is odd}] < 1/4$.

Thus to decide whether $x \in L$ or not, we only need to distinguish whether $Pr_y[|\{z : L(x, y, z) = 1\}|\ \textit{is odd}]$ is more than $3/4$ or less than $1/4$.

To distinguish these two in $P^{\#P}$, we compute $\sum_y P_n(\sum_z \#C(y, z))$. Now for a fixed $y$, the value of $P_n(\sum_z \#C(y, z))$ is either 0 or $-1 \mod 2^{2^m}$. Because of the definition of $P_n$, we can count the number of $y$'s for which the value is -1. Now we can check whether $Pr_y[|\{z : L(x, y, z) = 1\}|\ \textit{is odd}]$ is more than $3/4$ or less than $1/4$ by only one query of $\#P$. Here the expression $P_n(a)$ is a one-variable polynomial, and its degree is polynomial in $n$. Therefore using the concept of arithmetic on NTM, $P_n(a)$ is computable in polynomial time. ■

The rest of the proof of Toda's theorem is just putting Steps 1–5 together and using a simple induction.

# Chapter 12

**6.841/18.405J: Advanced Complexity Theory** April 1, 2002

Lecture 12

*Lecturer: Madhu Sudan* *Scribe: Grant Wang*

In this lecture, we introduce the concept of interaction in computation, discuss the motivation, and enumerate a few results.

## 12.1 Motivation for interaction

The motivation for interaction came from the field of cryptography. Consider the following scenario:

Suppose a user $X$ wishes to log into a computer $C$ remotely, where he has stored private data. Here, the user wishes to convince $C$ that he is indeed $X$, and $C$ wishes to verify that the user is $X$. The computer and the user talk back and forth until $C$ is either sure the user is $X$ or not. In such a situation, the user and the computer desire the following property: even if an eavesdropper $E$ has overhead the conversation between $X$ and $C$, $E$ cannot later succeed in logging on as user $X$.

Note that typical password security mechanisms fail to uphold such a property, e.g. `telnet`.

We can view such an interaction as a *proof of knowledge*: User $X$ wants to *prove* her identity to $C$, yet does not want to reveal any other knowledge about herself. The challenge/response scheme that is the conversation between $X$ and $C$ is an *interactive proof.*

## 12.2 Definition of Interaction

More concretely, an interactive proof consists of two parties, a prover $P$ and a verifier $V$, and a string $w$. The verifier is trying to verify some fact of $w$, and the prover $P$ is attempting to prove this to the verifier about $w$. The verifier $V$ is allowed to flip random coins that only he can see, which we model here by a random string $R$. Initially, he computes a function of both $w$, $R$ denoted by $q_1 = q_1(w, R)$, which he sends to the prover $P$. The prover receives $q_1$, and computes its own function of $w$ and $q_1$, denoted by $a_1 = a_1(w, q_1)$. This continues for $k$ rounds, until $V$ receives $a_k$ from the prover $P$. $V$ then computes a verdict function of the common knowledge $w$, random string $R$, and the answers $a_1 \ldots a_k$ from the prover, i.e. $V$ computes $verdict(w, R, a_1 \ldots a_k) = 0/1$. If the verifier is convinced, the verdict will be 1, and if the verifier is not convinced, the verdict will be 0.

Formally, a verifier is a collection of functions $(q_1(\cdot), \ldots q_k(\cdots))$, with a verdict function $verdict(\cdot)$, where each of the functions can be computed in probabilistic polynomial time, and a prover is a collection of functions $(a_1(\cdot)\dots a_k(\cdot))$, which are each computationally unbounded. We can think of the language accepted by a fixed verifier $V$. We say that $V$ accepts a language $L$ with completeness $c$ and soundness $s$, if:

$$\begin{aligned} w \in L &\Rightarrow \exists P \text{ s.t. } \Pr_R\left[verdict(P \leftrightarrow V) = 1\right] \geq c \\ w \notin L &\Rightarrow \forall P \text{ s.t. } \Pr_R\left[verdict(P \leftrightarrow V) = 1\right] \leq s \end{aligned}$$

## 12.3 Interactive Proofs for Graph Non-isomorphism

Let $\pi : \{1 \dots n\} : \{1 \dots n\}$ be a permutation relabeling the vertices of a graph $G$. We write such a relabeling as $\pi(G)$. We say that a graph $G$ is isomorphic to $H$ if there exists a permutation $\pi$ such that $\pi(G) = H$, i.e. $G$ and $H$ are really the same graph under a different labeling.

Clearly, it is easy to prove that a graph $G$ is isomorphic to a graph $H$: I can give you the permutation $\pi$, and you can check that $\pi(G) = H$. But how can I prove to you that $G$ and $H$ are non-isomorphic? There is a clever interactive proof for graph non-isomorphism.

Given graphs $G$ and $H$, the verifier picks a random permutation $\pi$, and chooses $F$ to be $G$ or $H$ randomly. Then, he applies $\pi(F)$ and sends this to the prover. The prover wants to show the verifier that $G$ and $H$ are non-isomorphic. The prover sends back $F' \in \{G, H\}$, which is his guess as to which graph $F$ the verifier picked. The verifier accepts if $F = F'$.

Note that the completeness is 1. If the graphs are non-isomorphic, the prover that correctly identifies $F$ to be isomorphic to either $G$ or $H$ will force the verifier to always accept. Note that the prover is computationally unbounded, so he can achieve this by trying all permutations $\pi$ and applying them to $F$ and determining if $\pi(F) = G$ or $\pi(F) = H$.

The soundness is $\frac{1}{2}$. If the graphs are isomorphic, the prover cannot identify $F$ to be $G$ or $H$ since $F = \pi(G) = \pi'(H)$. The best he can do is guess $G$ or $H$ at random, which will be correct $\frac{1}{2}$ of the time.

## 12.4 Resources for Interactive Proofs

What are the resources that affect the computational power of interactive proofs? We have already stated that $V$ consists of probabilistic, polynomial-time computable functions, and that $P$ is computationally unbounded. We also have other resources:

- Number of rounds of interaction: 1, constant, polynomially many rounds?
- One-sided error vs. two-sided error
- Secrecy: private coins vs. public coins

Of these three resources, secrecy seems to be essential. Consider the interactive proof for graph nonisomorphsim above. If the verifier allowed his random choice of $G, H$ to be known to the public, the prover could always answer correctly. Later, we will see the surprising result that public coins are equivalent to private coins.

## 12.5 Interactive Proofs and Arthur-Merlin games

The notion of interaction was developed independently by Goldwasser, Micali, and Rackoff (here it was called interactive proofs) and Babai (here known as Arthur-Merlin games). Whereas the motivation for Goldwasser, Micali, and Rackoff was cryptographic, the motivation for Babai was complexity-theoretic.

The key difference between Arthur-Merlin games and interactive proofs (as defined by Goldwasser, Micali and Rackoff) is public coins vs. private coins. In Arthur-Merlin games, all the coins that the verifier (Arthur) flips randomly are known to the prover (Merlin).

For interactive proofs, we write IP[$k(\cdot)$] to be the class of languages whose verifier has $k(n)$ rounds of interaction with completeness $\frac{2}{3}$ and soundness $\frac{1}{3}$. Definitionally, IP=IP[poly($\cdot$)].

For Arthur-Merlin games, we write AM to be the class of languages decided when Arthur flips some random coins, Merlin computes an answer $a$, and Arthur chooses to accept or reject deterministically. We write AM[$k(\cdot)$] to be $k(n)$ rounds where Arthur flips random coins, and Merlin computes an answer $a$ dependent on those random coin flips. We write MA to stand for the class of languages accepted when Merlin first speaks, and then Arthur decides to accept or reject when he can flip coins. Similarly define AMAMA, and AM.

## 12.6 Relations between complexity classes

It turns out that in interactive proofs (and Arthur-Merlin games), one round of interaction is equivalent in power to any constant number of rounds. We believe that poly($n$) number of rounds is greater in power, because if poly($n$) rounds was equivalent to a constant number of rounds, the polynomial hierarchy collapses.

The surprising result, due to Goldwasser and Sipser, is that private coins are equivalent to public coins, i.e. AM[$k(\cdot)$] = IP[$k(\cdot)$]. Definitionally, however, we refer to AM as interactive proofs with public coins in one-round, and IP as interactive proofs with private coins in polynomially many rounds.

## 12.7 Results for Arthur-Merlin games

**Lemma 12.1** *$AM[k] = AM$, for all constants $k$.*

**Proof Idea** The proof idea behind this is one we've already seen. We can think of Arthur-Merlin games in terms of operators on complexity classes, as we did in the proof of Toda's theorem. That is, $AM = BP \cdot \exists \cdot P$, and $AMAM = BP \cdot \exists \cdot BP \cdot \exists \cdot P$. If we can show that $\exists \cdot BP \cdot \exists \cdot P \subseteq BP \cdot \exists \cdot \exists \cdot P$, we will have that AMAM $\subseteq$ AM, and proceeding by induction, we can prove the claim. ∎

**Lemma 12.2** *$AM[k(n)] = AM[\frac{k(n)}{c}]$, for all constants $c$*

This is analogous to the speedup theorem for Turing machines: we can always decrease the number of rounds by a constant factor without losing any power.

**Lemma 12.3** *AM with 2-sided error = AM with 1-sided error*

## 12.8 IP vs. AM

We have the following results relating IP and AM:

**Theorem 12.4 (Goldwasser-Sipser)** *$AM[k(\cdot)] = IP[k(\cdot)]$, for all $k(\cdot)$*

This is the surprising result that public coins are equivalent to private coins.

**Theorem 12.5** *$IP \subseteq PSPACE$*

**Theorem 12.6** *$IP = \overline{IP}$*

**Theorem 12.7** *$PSPACE \subseteq IP$*

Note that by IP = PSPACE, we get that IP = $\overline{\text{IP}}$, since we know PSPACE is closed under complement. It is not known how to prove that IP is closed under complement without IP=PSPACE. The result that PSPACE $\subseteq$ IP was obtained in the 90's in two papers, one by Lund, Fortnow, and Nisan and the other by Shamir. As a consequence of IP=PSPACE, Condon, Feigenbaum, Lund, and Shor show that playing (in a solitaire manner) Mah-Jong (where tiles are stacked up randomly, and one can win by removing all tiles through the single action of removing matching pairs of tiles) is equivalent in difficulty to playing interactively the optimal GO player.

We will see these results in the next few lectures.

# Chapter 13

| **6.841/18.405J: Advanced Complexity Theory** | April 3, 2002 |
|---|---|
| Lecture 13 | |
| *Lecturer: Madhu Sudan* | *Scribe: Jason Hickey* |

In this lecture we will show the following:

1. IP $\subseteq$ PSPACE.
2. IP[poly] $\subseteq$ AM[poly]
3. IP[$k$] $\subseteq$ AM[$k$]

## 13.1 IP and PSPACE

Here we will show the "easier" direction of the proof that IP = PSPACE. That is, the proof that IP $\subseteq$ PSPACE. For concreteness, we will use $c = 2/3$ and $s = 1/3$ for the completeness and soundness of IP respectively. The idea is that for a fixed verifier $V$ for some language $A \in$ IP and some string $w$, we can compute in polynomial space whether or not there exists a prover $P$ such that:

$$\Pr_R[verdict(P \leftrightarrow V) = 1] \geq 2/3$$

This probability is the definition of $x \in A$. This will be done by comupting the greatest probability of acceptance inductively over the rounds of message history. If this probability is greater than 2/3, then such a prover exists.

**Theorem 13.1** IP $\subseteq$ PSPACE

**Proof** Let $Acc(w, q_1, q_2, ..., q_i, a_1, a_2, ..., a_i)$ be the acceptance probability of a message stream that has been specified up to the $i$th stage.

It is clear that we can compute $Acc(w, q_1, q_2, ..., q_n, a_1, a_2, ..., a_n)$ where the interaction has a total of $n$ stages, as the function is completely specified.

Assuming that we have computed the probability of accepting from the $i + 1$ stage of the message stream, we can compute the probability of the $i$th stage as follows. Given all questions so far, we can compute the set $S$ of all random strings $R$ that generate the message stream up to that point. For each $R \in S$, we can figure out what $q_{i+1}$ will be. So, we can compute:

$$\max_{a_{i+1}}[Acc(w, q_1, q_2, ..., q_{i+1}, a_1, a_2, ..., a_{i+1})]$$

Thus, we can compute $Acc(w)$ in polynomial space, and this is exactly $\Pr_R[verdict(P \leftrightarrow V) = 1]$. Therefore, $A \in$ PSPACE. ■

## 13.2 IP[poly] $\subseteq$ AM[poly]

IP corresponds to the case where the prover does not have access to the results of the random choices the verifier makes. AM is the case where the prover can access the random choices. The task is to show that making the choices private does not add any power.

**Theorem 13.2** IP[poly] $\subseteq$ AM[poly]

**Proof** First, we will fix an IP verifier $V$ and an input string $w$. We will consider an "interaction tree" for an interactive proof. This tree consists of nodes corresponding to the history of the interaction up to a given point and edges connect these nodes to nodes that represent immediate successors to this history. The leaves of this tree will be labelled accept or reject depending on whether the interaction corresponding to the path from the root of the tree to that leaf accepted or rejected $w$. We can assume without loss of generality that questions are bits. (A question can be converted into binary and then sent one bit at a time). Also, it can be assumed that each path through the tree corresponds to a unique random string. (It is possible to assure this by adding questions that depend specifically on the random string).

We will define $N_\sigma$ to be the number of accepting leaves in a the subtree of the interaction tree rooted at $\sigma$. Let the root of the tree be denoted *start*. The goal is to find $N_{start}$.

The goal of the AM Verifier (Arthur) is to verify that $N_{start}$ is at least $2k/3$, where $k$ is the number of random strings. At a given node $r$ with children $r_0$ and $r_1$ in the tree, the Prover (Merlin) will send Arthur $M_r$, $M_{r_0}$, and $M_{r_1}$. Arthur wants to verify that $M_r = N_r$, $M_{r_0} = N_{r_0}$, and $M_{r_1} = N_{r_1}$. Arthur does this by checking $M_r = M_{r_0} + M_{r_1}$ and recursively verifying $M_{r_0}$ or $M_{r_1}$. It is clear that Arthur cannot verify both children of every node in a polynomial number of steps, so Arthur must only choose one path to explore. Arthur picks the node to explore as follows: pick node $r_0$ with probability $M_{r_0}/(M_{r_0} + M_{r_1})$, and pick node $r_1$ otherwise. The completeness and soundness claims that follow establish the validity of this method. ■

The completeness of this method is 1 because there is zero chance of picking a node with value 0.
For the soundness,

$$Pr[\text{accepting at a node } \sigma] \leq \frac{N_\sigma}{M_\sigma}$$

This can be proved inductively. If $\sigma$ is a leaf, it clearly holds because $N_\sigma = 0$ or 1.
Assume that the claim holds for the children $\sigma_0$ and $\sigma_1$ of $\sigma$. Then,

$$Pr[\text{accepting at } \sigma] = \frac{M_{\sigma_0}}{M_{\sigma_0} + M_{\sigma_1}} Pr[\text{accepting at } \sigma_0] + \frac{M_{\sigma_1}}{M_{\sigma_0} + M_{\sigma_1}} Pr[\text{accepting at } \sigma_1]$$

By the inductive hypothesis,

$$\leq \frac{M_{\sigma_0}}{M_{\sigma_0} + M_{\sigma_1}} \frac{N_{\sigma_0}}{M_{\sigma_0}} + \frac{M_{\sigma_1}}{M_{\sigma_0} + M_{\sigma_1}} \frac{N_{\sigma_1}}{M_{\sigma_1}} = \frac{N_\sigma}{M_\sigma}$$

(The above theorem is due to [Goldwasser-Sipser] and [Furer-Goldreich-Mansour-Sipser-Zachos]. The proof is due to [Kilian].)

## 13.3 $\text{IP}[k] \subseteq \text{AM}[k]$

First, we will introduce a protocol for approximate set size that will be used in the proof of $\text{IP}[k] \subseteq \text{AM}[k]$.

The problem is as follows:
Suppose $S \subseteq \{0,1\}^n$ and has size either $|S| \geq \text{BIG} = 2^m$ or at most $\text{SMALL} = \frac{2^m}{100}$, where $m$ is on the order of $\sqrt{n}$. Also, Arthur can test membership of $S$. The question is, can Merlin convince Arthur that $S$ is BIG? The protocol for doing this is called the Goldwasser-Sipser protocol (GS):

- Merlin picks a hash function $h : \{0,1\}^n \to \{0,1\}^{m-4}$ and it sends to Arthur.
- Arthur pics $y \in \{0,1\}^{m-4}$ and sends it to Merlin.
- Merlin responds with $x \in S$ such that $h(x) = y$.

Soundness:
If $|S| \leq \frac{2^m}{100}$, then for any $h$, at most

$$\frac{2^m/100}{2^{m-4}} = \frac{16}{100} \leq \frac{1}{3}$$

of the $y$'s will have an $x \in S$ such that $h(x) = y$.

Completeness (sketch):
The idea is that we expect 16 elements of $S$ to map to a given $y$. Pairwise independence implies that any fixed $y$ is in the range of an $h$ with probability $9/10$. Markov's inequality implies that the number of $y$'s covered is $\geq 2/3$ with probability $2/3$.

**Theorem 13.3** $\text{IP}[k] \subseteq \text{AM}[k]$

**Proof** We will only prove $\text{IP}[1] \subseteq \text{AM}[O(1)]$, but extension to arbitrary $k$ follows similarly.
We will provide an $AM[O(1)]$ protocol to decide an aribtrary language in IP[1]. The protocol is as follows:

- Fix a verifier $V$ with completeness $2/3$ and soundness $1/\text{poly}$, and an input $w$.
- Let $Q = \{1, ..., q_i, ...\}$ be the set of all possible questions and $A = \{1, ..., a_i, ..\}$ is the set of all possible answers.
- For all $q \in Q$ and $a \in A$, let $S_q^a$ be the set of all random strings $R$ such that $V(R,w) = q$ and $V(R,w,a) = \text{accept}$. Let $a_q^*$ be the answer that maximizes $S_q^a$.
- Let $r$ be the length of random strings.
- So,

$$\sum_{q \in Q} \left| S_q^{a_q^*} \right| = (\text{probability of acceptance}) * 2^r$$

- Assume for simplicity, that $\left| S_q^{a_q^*} \right| = 0$ or $2^l$ or for every $q$. Now Arthur needs to be convinced that $\exists \frac{2}{3} * 2^{r-l}$ $q$'s such that $|S_q| \geq 2^l$.
- $Q = Q_0 \bigcup Q_1 ... \bigcup Q_r$, where $Q_i = \{q | 2^i \leq |S_q| \leq 2^{i+1}\}$.

- Since,

$$\sum_{q \in Q} \left| S_q^{a_q^*} \right| \geq \sum_{i=1}^{r} 2^i |Q_i| \geq \frac{2^r}{3}$$

  Only the last inequality needs to be verified.

- It needs to be verified that $\exists i$ such that $2^i |Q_i| \geq \frac{2^r}{3r}$.
- Now run 2 GS protocols one after the other.
- Merlin will prove $|Q_i| \geq \frac{2^r}{3r * 2^i}$.
- Merlin sends $h$, Arthur queries with $y$ and Merlin sends $q \in Q_i$ such that $h(q) = y$ (This is the first GS protocol).
- Arthur must now verify that $|S_q| \geq 2^l$. Run another GS protocol to achieve this.

Thus, only a constant number of rounds is needed to decide a problem in IP[1]. ■

# Chapter 14



| 6.841/18.405J: Advanced Complexity Theory | April 08, 2002 |
|---|---|
| Lecture 14 | |
| *Lecturer: Madhu Sudan* | *Scribe: Dah-Yoh Lim* |



In this lecture, we will show that $IP \supseteq PSPACE$; together with $IP \subseteq PSPACE$ (proved last lecture), this concludes our prove that $IP = PSPACE$. Actually, we will first prove that $\#P \subseteq IP$ , then introduce straight line programs of polynomials to aid us in proving $IP \supseteq PSPACE$.

## 14.1 $\#P \subseteq IP$

Suppose that we have a formula $\phi$, and we wish to count the number of satisfying assignments, $\#\phi$, by setting up an interactive protocol. If the prover claims that $\#\phi = N$, how can we check it? Using the self reducibility of $SAT$, we can progress down the self reducibility tree, and try to verify consistency at every level. Let $\phi_0$ denote the formula $\phi$ with its first variable set to 0, i.e. $\phi_0 = \phi(x_1 = 0)$; likewise, let $\phi_1 = \phi(x_1 = 1)$. Say that the prover tells us that $\phi_0 = N_0$ and $\phi_1 = N_1$. Therefore, the verifier should try to check that $\#\phi = N$ by checking that $N = N_0 + N_1 = (N_{00} + N_{01}) + (N_{10} + N_{11})$ and so on, until it reaches the nodes where all the varibles have been assigned, at which point it can directly check the prover's claim by evaluating the formula. The problem is that the tree is exponential, so the polynomial-timed verifier cannot walk down the whole tree.

The way out of this is to change the logical computations to arithmetic computations, so that we can somehow combine the checks. This is done by setting up an arithmetic way of looking at $\#SAT$.

### 14.1.1 Arithmetizing SAT

Consider the following correspondence:

| Boolean Constructs | | Arithmetic Polynomials |
|---|---|---|
| variables: | $x_i$ | $z_i$ |
| literals: | $x_i$, $\neg x_i$ | $z_i$, $(1 - z_i)$ |
| clauses: | $C_l = x_i \vee x_j \vee x_k$ | $P_l = 1 - (1 - z_i)(1 - z_j)(1 - z_k)$ |
| formulae: | $\phi(x_1, ..., x_n) = \wedge_{l=1}^{m} C_l$ | $Q(\mathbf{z}) = \Pi_{l=1}^{m} P_l(\mathbf{z})$ |

Note:

1. For our purposes, we can think of the arithmetic as done over $\mathbb{Z}$, $\mathbb{Z}_p$, or $\mathbb{F}$.
2. For $\mathbf{a} \in \{0, 1\}^n$, $Q(\mathbf{a}) = 1$ iff $\mathbf{a}$ satisfies $\phi$.

3. $Q$ is a polynomial of total degree $\leq 3 \cdot m$.
4. $\#\phi = \Sigma_{\mathbf{a} \in \{0,1\}^n} Q(\mathbf{a})$

### 14.1.2 $\#SAT \in IP$

So now, instead of working on the $N_\alpha$s directly (where $\alpha \in \{0,1\}$), we can work with the analogous $Q_\alpha$s (where $\alpha \in \{0,...,p-1\}$); all calculations are done modulo $p$. We should think of $p$ as a very large prime (we will determine its value later). This generalizes our previous self-reducibility tree, a binary one, to a $p$-nary tree. At the root node we have $Q_\lambda \stackrel{\text{def}}{=} \Sigma_{\mathbf{a} \in \{0,1\}^n} Q(\mathbf{a})$ ; at level one, we have $Q_0 \stackrel{\text{def}}{=} \#\phi = \Sigma_{a_2,...a_n \in \{0,1\}} Q(0, a_2, ..., a_n)$, $Q_1 \stackrel{\text{def}}{=} \#\phi = \Sigma_{a_2,...a_n \in \{0,1\}} Q(1, a_2, ..., a_n)$, and in general, $Q_\alpha \stackrel{\text{def}}{=} \#\phi = \Sigma_{a_2,...a_n \in \{0,1\}} Q(\alpha, a_2, ..., a_n)$.

Suppose the prover claims that $Q_\alpha = \#\phi = N$, and it gives $Q_0 = N_0, Q_1 = N_1, ..., Q_{p-1} = N_{p-1}$ to support its claim. Consider the polynomial $p(x) = \Sigma_{a_2,...,a_n} Q(x, a_2, ..., a_n)$, a univariate function of $x$ (the other variables are being summed over); $p(x)$ is still a polynomial of degree $\leq 3 \cdot m$, because it is just a sum of polynomials all of degree $\leq 3 \cdot m$.

The protocol starts as follows:

1. The prover gives $Q_\lambda, Q_0, Q_1$
2. The verifier verifies that $Q_\lambda \leq 2^n$ and rejects if not. The verifier asks for the polynomial $p(x) \stackrel{\text{def}}{=} \Sigma_{a_2,...,a_n} Q(x, a_2, ..., a_n)$.
3. The prover responds with $h(x)$ (by sending the coefficients).
4. The verifier verifies that $h(x)$ is of degree $\leq 3 \cdot m$, $Q_0 = h(0)$, $Q_1 = h(1)$, and $Q_\lambda = Q_0 + Q_1 (mod p)$; if any one fails, it rejects. Now it picks a random $\alpha \in \mathbb{Z}_p$ and sents it to the prover, asking it to prove that the polynomial $p(\alpha)$ as defined would evaluate to $Q_\alpha$.

Recursively:

2j. The verifier is trying to verity that $Q_\mathbf{b} = \Sigma_{\mathbf{s} \in \{0,1\}^{n-i}} Q(\mathbf{b}, \mathbf{s})$, where the vector $\mathbf{b} = b_1 b_2 \cdots b_i$ represents the sequence of choices that the verifier made (at random) from the root to the current node. Now, the verifier asks for the polynomial $p(x) \stackrel{\text{def}}{=} \Sigma_{\mathbf{s}' \in \{0,1\}^{n-i-1}} Q(\mathbf{b}, x, \mathbf{s}')$.

2j+1. The prover responds with a $h(x)$.

2j+2. The verifier verifies that $h(x)$ is of degree $\leq 3 \cdot m$ and $Q_\mathbf{b} = h(0) + h(1) (mod p)$; if any one fails, it rejects. Now it picks a random $\alpha \in \mathbb{Z}_p$ and sents it to the prover, asking it to prove that the polynomial $p(\alpha)$ as defined would evaluate to $Q_{\mathbf{b}\alpha}$.

... $\vdots$

2n+4. At the end, the verifier can verify directly the claims, since $Q_\mathbf{a} = Q(\mathbf{a})$, where $\mathbf{a}$ is the vector $\mathbf{a} = a_1 a_2 \cdots a_n$.

**Proof of Correctness of the protocol**

So far, we have only outlined the protocol, without any claims- this allows us to separate the protocol from its proof.

**completeness** This is quite obvious as the correct prover can start with the correct value of $\#\phi$, and at all iterations, the prover can send the correct polynomial. Certainly then, every (local consistency) check that the verifier makes will work out, so it accepts with probability 1.

**soundness** Without loss of generality assume that the prover always responds with $h(\cdot)$ s.t. $h(0)+h(1) = Q_{\mathbf{b}}$ at the $i$-th level, because or else the verifier would have rejected right away. Also, we assume that $Q_\lambda \neq \Sigma_{\mathbf{a}\in\{0,1\}^n} Q(\mathbf{a})$, i.e. it is a crooked prover that is trying to prove something that is false to the verifier.

**Claim 14.1** *If $Q_\lambda \neq \Sigma_{\mathbf{a}\in\{0,1\}^n} Q(\mathbf{a})$ then the inequality holds (mod p) with probability $\geq \frac{1}{10}$ provided that $p \in_R [n^2, 2n^2]$ or $[10mn, 20mn]$, which ever is larger.*

**Proof** $Q_\lambda - \Sigma_{\mathbf{a}\in\{0,1\}^n} Q(\mathbf{a})$ has at most $n \leq \frac{n^2}{10logn}$ prime factors for $p$ in the given range.■

**Claim 14.2** *Suppose $Q_{\mathbf{b}} \neq \Sigma_{\mathbf{s}} Q(\mathbf{b}, \mathbf{s})$; then, $Q_{\mathbf{b},\alpha} \neq \Sigma_{\mathbf{s}'} Q(\mathbf{b}, \alpha, \mathbf{s}')$ with a certain probability over $\alpha$.*

**Proof** $p(x) \stackrel{\text{def}}{=} \Sigma_{\mathbf{s}'} Q(\mathbf{b}, \alpha, \mathbf{s}')$; we know that $p(0) + p(1) = \Sigma_{\mathbf{s}} Q(\mathbf{b}, \mathbf{s}) \neq Q_{\mathbf{b}} = h(0) + h(1)$, i.e. $p(0)+p(1) \neq h(0)+h(1)$, so $p(\cdot) \neq h(\cdot)$. Since they are polynomials of degree $\leq 3 \cdot m$, by the Schwartz lemma we have that for random $\alpha$, $p(\alpha) \neq h(\alpha)$ with probability 1-$\frac{3m}{p}$. In each iteration, the prover has probability $\frac{3m}{p}$ of getting away with lying; by the union bound the probability that the prover successfully cheats the verifier is $\leq \frac{3mn}{p} \leq \frac{3}{10}$.■

Combining the above two claims, we see that the soundness is $\frac{4}{10}$.

## 14.2 Abstracting the Proof

Note that what helped us in the above proof is not some specific features of $\#P$. What we used is essentially the downward self-reducibility of the language to reduce a complicated property to a collection of progressively less complicated properties. So we could apply the same idea to all languages in $PSPACE$ (which is exactly the collection of self-reducible languages). The arithmetization allows us to effectively compress questions down to one question, without requiring any structure on the questions. Below we look at how we can extend the compression.

### 14.2.1 Extending Compression: low-degree curves

Suppose that computing $Q_{\mathbf{b}}(\mathbf{x})$ involves computing $Q'_{\mathbf{b}}(\mathbf{x_0})$ and $Q'_{\mathbf{b}}(\mathbf{x_1})$, where $\mathbf{x_0}$ and $\mathbf{x_1}$ are not related.

Consider lines in an $n$-dimensonal integer grid: $l : \mathbb{F} \to \mathbb{F}^n$ Geometrically a line is... a line. Algebraically, it is a collection of $n$ functions, each of which is a degree 1 polynomial; the $i$-th function gives the $i$-th coordinate. For any two points $\mathbf{x_0}$ and $\mathbf{x_1}$, there is a line through the two points. In other words, $\exists l$ s.t. $l(0) = \mathbf{x_0}$ and $l_1 = \mathbf{x_1}$; specifically, $l(t) = (1-t)\mathbf{x_0} + t\mathbf{x_1}$.

Since $Q'$ maps $\mathbb{F}^n$ to $\mathbb{F}$, the composition of $Q'$ and $l$, $Q' \circ l : \mathbb{F} \to \mathbb{F}$, is a univariate function. What is nice about this composition is that it preserves degrees: if $Q'$ is of degree $d$, $Q' \circ l$ is of degree $\leq d$.

Now, to extend our previous protocol's capabilities, we change the protocol to:

i. The verifier wants to verify that $Q(\mathbf{x}) = a$. To do so, it generates $\mathbf{x_0}, \mathbf{x_1}$ , and a line $l$ through the two points. Now it asks the prover for $Q' \circ l$.

i+1. The prover responds with a degree $d$ univariate polynomial $h$.

i+2. The verifier checks that $h$ is a univariate polynomial of degree $d$ and rejects if not. It checks the local consistency by checking that $a = h(0) + h(1)$. Iff it is locally consistent, the verifier goes on to verify recursively that $h(\alpha)$ is correct for random $\alpha$.

Note that as we progress down the tree, the polynomials involved gets simpler- eventually at the leave nodes the verifier will be powerful enough to check the much simpler condition by itself.

### 14.2.2 Straight line programs of polynomials

The above extension motivates the following definitions.

**Definition 14.3** *$p_0, ..., p_L$ is an (n,d,L,w)-straight line program of polynomials if:*

1. *Every $p_i$ is on at most $n$ variables.*
2. *Every $p_i$ is of degree at most $d$.*
3. *$p_0$ is easy to evaluate (i.e. computable in polynomial time)*
4. *$p_i$ is easy to evaluate given oracle access to $p_{i-1}$ (i.e. there is a polynomial time algorithm $A$ that, given $i, \mathbf{x}$ and an oracle for $p_{i-1}$ can compute $p_i(\mathbf{x})$ making at most $w$ non-adaptive queries to $p_i$; $w$ is the "width" of the straigt line program).*

**Definition 14.4** *Polynomial straight line program satisfaction is the language whose members are $(< p_0, ..., p_L >, \mathbf{x}, \alpha)$ s.t. $p_L(\mathbf{x}) = \alpha$, where $\mathbf{x} \in \mathbb{Z}^n$, $a \in \mathbb{Z}$, and $< p_0, ..., p_L >$ is an $(n, d, L, w)$-straigt line program of polynomials.*

**Lemma 14.5** *Polynomial straight line program satisfaction $\in IP$ for $w = 2$.*

**Proof** The rough idea is as follows:

- The verifier picks a random prime $p \approx poly(n, d, L, log||x||)$ and sends it to the prover. Set $a_L \leftarrow a$ and $x_L \leftarrow x$.
- For $i = L - 1$ downto 0 do:
  - Let $(\mathbf{x_0})_i$ and $(\mathbf{x_1})_i$ be queries of $A$ on input $i + 1$, $(\mathbf{x})_{i+1}$. Let $l_i$ b ethe line through $(\mathbf{x_0})_i$ and $(\mathbf{x_1})_i$. The verifier asks the prover for $p_i \circ l_i$.
  - The prover responds with $h_i$.
  - The verifier verifies that $A$'s answer on oracle values $h(0)$ and $h(1)$ is $a_{i+1}$ and rejects if not. It picks a random $\alpha \in \mathbb{Z}_p$ and sets $\mathbf{x_i} \leftarrow l_i(\alpha)$ and $a_i \leftarrow h_i(\alpha)$.

At the end the verifier verifies that $h_0(\alpha) = p_0(l_0(\alpha))$. ■

**Lemma 14.6** *Polynomial straight line program satisfaction is $PSPACE$ complete.*

**Proof** The basic idea is as follows:

- We fix a PSPACE machine $M$ taking $s$ space with input $\mathbf{x}$. Let $f_i(\mathbf{a}, \mathbf{b})$ be polynomials that have configurations of $M$ (namely $\mathbf{a}$ and $\mathbf{b}$, from $\{0, 1\}^s$) as its inputs, s.t. $f_i(\mathbf{a}, \mathbf{b}) = 1$ if configuration $\mathbf{a}$ yields configuration $\mathbf{b}$ in (exactly) $2^i$ steps; $f_i(\mathbf{a}, \mathbf{b}) = 0$ otherwise. Notice that $f_0$ is a constant degree polynomial of degree $C = O(1)$ in each variable.
- $f_{i+1}(\mathbf{a}, \mathbf{b}) = \Sigma_{\mathbf{c} \in \{0,1\}^s} f_i(\mathbf{a}, \mathbf{c}) \cdot f_i(\mathbf{c}, \mathbf{b})$ is also a polynomial of degree $C$ in each variable.

Unfortunately in the above, $w \neq 2$; but we can fix that by doing summation "slowly"- we define a longer sequence:

- $g_i = g_{is} = f_i$.
- $g_{i0}(\mathbf{a}, \mathbf{b}, \mathbf{c}) = g_{i-1,s}(\mathbf{a}, \mathbf{c}) \cdot g_{i-1,s}(\mathbf{c}, \mathbf{b})$.
- $g_{ij}(\mathbf{a}, \mathbf{b}, \mathbf{c}) = g_{i,j-1}(\mathbf{a}, \mathbf{b}, \mathbf{c}0) + g_{i,j-1}(\mathbf{a}, \mathbf{b}, \mathbf{c}1)$, where $\mathbf{c} \in \mathbb{Z}^{s-j}$.

- $g$ has degree at most $C$ in the variables of $\mathbf{a}, \mathbf{b}$, and at most $2C$ in the variables of $\mathbf{c}$.

Then, we have a sequence of width $w = 2$, as required: $g_0, g_{10}, g_{11}, ..., g_{1s}, g_{20}, ..., g_{ss}$. Therefore $PSPACE$ completeness follows. ■

We have shown that a $PSPACE$-complete problem, (Polynomial straight line program satisfaction), has an $IP$, implying that $PSPACE \subseteq IP$. Actually, we can further generalize the line arguments even "wider", for $w > 2$. We will leave this as an exercise- this will give a direct proof that the permanent has an interactive proof, where the prover only needs to be able to compute the permanent.

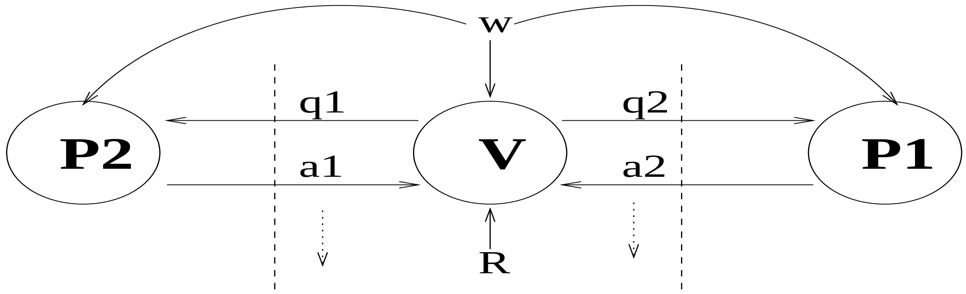
# Chapter 15

| **6.841/18.405J: Advanced Complexity Theory** | April 10, 2002 |
|---|---|
| Lecture 15 | |
| *Lecturer: Madhu Sudan* | *Scribe: David Woodruff* |

In this lecture we will cover Multiprover Interactive Proofs (MIPs), Oracle Interactive Proofs (OIPs), and Probabilistically Checkable Proofs (PCPs). We will see that 2IP, the complexity class of languages with interactive proofs with two provers, is stronger than IP in a certain cryptographic sense. We then introduce OIP to show that 2IP is equivalent to MIP, the set of languages having a polynomial number of provers. Finally, we introduce the class PCP and relate it to the previous complexity classes we have studied.

## Multiprover Interactive Proofs (MIP)

What happens if we allow the verifier to interact with more than one prover in an interactive proof? The provers have unbounded computational resources, but cannot interact with each other. If the prover wants to cheat the verifier, he has to make sure he will not be detected when the verifier interacts with the other provers. The complexity class of multiprover interactive proofs (MIP) was defined by Ben-Or, Goldwasser, Kilian and Wigderson. In the case of 2 provers we have the complexity class 2IP, shown in the following diagram:




where $\{q_i\}$ is the set of questions asked by the verifier $V$, $\{a_i\}$ is the set of answers given by provers $P_1$ and $P_2$, $w$ is the common input string, and $R$ is the string of $V$'s random coin tosses. After interacting with the provers, $V$ is required to produce a boolean verdict, Verdict$(w, R, a_1, \ldots, a_k)$. As in the definition of IP, $V$ is restricted to probabilistic polynomial time in the length of $w$.

Formally, a language $L \in$ 2IP if

- (completeness) $w \in L$ implies $\exists P_1, P_2$ such that $\Pr[P_1 \leftrightarrow V \leftrightarrow P_2 \text{ accepts }] = 1$.
- (soundness) $w \notin L$ implies $\forall P_1, P_2$, we have $\Pr[P_1 \leftrightarrow V \leftrightarrow P_2 \text{ accepts }] \leq 1/2$.

Clearly IP $\subseteq$ MIP because the verifier can choose to interact with just one prover. Moreover, it seems that by limiting the cheating possibilities of the prover in MIP, we should be able to construct more verifiers for more statements and hence MIP should be a larger class of languages than IP. In fact Babai, Fortnow, and Lund showed that MIP = NEXPTIME.

We note that 2IP is robust with respect to error in the sense that one can convert any two-sided error verifier $V$ into a one-sided error verifier $V'$. Also, one can amplify the error by repeating the above proof sequentially. In general, however, one cannot repeat the above proof in parallel without compromising soundness.

## An Application of 2IP: Zero-knowledge Proof

We give a cryptographic application of a two-prover interactive proof. Informally a *Zero Knowledge Proof* is a proof by which the prover can convince the verifier whether or not a string $x$ is in a language $L$ without giving the verifier any other information whatsoever, such as certain properties of $x$ which the verifier could not compute otherwise. Here is a two-prover protocol for graph 3-colorability:

1. $V$ picks a random edge $e = (u, v)$ in the input graph $G$. $V$ then picks a random endpoint $w$ of $e$, i.e., $w \in \{u, v\}$.
2. Next $V$ sends $e$ to $P_1$ who responds with the pair of colors $(c_1, c_2)$, where $c_1 = \mathit{color}(u)$ and $c_2 = \mathit{color}(v)$. If $c_1 = c_2$, then $V$ immediately rejects $G$.
3. $V$ then sends $w$ to $P_2$ who responds with $c_3 = \mathit{color}(w)$.
4. If $c_1 \neq c_3$ and $w = c_1$, then $V$ rejects. Similarly, if $c_2 \neq c_3$ and $w = c_2$, then $V$ rejects. Otherwise, $V$ accepts $G$.

We argue completeness, soundness, and zero-knowledge. Completeness is clear - if a graph $G$ is 3-colorable, the honest prover convinces the verifier of this with probability 1.

As for soundness, consider a graph $G = (V, E)$ which is not 3-colorable. Then there exists an edge $e$ of $G$ whose endpoints have the same color for any coloring. Suppose $P_1$ and $P_2$ have agreed upon a coloring of the vertices of $G$ before the protocol begins. Fix $P_2$. This effectively fixes a coloring of $G$. Then $V$ will choose an edge $e'$ to send to $P_1$ which will equal $e$ with probability $1/|E|$. In order for $V$ not to reject $G$ immediately, $P_1$ will have to give different colorings for the two endpoints of $e$, despite the coloring scheme $P_1$ and $P_2$ have agreed upon that assigns the same color to both endpoints. Then, when $V$ queries $P_2$ for the color of one of the randomly chosen endpoints of $e$, the color $P_2$ returns for at least one of the endpoints of $e$ will differ from the assignment given by $P_1$. Hence, the probability that $V$ will accept $G$ is at most $1 - 1/(2|E|)$. Repeating the above protocol sequentially a polynomial number of times, one can achieve a soundness of $1/2$. Note that if $|E|^2$ rounds will be run sequentially, the provers need to agree upon a sequence of $|E|^2$ color relabellings beforehand. The provers can collude with the knowledge of $G$, but must collude before $V$'s random coins are tossed.

Finally, we informally argue that the protocol is zero-knowledge. Before the beginning of the protocol the provers agree upon a random coloring of the vertices. That is to say, if $G = (V, E)$ is 3-colorable, then there exists a coloring assignment $f : V \to \{1, 2, 3\}$ such that if $(u, v) \in E, f(u) \neq f(v)$. Then if $\pi : \{1, 2, 3\} \to \{1, 2, 3\}$ is a permutation on three letters, we see that $\pi$ composed with $f$ will give another 3-coloring. Hence there are 6 random colorings of $G$ corresponding to $f$. The provers agree upon $f$ and the permutation $\pi$ before the beginning of the protocol. When $V$ learns a coloring $\mathit{color}(u)$ and $\mathit{color}(v)$ for an edge $(u, v)$, he doesn't learn anything other than the fact that $\mathit{color}(u) \neq \mathit{color}(v)$ since any combination of two colors for $u$ and $v$ is equally likely. Hence, the protocol is zero-knowledge.

## Oracle Interactive Proofs (OIP)

Clearly we have the inclusions IP $\subseteq$ 2IP $\subseteq$ 3IP $\subseteq \cdots \subseteq$ MIP, since the verifier can simply choose not to interact with some of the provers. But are polynomially many provers more powerful than 2 provers? To answer this question, Fortnow, Rompel and Sipser introduced the complexity class of Oracle Interactive Proofs (OIP).

The key difference between MIP and OIP is that the provers in Oracle Interactive Proofs are oracles, i.e., they are memoryless provers. Without loss of generality we can think of an oracle $O$ as a function from $\{0,1\}^*$ to $\{0,1\}$. Hence, the obvious definition:
$L \in$ OIP iff

1. (completeness) $w \in L$ implies $\exists O$ s.t. $\Pr[V \leftrightarrow O \text{ accepts }] = 1$
2. (soundness) $w \notin L$ implies $\forall O$ $\Pr[V \leftrightarrow O \text{ accepts }] \leq 1/2$

We first show MIP $\subseteq$ OIP. Intuitively, in OIP we are restricting to a smaller class of provers so the verifier is more likeley to accept more statements, so a language is more likely to meet the soundness criterion, so OIP is likely to be a larger class of languages than MIP.

Formally, we can simulate any Multiprover Interactive Proof with an Oracle Interactive Proof. Suppose there are $p$ provers in the Multiprover Interactive Proof. We simply convert each prover into a lookup table. If $V$ asks prover $i$ question $q_k$ with history $q_1, q_2, \ldots, q_{k-1}$, then there is an entry in the table that maps $(i, q_1, \ldots, q_k)$ to the answer $a_k$ that prover $i$ would respond with. We can create one table for all provers mapping any possible set of questions the verifier could ask to the answer given by each prover. But this is just an oracle $O$ and hence MIP $\subseteq$ OIP.

We will prove the reverse inclusion for the case when $V$ is a non-adaptive verifier, even though OIP $\subseteq$ MIP for adaptive verifiers as well. We will give a reduction but we will not formally argue completeness and soundness. Specifically, we shall argue that OIP $\subseteq$ 2IP, and together with MIP $\subseteq$ OIP, it will follow that 2IP = 3IP = $\cdots$ = MIP.

Suppose we have an Oracle Interactive Proof with verifier $V$ asking questions $q_1, \ldots, q_m$ to oracle $O$. Then we construct a verifier $V'$ in the two-prover setting which behaves as follows. It first asks the same questions $q_1, \ldots, q_m$ to prover $P_1$ and receives a sequence of answers $a_1, \ldots, a_m$. It then randomly chooses an index $j$ and sends $q_j$ to $P_2$. $P_2$ then responds with an answer $b$. Finally, $V'$ accepts if and only if $V$ would accept given answers $a_1, \ldots, a_m$ from $O$ and if $a_j = b$. Intuitively, although $P_1$ has more room to cheat than $O$ since he is not memoryless, $P_2$'s answer is used to ensure $P_1$ is not using the history of questions to base his answers on. Completeness of this protocol is clear. To see that it is sound, note both that the original oracle $O$ can cheat only with low probability and that if $P_1$ tries to cheat, he will be detected by $V$'s interaction with $P_2$ with nonnegligible probability.

Now let's compare IP and OIP = MIP. We saw in the last lecture that IP = PSPACE, developed by Lund, Fortnow, Karloff, and Nisan, and later proven by Shamir. On the other hand, Babai, Fortnow, and Lund showed that OIP = NEXPTIME. NEXPTIME can be thought of as the complexity class of languages of short theorems with long proofs and polynomial-time verification in the length of the proof. Note that the inclusion OIP $\subseteq$ NEXPTIME is pretty clear. For languages in OIP we have a probabilistic polynomial-time verifier whereas for languages in NEXPTIME we are allowed a deterministic exponential-time verifier. Moreover, we can write down the oracle as a table of (question, answer) pairs in NEXPTIME. We would like to "scale down" the equality MIP = NEXPTIME to obtain an equality of the form MIP' = NP, where verifiers in MIP' run in logarithmic time (in the length of w), but this is not possible since the verifier would not even be able to read the entire input.

Hence, we see that IP $\neq$ OIP unless PSPACE = NEXPTIME.

## Probabilistically Checkable Proofs (PCP)

We now parameterize Oracle Interactive Proofs more precisely and give them a new name, Probabilistically Checkable Proofs (PCP). In particular, we keep track of the number of coins $V$ tosses, $r(|w|)$, and the number of queries $V$ makes to $O$, $q(|w|)$. As before, we restrict $V$ to run in polynomial time. Formally,

$L \in \mathrm{PCP}_{c,s}[r, q]$ iff $\exists$ an OIP for $L$ with an $(r, q)$-restricted verifier $V$ with completeness $c$ and soundness $s$.

The following identities are immediate:

- $\mathrm{PCP}_{2/3,1/3}[\mathrm{poly}(n), 0] = \mathrm{BPP}$.
- GNI $\in$ $\mathrm{PCP}_{1,1/2}[\mathrm{poly}(n), 1]$ as shown in class.
- $\mathrm{PCP}_{1,1/2}[\mathrm{poly}(n), \mathrm{poly}(n)] = \mathrm{OIP}$.
- $\mathrm{NP} = \mathrm{PCP}_{1,0}[0, \mathrm{poly}(n)] = \mathrm{PCP}_{1,0}[O(\log(n)), \mathrm{poly}(n)]$
- $\mathrm{PCP}_{1,1/2}[O(\log(n)), O(\log(n))] \subseteq \mathrm{NP}$.

What if we try to "scale down" from NEXPTIME to NP to get proofs for NP languages? Work by Babai, Fortnow, Levin, and Szegedy, and by Feige, Goldwasser, Lovász, Safra, and Szegedy showed that NP $\subseteq$ $\mathrm{PCP}_{1,1/2}[\mathrm{poly}(\log(n)), \mathrm{poly}(\log(n))]$. Later work by Arora and Safra showed that NP $=$ $\mathrm{PCP}_{1,1/2}[\log(n), \sqrt{(\log(n))}]$, and work by Arora, Lund, Motwain, Sudan, and Szegedy showed that NP $=$ $\mathrm{PCP}_{1,1/2}[\log(n), k]$ for a constant $k$. Work by Håsted reduced the number of query bits to 3, showing that $\forall \epsilon > 0$ NP $=$ $\mathrm{PCP}_{1-\epsilon,1/2}[\log(n), 3]$, and finally Guruswami, Lewin, Sudan, and Trevisin got perfect completeness by showing $\forall \epsilon > 0$ NP $= \mathrm{PCP}_{1,1/2}[\log(n), 3]$. In other words, a proof of any statement in NP can be written in such a way that it can be verified by looking at only 3 bits of the proof.

We will now use these results to show that if NP $\neq$ P, then even approximating an NP-hard problem is very hard. Let $V$ be a verifier for a $\mathrm{PCP}_{1,1/2}[\log(n), 3]$ language $L$. For each possible sequence of the verifier's random coins $R_i$, we shall construct a decision tree corresponding to oracle $f$'s responses to $V$'s 3 queries. The decision tree is a balanced binary tree of depth 3. We start at the root node $q_1$. If $f(q_1) = 0$, we examine $q_1$'s left child, otherwise we examing $q_1$'s right child. Based on our response $f(q_1)$ and $R_i$, we can compute the next question $q_2$ asked by $V$. We then branch according to whether $f(q_2) = 0$ or $f(q_2) = 1$. We continue branching in this way until we reach a leaf node of the tree, which is either an Accept node or a Reject node, depending on whether $V$, given toin cosses $R_i$, accepts or rejects based on $f$'s responses to his questions. Each path in this decision tree can be written as a 3-CNF $\phi_i$ formula of at most 8 clauses. We do this for all polynomially many random strings $R_i$. We then define $\Phi = \bigwedge_{i=1}^{poly} \phi_i$. Completeness of $L$ implies $\Phi$ is satisfiable. Soundness implies that if input $x$ is not in $L$, then at least $1/2$ of all formulae $\phi_i$ are not satisfied for any assignment of oracle answers. For a formula to not be satisfied, at least 1 of 8 clauses must not be satisfied. Hence, $1/16$ of all clauses of $\Phi$ are not satisfied. This says that unless P = NP, even approximating hard problems is very hard.

# Chapter 16

| **6.841/18.405J: Advanced Complexity Theory** | April 17, 2002 |
|---|---|
| Lecture 16 | |
| *Lecturer: Madhu Sudan* | *Scribe: Austin Che* |

**Recall** $\mathrm{PCP}_{c,s}[r, q]$

- Verifier tosses $r(n)$ coins.
- Queries the proof oracle with $q(n)$ bits.
- Completeness $c(n)$. Omit if $c = 1$.
- Soundness $s(n)$. Zero subscripts means $c = 1$ and $s = \frac{1}{2}$.

**Last time** $\mathrm{NP} = \mathrm{PCP}_{\frac{1}{2}+\epsilon}[O(\log n), 3]$ [Håstad]

**Proposition 16.1** $\mathrm{NP} = \mathrm{PCP}_{\frac{1}{2}+\epsilon}[O(\log n), 3] \implies$ *MaxSAT is hard to approximate to within* $\frac{15}{16} + \epsilon'$.

MaxSAT is the problem of satisfying as many clauses as possible.

**Today** A weaker statement: $\mathrm{NP} \subseteq \mathrm{PCP}_{1,\frac{1}{2}}[\mathrm{poly}\log, \mathrm{poly}\log]$

1. Set up an algebraic promise problem (GapPCS).
2. Show it is NP-hard (similar to IP=PSPACE proof).
3. Give a PCP verifier for this problem.

**Constraint Satisfaction Problems**
$x_1, \ldots, x_n$ (variables)
$c_1, \ldots, c_t$ (constraints)
Find an assignment to the $n$ variables such that all (or many) of the constraints are satisfied.

*Examples:*

1. 3SAT. $x_i$ boolean. $c_j = x_{i_1} \vee x_{i_2} \vee \overline{x_{i_3}}$.
2. 3COL. $x_i$ tertiary (colors). $c_j =$ "$x_{i_1} \neq x''_{i_2}$

**Polynomial Constraint Satisfaction Problems**

$F$ is a field and $|F|^m = n$. Also, have a degree parameter $d$ and $|F| \gg d$.
In 3SAT, an assignment $A : [n] \to \{0,1\}$.
Here, assignments will be functions $f : F^m \to F$. Variables will be vectors in $F^m$.
Constraints will be $c_j = (A_j, x_1^{(j)}, x_2^{(j)}, \ldots, x_k^{(j)})$. $A_j$ is an algebraic circuit $F^k \to F$.
A constraint $c_j$ is satisfied by $f$ if $A_j(f(x_1^{(j)}), f(x_2^{(j)}), \ldots, f(x_k^{(j)})) = 0$.

**GapPCS**

We define a promise problem based on the polynomial constraint satisfaction problem.

- YES instances: $\exists f : F^m \to F$ that satisfies all constraints and $f$ is a degree $d$ polynomial.
- NO instances: $\forall f : F^m \to F$ that are of degree $d$, at least 90% of constraints are unsatisfied by $f$.

**Hardness of GapPCS**

To show that GapPCS is NP-hard, we reduce SAT on $N$ variables to GapPCS in time $|F|^m$ with:

- $k, d = (\log N)^3$
- $m \geq \frac{\log N}{\log \log N}$
- $|F| \approx (\log N)^{10}$
- $t \approx |F|^m$

Then the reduction is done in polynomial time in $N$:

$$|F|^m = \left((\log N)^{10}\right)^{\frac{\log N}{\log \log N}} = 2^{\frac{10 \log N \log \log N}{\log \log N}} = 2^{10 \log N} = N^{10}$$

**PCP Verifier for GapPCS**

1. Expect to be given a proof oracle $f : F^m \to F$.
2. Verifier tests that $f$ is close to some degree $d$ polynomial $p : F^m \to F$ (low degree testing).
3. Build an oracle computing $p : F^m \to F$" from oracle for $f : F^m \to F$ (self correction).
4. Pick random $j$ and verify $c_j$ is satisfied by $p$ (not $f$).

We note that

- Self correction can be done in time $\mathrm{poly}(m, d)$
  (Contrast with the number of coefficients of $p$: $(\frac{d}{m})^m \leq \#$ coeffs $\leq d^m$)
- Low degree testing can also be done in time $\mathrm{poly}(m, d)$.
- To verify $c_j$, the verifier needs to make $\mathrm{poly} \log N$ queries.
- We need $\log t$ random bits to select $j$, for low degree testing, we need $O(m \log |F|) = O(\log n)$ bits, and for self correction, we need $m \log |F|$ bits. This shows that the verifier uses $O(\log N)$ random bits.

**Self Correction**

- Given oracle $f : F^m \to F$ such that there exists a polynomial $p : F^m \to F$ of degree $d$ and
$$\Pr_x[f(x) \neq p(x)] \leq \delta$$
- Also given $a \in F^m$
- Compute $p(a)$. For all $a$, should be computing $p(a)$ correctly with high probability over internal randomness. We cannot just use $f(a)$ as for some fraction of $a$, $f(a)$ may be incorrect.

*Algorithm*

1. Pick $r \in_R F^m$.
2. Take the line $l(t) = (1-t)a + tr$ and we'll look at $p$ along this line.
3. Let $\tau_1, \tau_2, \ldots, \tau_{d+1}$ be distinct and non-zero elements from $F$. These do not need to be random.
4. Compute coefficients of $h : F \to F$ of degree $d$ such that $h(\tau_i) = f(l(\tau_i))$.
5. Output $h(0)$.

**Claim 16.2** *Self correction outputs $p(a)$ with probability $\geq 1 - (d+1)\delta$.*

**Proof** $l(\tau_i)$ is a random point in $F^m$ over random choice of $r$ for all non-zero $\tau_i$.

$$\Pr_r[f(l(\tau_i)) \neq p(l(\tau_i))] \leq \delta$$

By the union bound,
$$\Pr_r[\exists i \in \{1, \ldots, d+1\} f(l(\tau_i)) \neq p(l(\tau_i))] \leq (d+1)\delta$$
If the above event does not occur, then for all $i$, $h(\tau_i) = f(l(\tau_i)) = p(l(\tau_i))$.
So with probability $1-(d+1)\delta$, $h = p|_l$ and $h(0) = p(l(0)) = p(a)$. ■

The above is due to [Beaver, Feigenbaum] and [Lipton].
They were interested in how to compute a function $f(a)$ without revealing $a$.

**Low Degree Testing**

- Given oracle $f : F^m \to F$
- Completeness: if $f = p$ of degree $d$, then must accept with probability 1.
- Soundness: if $\forall p$ of degree $d$,
$$\Pr_x[f(x) \neq p(x)] > \delta \implies \text{must reject with high probability}$$

*Algorithm*

1. Repeat many times
   (a) Pick $a \in_R F^m$ at random.
   (b) Use the self correction algorithm to find $p(a)$ and verify $p(a) = f(a)$.

**Theorem 16.3 (Rubinfeld-Sudan, ALMSS)** *Soundness of above algorithm. $\exists \delta_0$ such that if $f$ is $\delta$-far from any polynomial $p$ then $f$ is rejected with probability $min\{\frac{\delta}{2}, \delta_0\}$.*

[Rubinfeld, Sudan] showed $\delta_0 = O(\frac{1}{d})$ and [ALMSS] showed that $\delta_0 = 10^{-3}$.

# Chapter 17

| **6.841/18.405J: Advanced Complexity Theory** | April 22, 2001 |
|---|---|
| Lecture 17 | |
| *Lecturer: Madhu Sudan* | *Scribe: Claire Monteleoni* |

Today's topics:

- Proof of NP $\subseteq$ PCP[$poly \log, poly \log$]
- $\Leftarrow$ Hardness of Polynomial Constraint Satisfaction

## 17.1 Gap version of PCS for SAT

The general goal of Polynomial Constraint Satisfaction is to find a function $f : F^m \to F$ satisfying as many constraints of $C_1 \ldots C_t$ as possible. Where $C_j$ = algebraic function whose inputs are $f(x_1^{(j)}), \ldots, f(x_k^{(j)})$.

The Gap version for SAT is hard. $\Phi \to k, d, F, C_1, \ldots, C_t$, such that:

$$\begin{cases} \Phi \in SAT \to \text{all constraints are satisfied by degree } d \text{ polynomial } f. \\ \Phi \notin SAT \to 90\% \text{ of constraints are not satisfied by degree } d \text{ polynomial } f. \end{cases}$$

The idea here is to express satisfiability by finding a function $f$ with the constraint on it's degree ($d$) that meets the level of satisfaction specified above. The function will take roughly $n$-bits to specify (cf. a SAT assignment is specified with $n$-bits).

The goal is to express SAT in terms of algebra, so that we can have some algebraic method of proving satisfiability. The method will be to arithmetize satisfiability. I.e. given $\Psi$, a candidate formula:

- cf. $\#P$: we counted the number of satisfying assignments for a formula $\Psi$.
- Now instead, we first fix an assignment, and then count the number of failed clauses of $\Psi$. If the number of failed clauses is zero, then $\Psi$ is satisfiable.

### 17.1.1 Step 1: Arithmetize the assignment

The notion of assignment is: $(w_1, \ldots, w_n) \leftarrow (a_1, \ldots, a_n) : n = 2^m$, for the variables $(w)$ in $\Psi$.

- Define the assignment function $A$, as: $A : \{0,1\}^m \rightarrow \{0,1\}$. So $A(i) = a_i$ which is either a 0 or a 1, and thus sets the value of $w_i$.

- Create a polynomial extension of the assignment function. $\hat{A} : F^m \rightarrow F$, such that:

  1. $\hat{A}(x) = A(x)$ if $x \in \{0,1\}^m \subseteq F^m$.
  2. $\hat{A}(x_1, \ldots, x_m)$ is a polynomial of degree 1 in each $x_i$. Note: this does not preclude higher overall degree of the polynomial, due to cross-terms (eg. $\hat{A}(x_1, \ldots, x_5) = x_1x_3x_5 + x_4 - x_2x_3$).

  **Claim** Existence of such an $\hat{A}$.
  For every $A : H^m \rightarrow F \quad : H \subseteq F$,
  $\exists \hat{A} : F^m \rightarrow F$ such that:

  - $\hat{A}$ extends $A$. (i.e. $\hat{A}(x) = A(x)$ for $x \in H^m$)
  - degree of $\hat{A}$ in each of its variables is $|H|$ - 1.

  **Proof** [omitted, exercises, algebra textbook] Induction on m (or explicit multivariate interpolation).[1]

Note: writing down $\hat{A}$ is a little more work than writing down $A$. Prover must write down the table of values that specify the polynomial $\hat{A}$. That is part of the existential step for the prover.

### 17.1.2 Step 2: Express the formula as a function

To express the formula $\Psi$ as a function, we define:

- $\Phi : \{0,1\}^m \times \{0,1\}^m \times \{0,1\}^m \times \{0,1\} \times \{0,1\} \times \{0,1\} \rightarrow \{0,1\}$
  The input to $\Phi$ above is just another notation for a 3CNF clause. So to make this mapping explicit, note that:
  $\Phi(w_{i_1}, w_{i_2}, w_{i_3}, b_1, b_2, b_3) = (A(w_{i_1}) = b_1) \vee (A(w_{i_2}) = b_2) \vee (A(w_{i_3}) = b_3)$
- Create a polynomial extension of $\Phi$. $\hat{\Phi} : F^{3m+3} \rightarrow F$, such that $\hat{\Phi}$ extends $\Phi$, and is degree 1 in each of it's variables.

Note: $\hat{\Phi}$ can be computed by the verifier.

### 17.1.3 Step 3: Arithmetize satisfiability

Create an arithmetic expression which encodes the satisfiability of $\Psi$. SAT? : $F^{3m+3} \rightarrow F$, such that:
SAT?(Does $\hat{A} : F^m \rightarrow F$, satisfy $\hat{\Phi}$)

The function SAT? behaves as follows:
SAT?$(w_{i_1}, w_{i_2}, w_{i_3}, b_1, b_2, b_3) = 0 \implies$

$$\begin{cases} A \text{ satisfies clause } (w_{i_1}, w_{i_2}, w_{i_3}, b_1, b_2, b_3) \text{ OR:} \\ \text{that clause } \notin \Psi \end{cases}$$

The idea here is that we will ask the prover to give us more and more information about the formula we are trying to verify. So now the goal is to create an expression that will be 0 iff the formula is satisfied.

[1] For the case of m=1, we can use the Polynomial Interpolation Theorem. It is more complicated for higher m.

What does this expression look like?

$$\text{SAT?} = (\hat{A}(w_{i_1}) - b_1)(\hat{A}(w_{i_2}) - b_2)(\hat{A}(w_{i_3}) - b_3)\hat{\Phi}(w_{i_1}, w_{i_2}, w_{i_3}, b_1, b_2, b_3)$$

Note that by definition of $\hat{A}$ and $\hat{\Phi}$, SAT? will only be at most degree 2 in each variable, so it is a polynomial of degree $2m'$ at most, where $m' = 3m + 3$. Thus its degree is logarithmic in $n$, where $n = |\Psi|$.

So now we have a well-defined scheme: search for $\hat{A}$ that causes SAT? to be zero somewhere. The question is, on what small subdomain must it be zero? Given $\hat{A}$ and SAT?, how do we know if they are implemented correctly? SAT? is zero on all of $\{0,1\}^{m'}$. There is a good way to check if two polynomials are consistent with one another, but how do we check if a given polynomial is zero on some small subdomain?

**Idea** cf. the proof of $\#P \subseteq IP$
See Figure 1.

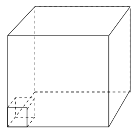


**Figure 17.1**: Similar to the #P $\subseteq$ IP proof, in which we were concerned with the smaller cube, $\{0,1\}^n$, where the larger cube was $Z_p^n$. Now instead we are interested in checking that all values in the smaller cube are zero, which represents $\{0,1\}^{\log n}$, and the larger cube is the field $F^{m'}$.

So the idea is that we will slowly make the SAT? polynomial nicer, working on one dimension at a time (i.e. one coordinate at a time), we will make sure that it will be zero for all values on that dimension. So we define SAT? as:

$$SAT? = \begin{cases} P_0 : F^{m'} \to F \text{ such that } P_0 \text{ is 0 on } \{0,1\}^{m'} \\ \vdots \\ P_i : F^{m'} \to F \text{ such that } P_i \text{ is 0 on } F^i \times \{0,1\}^{m'-i} \\ \vdots \\ P_{m'} : F^{m'} \to F \text{ such that } P_{m'} \text{ is 0 on } F^{m'} \end{cases}$$

Note that these will all be degree $2m'$ polynomials. These will be created by induction on $i$. In order to compute $P_{i+1}$, you will need to make oracle calls to $P_i$. Here is the inductive step: given $P_i(y_1, \ldots, y_i, y_{i+1}, \ldots, y_{m'})$, we need to construct $P_{i+1}$. We will show how this is done, in the case $m' = 1$.

In this case, given $P_0 : F \to F$, we want to implement $P_1 : F \to F$ so that it is:

$$= \begin{cases} 0 \text{ if } (P_0(0) = 0) \wedge (P_0(1) = 0) \\ \text{non-zero otherwise} \end{cases}$$

To implement this, we use the following algebraic gadget: $P_1(y) = yP_0(0) + P_0(1)$

Now note that in the general case, when we move from $P_i$ to $P_{i+1}$, everything else is fixed, so we can just focus on one variable. Thus we just construct $P_{i+1}$ as follows:

$$P_{i+1}(y_1, \dots, y_i, y_{i+1}, y_{i+2}, \dots, y_{m'}) = y_{i+1}P_i(y_1, \dots, y_i, 0, y_{i+2}, \dots, y_{m'}) + P_i(y_1, \dots, y_i, 1, y_{i+2}, \dots, y_{m'}) \tag{17.1}$$

Observe that the degree is increasing as we go from $i$ to $i+1$.

### 17.1.4 Computing with this construction

First the prover must write down $\hat{A}$. Then in order to specify the function SAT?, the prover must write down $P_0$, and then each of the higher polynomials $P_i$.

So how does the verify act to verify these polynomials? The verifier will:

- Check that $\hat{A}$ and SAT? are low degree polynomials, for all $P_0, \dots, P_{m'}$. This amounts to checking that the degree is at least 2 in each variable, i.e. that the total degree of the polynomial is at most $2m'$.

- Check that all the specified relationships hold. This is done as follows. Since the polynomials were each given as a table of values, table look-up can be done to verify that two sides of an equation (1) evaluate to the same value. So pick an $i$ at random and test this. This checking can be done in $O(m')$ =log$n$ time, i.e. one line at a time.[2]

  How much randomness does the verifier need? The verifier performs $O(m')$ tests, and each test needs $m'\log|F|$ randomness, so the verifier needs $O((m')^2|F|)$ randomness. But note however that in each case the probability of an error is extremely small. Using the Union Bound, we could instead use the *same* random point to verify all of the equations. The error probability is $\frac{(m')^2}{|F|}$.

### 17.1.5 Application back to PCS

Now we must do the final "shrinkwrapping" to show what the above implies about the polynomial constraint satisfaction problem.

First we need to combine all the necessary elements into one single polynomial. Thus in addition to our $P_0, \dots, P_{m'}$, as defined above, we define $P_{-1} = \hat{A} : F^m \to F \underbrace{\rightsquigarrow}_{padding} F^{m'} \to F$, where we pad this function with dummy variables, on which it does not really depend, in order to increase the dimension of its domain to $m'$. Now we define the function $f : F^{m'+1} \to F$ as follows:
$f(i, \hat{y}) = P_i(\hat{y})$ if $i \in \{-1, \dots, m'\}$
Note that $f$ is a polynomial of degree $m'+1$ in $i$ and at most 2 in each of the variables in the $\hat{y}$ vector.

So from the polynomial constraint satisfaction problem, $C_j$ corresponds to a random choice of a "verifier" that verifies the consistency between $P_i$ and $P_{i+1}$ (equation (1)) for all $i$.

Question: how large does $F$ have to be? It must be: $|F| \geq (m')^2$. I.e. for $|F| = (\log n)^2$, the size of the proof is $|F|^m = (\log n)^{2\log n} \approx n^{\log\log n}$. So the verifier must write down this much.

[2]Schwartz-Zippel

### 17.1.6 Further enhancements

It is possible to achieve shorter encodings than listed above, by changing base. Above we used the field $F = \{0,1\}$, and $|F^m| = n \implies m = \log n$. Now instead pick the field $H = \{0, 1, \dots, \log(n-1)\}$. Then $|H^m| = n \implies m = \frac{\log n}{\log\log n}$.[3] For example for $|F| \geq |H|^4 = (\log n)^4 \rightsquigarrow F^{m'} \approx n^{12}$, which is better than $n^{\log\log n}$. This will be summarized next lecture.

[3]Note that the proof above changes so that instead of the polynomial being degree 2 in each variable, the degree is now $|H|$ in each variable.

# Chapter 18

| **6.841/18.405J: Advanced Complexity Theory** | | April 24, 2002 |
|---|---|---|
| | Lecture 18 | |
| *Lecturer: Madhu Sudan* | | *Scribe: Nitin Thaper* |

**Today**

- A high level view of NP $\subseteq$ PCP[poly log, O(1)]

  Our main goal will be to look at ways of reducing the number of queries down to a constant.

## 18.1 A p-prover 1-round proof system

A p-prover, 1-round proof system consists of a verifier, tossing $R$ private coins and asking 1 question each from $p$ provers. At the end of the protocol, the verifier returns Verdict$(R, a_1, a_2, \ldots, a_p)$ = accept/reject.

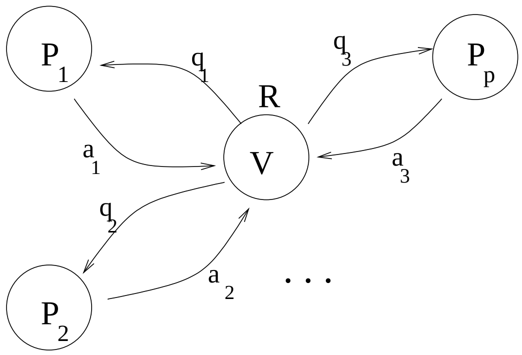



**Figure 18.1**: A p-prover, 1-round proof system

We have seen such a model in the context of MIP earlier but today we shall say something stronger about it.

**Theorem 18.1** $\exists$ *3-prover proof system for NP where*

- *verifier V tosses* $\log n$ *coins*

- $|a_i| = O(poly \log n)$
- *Completeness = 1, Soundness = 0.1 (arbitrarily small)*

We shall assume this result for the remainder of the lecture (Proving it is part of your problem set)

## 18.2 Reducing # queries down further

Observe that asking the question: Is Verdict$(R, a_1, a_2, a_3)$ = accept ?
is equivalent to creating a poly log n sized circuit, $C_R(a_1, a_2, a_3)$ and asking: Does $C_R(a_1, a_2, a_3)$ accept ?

Now in order to reduce the number of queries, intuitively, we would like the provers to commit to their answers, not necessarily reveal them. Then we can run a NP protocol with another set of 3 provers to check if $C_R(a_1, a_2, a_3)$ is true.

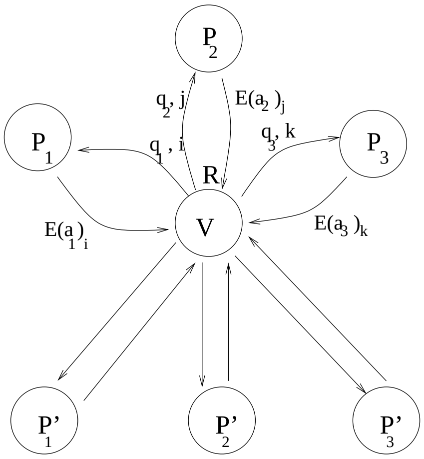



**Figure 18.2**: A 6-prover, 1 round protocol for NP. Provers $P_1, P_2, P_3$ just return a single, encoded bit.

Without giving a formal proof, we claim that this 6-prover protocol is sufficient for any NP problem. Observe that this protocol lets us trade off a constant number of provers with an exponential decrease in the number of query bits. More formally, we've just argued that:

$$\begin{aligned} NP &\subseteq MIP[3(provers), \log n(randombits), \log n(queries)] \\ &\subseteq MIP[6(provers), \log n(randombits), \log\log n(queries)] \end{aligned}$$

This process can be repeated again and again to reduce the number of queries to $\log\log\ldots\log n$. This is still not a constant though! In order to get $O(1)$ queries we have to be clever by trading off random coin tosses against queries.

## 18.3 Is NP $\subseteq$ PCP[poly n, O(1)] ?

We shall look at a somewhat orthogonal issue now, namely, is it possible to get a constant number of queries even if we were allowed polynomially many coin tosses. If the answer were yes, we might expect to put together things such that we got our desired result of proving NP $\subseteq$ PCP[log,O(1)].

As usual we'll look at the SAT problem. Consider $\psi \in SAT$ and let $\pi$ be the proof string.

### 18.3.1 Structure of $\pi$

Observe that each clause $C_i \in \psi$ can be viewed as a degree 3 polynomial, $p_i$ over $F_2$, such that

$$p_i = 0 \Leftrightarrow C_i \text{ is satisfied}$$

For example, $C_1 = x_1 \vee \bar{x}_2 \vee x_3$ corresponds to the polynomial, $p_1(x_1, x_2, x_3) = (1 - x_1)x_2(1 - x_3)$. We can derive such a correspondence for all the $m$ clauses.

Given some satisfying assignment $a$, the honest prover writes down the value of $p(a)$ for every polynomial of degree 3. Since there are $O(n^3)$ possible coefficents, there are $2^{O(n^3)}$ possible polynomials.
Thus, $|\pi| = 2^{O(n^3)}$.

An arbitrary prover, on the other hand, just writes down some function of the polynomials $\{F[p]\}_p$.

The verifier needs to make sure that:

1. $p_j(a) = 0, \forall j = 1..m$
2. $\exists a$ s.t. $F(p) = p(a) \, \forall p$

### 18.3.2 Doing Step 1 using constant queries

In order to verify that all clauses are satisfied we need to sonehow combine their corresponding polynomials into a single one, so that a single query reveals the answer with high confidence. A simple sum of all the polynomials wouldn't work since an even number of unsatisfied clauses would still fool the verifier. Instead we do the following:

Pick $r_1, r_2, \ldots, r_m \in_R 0, 1$ and consider the polynomial

$$p_r(x) = \sum_j r_j p_j(x)$$

Observe that this corresponds to taking the sum of a random subset of the polynomials $p_1, p_2, \ldots, p_m$.

Check if:

$$F[p_r] \quad = \quad 0 \tag{18.1}$$

Repeat this test a few times to get good confidence.

What if the prover is malicious ? In other words, $F(p)$ usually equals $p(a)$ but not on a small subset. Handling this problem is somewhat analogous to self-correction. We'll use the $F(p)$ oracle to get an oracle for $p(a)$.

<u>Given:</u> Oracle $F$ s.t. $\exists a$ s.t. for most $p$ (all but a $\delta$ fraction) $F(p) = p(a)$.
Q, a polynomial of degree 3
<u>Goal:</u> Q(a)

Observe that $Q(a) = (P + Q)(a) - P(a)$.

So pick $P$ at random (using $O(n^3)$ random bits) and return

$$Q(a) = F[P + Q] - F[P]$$

It's not hard to see that $Q(a)$ is right with probability at least $1 - 2\delta$.

Thus in equation (1), the verifier should actually be checking $F[P + p_r] - F[P]$ instead of $F[p_{\mathbf{r}}]$.

### 18.3.3 Doing Step 2 using constant queries

Given: Oracle $F$.
Goal: Does $\exists a$ s.t. $Pr_p[F[p] = p(a)] \geq 1 - \delta$

Idea: Use linearity.
Pick P,Q at random. If F[P+Q] = F[P] + F[Q] then

$$\exists a_i \, \exists b_{ij} \, \exists c_{ijk} \; s.t. \, F[P] = \sum p_{ijk} c_{ijk}$$

Note that

$$\#\text{possible F's} = 2^{2^{n^3}}$$

while,

$$\#\text{possible}\, c_{ijk} = 2^{n^3}$$

Phase 2: Use multiplication.
The polynomials can be partitioned into three classes depending on their degree.

| $a_i$ | $b_{ij}$ | $c_{ijk}$ |
|---|---|---|
| $F_1$ | $F_2$ | $F_3$ |

**Figure 18.3**: Proof $\pi$ partitioned into polynomials of different degrees

Pick at random $L_1, L_2$ of degree 1, $Q$ of degree 2. Test if $F_1[L_1]F_1[L_2] = F_2[Q + L_1 L_2] - F_2[Q]$.
Similarly test for degree 3 polynomials by picking $L_1 \in F_1$, $L_2 \in F_2$.

Overall, by reading $\sim 19 = O(1)$ bits from the proof we are still able to get a non-zero probability of detecting cheating. We will not formally prove the corectness of this procedure but a result due to Blum, Ruby and Rubinfeld shows that this is indeed the case.

# Chapter 19

| **6.841/18.405J: Advanced Complexity Theory** | April 29, 2002 |
|---|---|
| Lecture 19 | |
| *Lecturer: Madhu Sudan* | *Scribe: Christos Kapoutsis* |

## 19.1 Introduction

In the complexity-theoretic terms we have seen so far, the formal way to express the intuitive fact that "a problem is very difficult to practically solve by a computer" is to say that "the problem is NP-hard" or, if we restrict our attention to just problems in NP, to say that "the problem is NP-complete". If we indeed accept that this matching between intuition and formalism is a correct one and we further assume that P$\neq$NP, then it seems like we should also accept the following two conclusions:

- If a formal problem correctly describes a practical one and also proves to be NP-complete, then we should stop looking for efficient solutions to the practical problem.
- Cryptography is feasible.

The truth is that neither conclusion can be derived without additional assumptions. Today we discuss why this is so for the first of them.

The motivation comes from real-world examples of algorithms that solve NP-complete problems in a "satisfactory" way, in all practical respects. By studying what "satisfactory" means in those cases, we see that two important choices that we made when we defined *what it means for an algorithm to solve a problem* are actually not necessary, as far as practical solvability is concerned:

- **Exact solution:** We required that for every instance of the problem the algorithm should return the *exact* answer. In certain practical cases, we are just as happy even with some *approximation* of this exact answer.
- **Worst case instances:** We required that the algorithm should return the exact answer for *every* instance of the problem. In certain practical cases, hard instances (i.e., the instances responsible for the problem being NP-complete) rarely happen, and we are just as happy even with an exact answer *most of the times*.

In the next two sections we see what modifications in our theory so far are appropriate for the study of the above cases.

## 19.2 Approximation algorithms

Formally, a *combinatorial optimization problem* is any polynomial-time computable function $f : \{0,1\}^* \times \{0,1\}^* \to \mathbb{Z}^+$ mapping every instance $x \in \{0,1\}^*$ of the problem and every possible answer $y \in \{0,1\}^*$ to it to some value $f(x,y) \in \mathbb{Z}^+$ indicating how "good" $y$ is as a solution to $x$. An algorithm $A$ *solves* $f$ if, for every $x$, $A(x)$ is $f(x,y)$ for the best solution $y$ to $x$,

$$A(x) = \max_y \{f(x,y)\}.$$

Sometimes $f(x,y)$ indicates how "bad" $y$ is as a solution to $x$, in which case it is the minimun we are after, instead of the maximum.

For example, the problem of finding the size of the maximum clique in a graph is defined by the function MAXCLIQUE mapping every graph $G = (V,E)$ and every $c \subseteq V$ to the value

$$\text{MAXCLIQUE}(G,c) = \begin{cases} 0 & \text{if } c \text{ is not a clique in } G, \\ \text{the size of } c & \text{otherwise,} \end{cases}$$

and we want to maximize $\text{MAXCLIQUE}(G,\cdot)$. Similarly, the optimization problem of finding the chromatic number of a graph is defined by the function CHROMATIC mapping every graph $G = (V,E)$ and every coloring $c : V \to \mathbb{N}$ of its vertices to the value

$$\text{CHROMATIC}(G,c) = \begin{cases} \infty & \text{if } c \text{ is an illegal coloring,} \\ \text{the number of colors used by } c & \text{otherwise,} \end{cases}$$

and we want to minimize $\text{CHROMATIC}(G,\cdot)$. The problems MAXSAT and MAX3SAT are also defined in a similar manner.

Relaxing the requirement that the exact optimum is returned, we get the notion of an *$\alpha$-approximating algorithm for a problem $f$*, where $\alpha$ describes the proximity to the exact optimum that we want to achieve as a function from input sizes to reals not less than 1. We say $A$ is such an algorithm if, for every instance $x$ of the problem,

$$\frac{\max_y\{f(x,y)\}}{\alpha(|x|)} \leq A(x) \leq \max_y\{f(x,y)\}.$$

In general, we say that any pair $(f,\alpha)$ defines an *approximability problem* and that $A$ *solves* $(f,\alpha)$ if $A$ $\alpha$-approximates $f$.

Clearly, the step from optimization problems to approximability problems can be seen as the step from considering only the fixed approximation ratio $\alpha(n) = 1$ to allowing other approximation ratios as well. Work by Johnson in the late 70's showed that this view offers little help: Problems that behave similarly for $\alpha(n) = 1$ may exhibit radically different behaviour for other approximation ratios. For example, for $f$ any of the problems MAXSAT, MAXCLIQUE, CHROMATIC, we know that the approximability problem $(f,1)$ is NP-complete. However:

- $(\text{MAXSAT}, 2)$ is in P.
- For all $\epsilon > 0$ and $c > 1$, $(\text{MAXCLIQUE}, 1+\epsilon)$ is as hard as $(\text{MAXCLIQUE}, c)$. That is, if one can approximate the maximum clique size of a graph within a constant factor, one can also approximate it with a ratio as close to 1 as desired.
- $(\text{CHROMATIC}, \frac{6}{5})$ is NP-complete.

This clearly shows that reasoning like $(f,\alpha) \equiv (g,\alpha) \implies (f,\alpha') \equiv (g,\alpha')$ is wrong.

The PCP methodology has provided techniques for studying approximation problems, and some of the results known today are:

- (MAXSAT, $\frac{8}{7}$) is NP-complete.
- (MAX3SAT, $\frac{8}{7} - \epsilon$) is NP-complete, for all $\epsilon > 0$.
- (MAX3SAT, $\frac{8}{7} + \epsilon$) is in P, for all $\epsilon > 0$.
- (MAXCLIQUE, $n^{1-\epsilon}$) is NP-complete, for all $\epsilon > 0$.
- (CHROMATIC, $n^{1-\epsilon}$) is NP-complete, for all $\epsilon > 0$.

## 19.3 Average case complexity

Formally, a *distributional decision problem* is a pair $(L, D)$, where $L \subseteq \{0,1\}^*$ is a language and $D$ is a distribution over $\{0,1\}^*$. (That is, $D$ is a function $D : \{0,1\}^* \to [0,1]$ such that $\sum_{x\in\{0,1\}^*} D(x) = 1$.)

To solve the problem, an algorithm has to correctly determine membership in $L$ when the inputs are drawn according to $D$. The algorithm is considered efficient if it "runs in expected polynomial time". The quotes indicate that the notion of efficiency is tricky to formalize, especially when one considers problems in NP. Work on how to do this correctly was done by Levin (1984).

### 19.3.1 Random self-reducibility of the permanent

For problems of high complexity (PSPACE, #P), the notions seem to be clearer. We begin the study of average case complexity with a theorem that computing the permanent is as hard on average as in worst case. Recall that a permanent of a matrix $M = [m_{i,j}]$ is given by

$$\mathrm{perm}(M) = \sum_{\pi\in S_n} \prod_{i=1}^{n} m_{i,\pi(i)},$$

where $S_n$ is the set of all permutations on $[n] = \{0, 1, \ldots, n-1\}$.

We start with the definition of a distribution $D$ over matrices. For every $n \in \mathbb{Z}^+$, consider the following random experiment:

1. randomly select a prime $p$ of size[1] $\Theta(n^{10})$, then
2. randomly select a matrix from $\mathbb{Z}_p^{n\times n}$ (i.e., construct an $n \times n$ matrix by randomly selecting each one of its elements from $\mathbb{Z}_p$).

Let $D_n$ be the distribution over matrices defined by this experiment; then let $D = (D_n)_{n\geq 1}$. [Note that this doesn't exactly match the definition of a distribution over $\{0,1\}^*$ given above. This is ok, as it won't harm the correctness of the theorem. Alternatively, instead of one separate experiment for every $n \in \mathbb{Z}^+$, we could have only one experiment:

0′. randomly select $n$ from $\mathbb{Z}^+$ with probability $\frac{1}{n(n+1)}$, then

1′. do Step 1 from above for the selected $n$, then

2′. do Step 2 from above for the selected $n$,

and have $D$ be the distribution over matrices defined by this experiment (this is the correct definition according to Levin).]

The theorem we are going to prove is the following:

[1] "Size" here is the actual size of the number; not its length in bits.

**Theorem 19.1** If *there exists a polynomial-time algorithm $A$ such that, for all sufficiently large $n$ and* for matrices drawn from $D_n$, *$A$ computes the permanent correctly with probability* $1 - \frac{1}{\mathrm{poly}(n)}$, then *there is also a randomized, polynomial-time algorithm $B$ that, for all sufficiently large $n$ and* for every matrix $M$ in $D_n$, *$B$ computes the permanent of $M$ correctly with high probability.*

**Proof** We don't get into the issues concerning the random selection of the prime number $p$. We consider a fixed $p$ of size $\Theta(n^{10})$ and assume that a polynomial-time algorithm $A(\cdot, p)$ computes the permanent correctly with probability $1 - \frac{1}{\mathrm{poly}(n)}$ for matrices randomly selected from $Z_p^{n\times n}$. Consider the algorithm $B(\cdot, p)$ that on input $M \in \mathbb{Z}_p^{n\times n}$ computes as follows:

1. picks a random matrix $R$ from $\mathbb{Z}_p^{n\times n}$,
2. for each $i = 1, 2, \ldots, n+1$, computes $y_i \leftarrow A(M + i \cdot R, p)$ (where $i \cdot R$ stands for multiplying every entry of $R$ by $i$),
3. interpolates and finds a polynomial $q(x)$ of degree $n$ such that $q(i) = y_i$, for all $i = 1, 2, \ldots, n+1$.
4. outputs $q(0)$.

It can be shown that with high probability over its internal coin tosses, $B(\cdot, p)$ correctly computes the permanent of its input. ■

Note that the proof uses the idea of "self-correction" that we have seen in previous lectures: symbolically, $\mathrm{perm}(M)$ is a degree $n$ polynomial in the $n^2$ variables $m_{1,1}, m_{1,2}, \ldots, m_{n,n}$; and $A$ is an oracle that computes this polynomial correctly quite often; so, $B$ is what we get by applying self-correction to $A$.

### 19.3.2 The class DNP

In this section we give the definition of the class DNP (Distributional NP). In the next lectures we will give theorems about it.

First, we need to associate with every distribution $D$ the function $\tilde{D} : \{0,1\}^* \to [0,1]$, defined by

$$\tilde{D}(x) = \sum_{y \le x} D(y),$$

so that $\tilde{D}(x)$ is the probability that a string selected according to $D$ "comes before" $x$. Here, a fixed, reasonable liner ordering of $\{0,1\}^*$ is assumed; e.g., the usual ordering $0, 1, 00, 01, 10, 11, \ldots$.

Now, we are interested in two special kinds of distributions over $\{0,1\}^*$.

**Definition 19.2** *A distribution $D$ over $\{0,1\}^*$ is P-computable if $\tilde{D}$ is polynomial-time computable (i.e., there exists a polynomial-time algorithm that on input $x$ outputs $\tilde{D}(x)$).*

An example of such a distribution is the *uniform* distribution, defined by $D(x) = \frac{1}{|x|(|x|+1)} 2^{-|x|}$.

**Definition 19.3** *A distribution $D$ over $\{0,1\}^*$ is P-samplable if there exists a probabilistic algorithm $A$ and a polynomial $p$ such that $A$ outputs $x$ with probability $D(x)$ and after $p(|x|)$ steps.*

A P-computable distribution is P-samplable, as well. But the converse is probably false.

**Definition 19.4** *The class DNP contains exactly the distributional decision problems $(L, D)$ for which the language $L$ is in NP and the distribution $D$ is P-computable.*

Note that considering NP languages with P-samplable distributions results in a class larger than DNP.

# Chapter 20

**6.841/18.405J: Advanced Complexity Theory** May 1, 2002

Lecture 20

*Lecturer: Madhu Sudan* *Scribe: Eleni Drinea*

Today we will first elaborate on the definition of Distributed NP, introduced in the last lecture. We will then define what it means for a problem to be in Avg-P and start discussing a completeness result for DNP.

## 20.1 Definitional Issues

In the definition of a distributional problem in the last lecture, the input distribution was a single distribution on all inputs of all sizes. Equivalently, according to Impagliazzo, we can think of the input distribution as being on a finite set of possible inputs (e.g., of at most some fixed size $n$). Thus for today's lecture, the distribution $D$ we will work with is $D = \{D_n\}_{m=1}^{\infty}$, i.e., a collection of distributions $D_n$.

There are two classes of distributions that we are interested in:

- $P$-computable distributions $D$: let $\tilde{D} = \sum_{y \leq x} D(y)$, where we think of the ordering "$y \leq x$" on $\{0,1\}^n$. We say that $D$ is $P$-computable if $\tilde{D}$ is polynomial time computable. This is the nicest class of distributions one can think of. Recall that we defined a pair $(L, D)$ to be a decision problem in DNP if $L \in NP$ and $D$ is P-computable.
- P-samplable distributions $D$: this is a quite elaborate class of distributions. We say that $D$ is $P$-samplable if there exists a polynomial time algorithm $A$ that outputs $x$ with probability $D(x)$. Note that a $P$-samplable distribution need not be $P$-computable. For example, if we pick an assigment $x \in \{0,1\}^m$ on $m$ variables, and construct a formula $F$ on (say) $4m$ clauses that is satisfied by $x$, then the distribution on such formulae is $P$-samplable. However, in order to compute $\tilde{D}(F)$, we probably need #P power in order to compute the probability of each $F' < F$ (since we need to count the number of assignments that satisfy $F'$). Thus this distribution is probably not $P$-computable.

## 20.2 $\delta$-good algorithms and Avg-P

We define problems in *Distributed NP* to be pairs $(R, D)$ such that $R$ is a polynomial time computable binary relation $R(x,y)$ and $D$ is a distribution on $x$ (the first of $R$'s arguments). Given $x$ distributed according to $D$, we want to find a $y$ such that $R(x,y)$, if such a $y$ exists. A notion that is related to how well we want to solve this search problem is Avg-P.

A problem is in Avg-P if there exists an "efficient" algorithm $B$ that "solves" this problem. Of course, we need to be more specific about the notions in quotes above. For the notion of "solvability", $\delta$-good algorithms are satisfactory.

**Definition 20.1** *An algorithm $A$ is $\delta$-good for $R$ and $D$ if*

$$\Pr_{x\leftarrow D}\left[\begin{array}{l} \textit{if } \exists y \textit{ s.t. } R(x,y), \textit{ then } R(x,A(x)) \textit{ is true} \\ \textit{and} \\ \textit{if } \forall y\ \neg R(x,y), \textit{ then } A(x)=\textit{"error"} \end{array}\right] \geq 1-\delta$$

Note that we are using only benign algorithms, that never make errors. However, they may need more time to produce a definite answer; in this case they output "?". In other words, $A$ given $x$ can produce three outputs: $y$, in which case $R(x,y)$ is true, "error", in which case there is no $y$ such that $R(x,y)$, or "?", if $A$ does not have enough time to decide.

Now let's look at the efficiency requirements for algorithms for problems in Avg-P. For example, consider an algorithm $A$ that solves a problem $(R,D)$ in the following way: with probability $1-\frac{1}{2\sqrt{n}}$ it takes time $n$ and with probability $\frac{1}{2\sqrt{n}}$ it takes time $t=2^n$. The expected running time of the algorithm is $2^{\Theta(n)}$; then $A$ is a *bad* algorithm for the problem if our criterion is the expected running time. However if we change $t$ to $2^{n^{1/3}}$, then (under the same criterion) $A$ becomes *good*. Similarly, if $A$ runs on a 2-tape TM and $t=2^{.75\sqrt{m}}$ time, then $A$ is *good*. However if we simulate this algorithm on an 1-tape TM, with probability $\frac{1}{2\sqrt{n}}$ we will need time $2^{1.5\sqrt{m}}$ (because of the quadratic overhead of the simulation). Thus the expected running time of the simulation is exponential in $n$ and the algorithm will now be considered *bad*.

The observations so far already suggest that the expected running time is not suitable as a criterion for the efficiency of an algorithm that solves a problem in Avg-P. First it is hardwired in the model of computation; and even polynomial changes in the running time do not maintain the property of *goodness*. Moreover, using such algorithms may cause problems in the composition of reductions. In particular, suppose $(R,D)\in$ Avg-P. Then there may exist $(R',D')$ that reduces in polynomial time to $(R,D)$ and yet, $(R',D')\notin$ Avg-P. This is certainly something that the notion of reduction should not allow.

The above discussion leads us to the following definition for Avg-P:

**Definition 20.2** *A problem $(R,D)$ is in Avg-P if there exists an algorithm $B$ on two inputs, $x$ and $\delta$, such that $B(\cdot,\delta)$ is $\delta$-good for $(R,D)$ and $B$ runs in time polynomial in the length of $x$ and in $1/\delta$.*

This definition of Avg-P is robust (e.g., the problem with the reductions no longer exists) and also makes sense, as the running time increases when we increase the probability that the algorithm returns a definite answer (i.e., we decrease $\delta$).

## 20.3 Towards a completeness result

The main question that arises at this point is whether DNP $\subseteq$ Avg-P, with the distributions considered either P-computable or P-samplable. The reasonable way to go about this issue is to define a complete problem in DNP and ask whether this is in Avg-P.

### 20.3.1 $\alpha$-dominance between distributions

Before we actually address this question, it is reasonable to ask whether DNP problems with P-samplable distributions are harder than DNP problems with P-computable distributions. Impagliazzo and Levin proved that this is not the case; every DNP problem complete for P-computable distributions is also complete for all samplable distributions. In particular, starting with problem $(R,D)$, where $R$ is an arbitrary (polynomial time) relation and $D$ is P-samplable, we can reduce this to some problem $(R'_{(R,D)},D')$, where $D'$ is P-computable, e.g. approximately uniform. (We will soon discuss what is means to reduce a distributional problem to another distributional problem.)

Before the Levin-Impagliazzo theorem, Levin proved that every problem $(R, D)$, where $D$ is P-computable, can be reduced to $(\Pi, U)$, where $\Pi$ is a fixed problem and $U$ is a uniform distribution. Thus there is a complete problem for DNP, with P-computable distributions.

Our goal for now is to give the high-level description of how the reduction from DNP with P-samplable distributions to DNP with P-computable distributions works. However, before we get to the actual reduction, we will discuss what it means to "reduce" a distributional problem to another.

From now on, we will be thinking of $D$ as being a sampling algorithm. I.e., $D$ gets $x$, which is a uniformly distributed $n$-lettered string and outputs an $n$-lettered string $y$ distributed according to $D$.

Now suppose we are given an instance of $(R, D)$ and want to reduce it to an instance of $(R', D')$. This may be too strong to require in general and we do not really need to achieve that much: maybe we are satisfied if the instances of $(R', D')$ are not produced exactly according to $D'$ but rather according to some $D''$ which is nicely related to $D'$. In other words, we would like to say that if $A$ is an algorithm that is $\delta$-good for $(R', D')$ then $A$ is also $\delta'$-good for some $(R', D'')$, given that $D''$ is related in a certain way to $D'$. It turns out the right way to formulate this relation is the notion of $\alpha$-dominance.

**Definition 20.3** *We say that a distribution $D_1$ $\alpha$-dominates a distribution $D_2$ if for all $x$*

$$D_1(x) \geq \frac{D_2(x)}{\alpha}$$

The intuition here is that if $A$ is good on some $D_1$, then it will still be good on some $D_2$ that is $\alpha$-dominated by $D_1$. This is formally stated in the following theorem.

**Theorem 20.4** *If $A$ is $\delta$-good for $(R, D_1)$ and $D_1$ $\alpha$-dominates $D_2$, then $A$ is $\alpha\delta$-good for $(R, D_2)$.*

Recall that in Avg-P we are in control of the $\delta$; for example, if $D_1$ has a bad $\alpha$ dominance over $D_2$, then we can adjust $\delta$ so that $\alpha\delta$ is such that the algorithm is still good enough for $(R, D_2)$.

### 20.3.2 The Impagliazzo-Levin Reduction

Let us consider the distribution $D_m$ that picks an integer $k$ uniformly at random from the set $\{1, \ldots, n\}$ and outputs $(k, w)$, where $w$ is chosen uniformly at random from $\{0,1\}^k$. This distribution is essentially "uniform". In general, consider $D_n$ that outputs a collection of tuples, such that the first element of the tuple specifies the rest. For example, $D_n : (k, x, y, z, i, w)$, where $x, y, z$ have length $k$, $i$ is an integer in $1, \ldots, k$ and $w$ has length $i$. Then these are certainly P-computable distributions; we will also consider these as uniform distributions.

Our goal is to reduce a problem $(R, D)$, supposedly hard, where $R$ is arbitrary and $D$ is P-samplable, to a problem $(R', D')$, where $D'$ is uniform (as previously discussed). In this setup, the universe picks $z \in \{0,1\}^n$, applies $D$ and outputs $D(z) = x \in \{0,1\}^n$. Now we need to find $R'$ such that $(R', D')$ is hard, with $D'$ being a uniform distribution.

A first attempt for $R'$ would be $R'(z, y) = R(D(z), y)$. Clearly $R'$ is polynomial time computable since $D$ is P-samplable and if $R$ is hard on $D$, so is $R'$ on the uniform distribution. However, this attempt fails, in that, looking at the formal reduction, given $x$, we should find some $z$ such that $D(z) = x$. But in general $D^{-1}(x)$ may not be tractable (e.g., as we mentined earlier today, $z$ could be an assignment over $m$ variables and $D(z) = x$ a formula on a certain number of clauses satisfied by $z$).

The idea that works is to implicitly (rather than explicitly) specify $z$. This means the following: suppose all the preimages of $x$ are in the set $S$, which consists of $2^k$ elements; suppose further that $z$ is the $i$-th element in $S$. Then $z$ can be specified by $(x, k, i)$, where $i \in \{0,1\}^k$ is the index of $z$ in $S$.

However, enumerating all the elements in $S$ is tricky. What we do instead is the following: we define the distribution $D_2$ that outputs tuples of the form

$$(x, h, k, w),$$

where a $z$ is picked from the $n$-lettered universe and then $x = D(z)$ is computed; $h$ is a randomly generated hash function on the $n$-lettered strings; $k$ is uniformly chosen from $\{0, \ldots, n\}$ and is a guess for the logarithm of the number of preimages of $x$; $h(z) = w \in \{0,1\}^k$ (we can think of applying $h$ on $z$ and then just look at the first $k$ coordinates). We claim that if solving $R$ on $D$ is hard, then solving $R$ on $D_2$ is also hard.

Now we define a uniform distribution $D_1$ that outputs tuples

$$(x, h, k, w),$$

where $x$ is uniformly chosen from the $n$ lettered strings, $h$ is randomly generated, $k \in \{1, \ldots, n\}$, and $w$ is uniformly chosen from $\{0,1\}^k$. We also define $R'$ as

$$R'\left((x, h, k, w), (y, z)\right) = R(x, y) \text{AND} \left(h(z) = w, D(z) = x\right)$$

where $h(z)$ is restricted to the first $k$ coordinates.

The interesting thing about defining $D_2$ is that now we can claim that $R'$ on $D_1$ is as hard to solve as $R$ on $D_2$. The basic idea behind the proof is that $D_1$ dominates $D_2$ within a polynomial factor.

# Chapter 21

| **6.841/18.405J: Advanced Complexity Theory** | | September 5, 2001 |
|---|---|---|
| | Lecture 21 | |
| *Lecturer: Madhu Sudan* | | *Scribe: Ashish Mishra* |

Today's lecture:

- A DNP-Complete Problem
- Lattice Problems and Worst-Case vs. Average-Case complexity: connection to NP

## 21.1 Recap from last lecture

We present a high-level view of the notion of average-case complexity. Some progress has been made in the last 10-15 years on analyzing these problems, including connecting problems of worst-case and average-case complexity, and developing the first cryptosystem relying on worst-case, rather than average-case complexity.

In the last lecture we talked about DNP (Distributed NP).

**Definition 21.1** *DNP is the class of problems specified by $(R, D)$ where $R$ is a poly time computable binary relation and $D : \{0,1\}^n \to \{0,1\}^n$ is a poly time computable function.*

Effectively, $D$ specifies the distribution induced by applying the function to a uniformly chosen random input, $\{D(z)\}_{z \in_U \{0,1\}^n}$. In practice we won't concern ourselves with the exact length of the random input since random lengths can be mapped to each other to a good extent.

**Definition 21.2** *An algorithm $A$ is $\delta$-good for $(R, D)$ if $A$ solves $(1-\delta)$ fraction of the instances of $R$ drawn according to $D$, never giving a wrong answer.*

**Definition 21.3** *$(R, D) \in$ Avg-P iff there is a $B(x, \delta)$ such that $A_\delta(.) = B(., \delta)$ is $\delta$-good for $(R, D)$ and $B(x, \delta)$ runs in poly$(|x|, 1/\delta)$ time.*

This definition is drawn from Impagliazzo's survey.

The idea of reductions is straightforward: we pick a random $z$, and give instance $D(z)$. The reduction specifies a particular distribution. To relate different distributions, we introduce the notion of "domination".

**Definition 21.4** *Distribution $D_1$ $\alpha$-dominates distribution $D_2$ if $\forall x$,*

$$\mathrm{Pr}_{D_1}[x] \geq \frac{\mathrm{Pr}_{D_2}[x]}{\alpha}$$

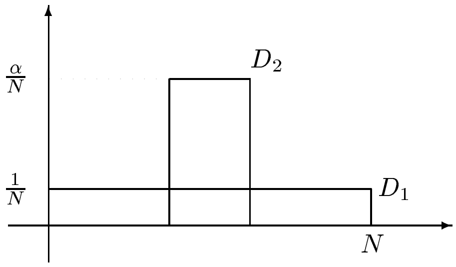



**Figure 21.1**: Dominating distributions

Domination is a very one-sided relationship and it is easy to construct distributions $D_1, D_2$ as in Figure 21.1 where $D_1$ $\alpha$-dominates $D_2$ — but $D_2$ doesn't $\alpha'$-dominate $D_1$ for any finite $\alpha'$.

**Theorem 21.5** *If algorithm $A$ is $\delta$-good for $(R, D_1)$ and $D_1$ $\alpha$-dominates $D_2$, then $A$ is $(\alpha\delta)$-good for $(R, D_2)$.*

The Impagliazzo-Levin Lemma states that every $(R, D)$ problem reduces to some $(R', U)$ where $U$ is an essentially uniform distribution.

Consider $R'$ which is the composition $R \circ D$. To implement this, we need to have a uniformly randomly chosen pre-image of $x$ under $D$. Computing the inverse of $D$ may be hard, so instead we specify $z$ implicitly: by giving an index $w$ of $z$ within the pre-images of $x$.

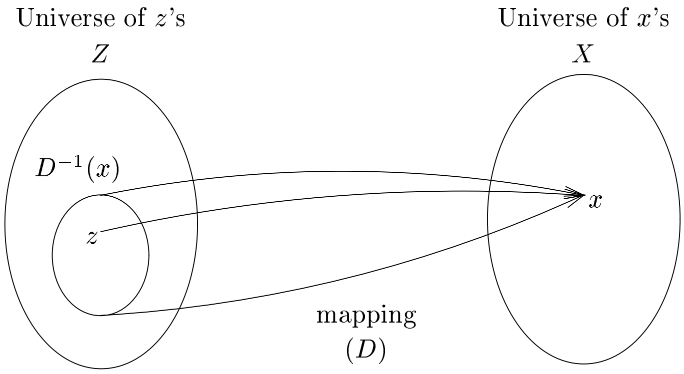



**Figure 21.2**: Picking a Distribution

We are picking $x \in \{0,1\}^n, w \in \{0,1\}^k$, where $x$ has roughly $2^k$ pre-images under $D$. $x$ and $w$ specify $z \in \{0,1\}^n$, and we would like the distribution of $z$ to be nearly uniform. We can do this by hashing the $(n+k)$ length string $(x, w)$ down to an $n$-length string. The hash is drawn uniformly from a family of pairwise independent hash functions.

Formally $R'$ is expressed as a tuple $(u, k, h_1, h_2)$ where $u \in \{0,1\}^n$ is the hash of $(x, w)$, $k \in \{0, 1...n\}$ is the log of the number of pre-images of $x$, and $h_1 : \{0,1\}^n \to \{0,1\}^k$, $h_2 : \{0,1\}^{n+k} \to \{0,1\}^n$ are hash functions. $R'((u, k, h_1, h_2), (z, y))$ iff:

- $y$ is a witness to $z$ under $R$, i.e. $R(D(z), y)$.
- $u = h_2(D(z), h_1(z))$.

Our distribution on $U$ is to pick $u, k$ uniformly from $\{0,1\}^n$ and $\{0, 1...n\}$ respectively, and $h_1, h_2$ uniformly from the families of hash functions on that many bits. The tricky part is guessing the size of the pre-image correctly. But with a random guess we have a $\frac{1}{n+1}$ chance of getting the right value, within a factor of 2.

## 21.2 Proof of the Impagliazzo-Levin Lemma

### 21.2.1 Reduction

Given $x$,

1. Uniformly pick $k \in_U \{0, 1, ...n\}$. (so we have $\frac{1}{n+1}$ chance of picking the right value)
2. Uniformly pick $w \in_U \{0,1\}^k$.
   We don't yet know $z$ or a distribution for $z$, but we can stipulate that $w = h_1(z)$.
3. Uniformly pick hash functions $h_1$, $h_2$.
4. Output: $(u = h_2(x, w), k, h_1, h_2)$.

If we have accurately estimated the number of pre-images $2^k$ above, then with high probability there will exist unique $z$ such that $D(z) = x$ and $h_1(z) = w$.

### 21.2.2 Analysis

We can construct a corresponding distribution $D_2$ on $(u, k, h_1, h_2)$ as follows:

1. Uniformly pick $z \in_U \{0,1\}^n$.
2. Set $k = \log_2 |D^{-1}(D(z))|$.
   We don't know how to compute this, we are just defining a distribution for the purpose of analysis.
3. Uniformly pick hash functions $h_1$, $h_2$.
4. Output: $(u = h_2(D(z), h_1(z)), k, h_1, h_2)$.

We make the following claims on $D_2$:

- The distribution $U$ $O(n)$-dominates $D_2$.
- $(R', D_2)$ is at least as hard as $(R, D)$.

This completes the proof that given $(R, D)$, there exists a problem $(R', U)$ which is equally hard.

However, the above does not provide us with a single hard problem independent of $R$ and $D$. To do this, we create a universal relation:

$$R_U \left\langle \begin{array}{c} (R, x) \\ y \end{array} \right.$$

with $R_U((R, x), y)$ holding iff $R(x, y)$ holds. We need not worry what distribution to apply to $R_U$ — as long as $x$ is chosen with finite probability, any distribution suffices.

This means that every interesting problem (e.g. factoring, SAT) has a uniform distribution describing it.

## 21.3 Lattice problems

DNP gives us a theory of average-case problems. Effectively, it relates average-case problems to each other. But it does *not* relate average-case to worst-case problems. What might be desirable is a connection analogous to that between approximation problems and exact calculation problems. Currently we don't know of any such result relating worst-case and average-case hardness in NP.

In 1996, Ajtai showed the existence of a lattice problem $R'$ that we don't know how to solve in RP; and an instance $(R, D)$ of Avg-P such that $R'$ is reducible to $(R, D)$. Essentially this gives a reduction from a "hard" problem in worst-case complexity to a related problem in average-case complexity. If $R'$ was known to be NP-complete, this would be a dream theorem in our current context. As it is, it represents a major breakthrough in classifying average-case complexity.

### 21.3.1 Definitions

Lattices consist of discrete points in $\mathbb{R}^n$, in some regular symmetric form. Figure 21.3 illustrates an example with only 2 dimensions for pictorial effect.

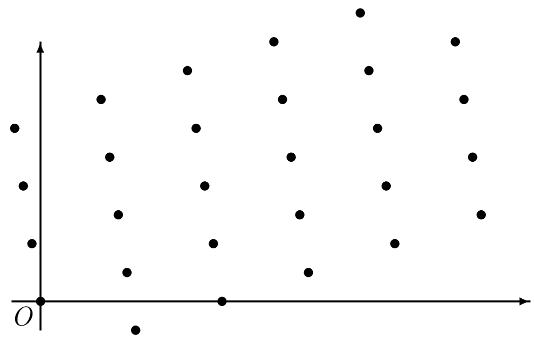



**Figure 21.3**: 2-dimensional lattice

It is important to distinguish this geometric notion of lattices from the algebraic entities involving partially ordered sets, as used in other contexts. Mathematically

**Definition 21.6** *A lattice $L$ is a discrete additive subset of $\mathbb{R}^n$.*

**Discrete:** There is some $d > 0$ such that for any $x \in L$, $\text{Ball}(x, d) \cap L = \{x\}$

**Additive:** For any $x, y \in L$, it follows that $x + y, x - y \in L$

If $L$ is nonempty then it clearly must contain the origin, since from additivity $x - x \in L$. There are two ways to specify a lattice, known as the *primal* and the *dual* representations.

#### Primal Representation

This gives a basis for the lattice: $b_1, b_2, ...b_m \in \mathbb{R}^n$, with the $b_i$'s linearly independent ($\Rightarrow m \leq n$). Then

$$L(b_1, ...b_m) = \left\{ \sum_{i=1}^{m} z_i b_i \mid z_i \in \mathbb{Z} \; \forall i \right\}$$

We often think about lattices on $\mathcal{Q}^n$, $\mathbb{Z}^n$ rather than $\mathbb{R}^n$.

**Dual Representation**

While the primal representation gives a constructive description of the lattice, this representation instead specifies a set of constraints on it. Mathematically we are given $b_1^*, b_2^* ... b_m^*$ with $m \geq n$. Then

$$L(b_1^*, ... b_m^*) = \left\{ v \mid \langle v, b_j^* \rangle \in \mathbb{Z} \,\forall j \right\}$$

where $\langle v, b \rangle$ denotes the inner product of $v$ and $b$.

If we have precisely $n$ vectors, the dual basis vectors are simply given by the inverse of the matrix composed of basis vectors. In other cases, it is not easy to see what the dual basis vectors intuitively represent, but it is still algorithmically easy to go from one representation to another.

### 21.3.2 Problems relating to lattices

**Easy Problems** :

1. Computing intersection of two lattices (itself a lattice).
2. Computing bases of a lattice.

**Hard Problems** :

1. **Short Vector Problem (SVP):** Given a basis, does there exist a non-zero vector of length $\leq d$ in the lattice?

   Ajtai showed this problem to be NP-complete under randomized reductions.
2. Given a basis $b_1, b_2 ... b_m$, can you find a short basis of length $\leq d$ for the same lattice?
3. Given $b_1, b_2 ... b_m$, and a target $t \in \mathbb{R}^n$, what is the nearest vector to $t$ in $L$? (NP-hard)

All these hard problems are optimization problems, so we can consider approximation algorithms to solve them. Lenstra, Lenstra and Lovasz gave a $2^n$-approximation algorithm to SVP on an $n$-dimensional lattice. This was subsequently improved, but not significantly. We now have $2^{o(n)}$-approximations, but not $2^{\sqrt{n}}$.

Though this approximation might seem too weak to be any use, it actually has significant applications. LLL showed how to use it to factor integer polynomials in poly time. Other uses of lattice problems are in cryptanalysis, integer programming, diophantine systems, and even in building cryptosystems [Ajtai-Dvork]. This shows that lattice problems are of immense interest.

### 21.3.3 Ajtai's theorem

Ajtai's theorem involves polynomial approximations to the short vector/short basis problem. Ajtai compared the following two problems, and shows that an Avg-P solution to the average-case problem implies an RP solution to the worst-case problem.

**Worst-case problem**

Given $L(b_1, b_2 ... b_m) \subset \mathbb{R}^n$, find an $n^{c_1}$-approximate small basis for $L$.

Currently, there is no knowledge about the hardness of this problem.

**Average-case problem**

Given $L^*(b_1^*, b_2^* ... b_m^*) \subset \mathbb{R}^n$, find an $n^{c_2}$-approximate short vector in $L^*$.

The distribution $D$ is as follows:

- Fix $q = n^{c_3}$
- Fix $m = \Theta(n \log q)$
- Randomly choose $(b_1^* ... b_m^*) \in_R \left\{0, \frac{1}{q}, \frac{2}{q}, ... \frac{q-1}{q}, 1\right\}^n$

$L^*$ clearly is an infinite lattice, since $q\mathbb{Z}^n \subset L^*$.

# Chapter 22

| **6.841/18.405J: Advanced Complexity Theory** | May 8, 2002 |
|---|---|
| Lecture 22 | |
| *Lecturer: Madhu Sudan* | *Scribe: Emanuele Viola* |

In this lecture we will talk about Quantum Complexity. However, let's first say some last things about Average Case Complexity.

## 22.1 Average Case Complexity

Average Case Complexity is vastly non understood. One of the main open problems is understanding the average complexity of $3-SAT$. For every $n$ consider the following distribution on instances of $3-SAT$: pick a random formula on $n$ variables with $\Delta n$ clauses, where by a random formula we mean that every clause is chosen uniformly and independently at random among all possible clauses. It can be proven that:

- If $\Delta \geq 6$ then, with probability going exponentially (in $n$) to 1, the formula is not satisfiable.
- If $\Delta \leq 3$ then, with probability going exponentially (in $n$) to 1, the formula is satisfiable.

The threshold seems to be 4.2. In other words, $3-SAT$ with respect to the above distribution for $\Delta = 4.2$ seems to be a good candidate for a problem hard on average.

In practice, there are many heuristics that people use to find satisfying assignments that work well. The drawback is that when the heuristic algorithm fails, we do not know if it failed because there is no satisfying assignment or because it needs more time. Our hope would be an algorithm that provides us with a *proof* that the given formula is unsatisfiable, if that is the case. But do such proofs exist? This question is one of the main questions of *Proof Complexity*, and, modulo distribution on the instances, is the $co-NP$ vs $NP$ question.

Our hope is relating $NP$-completeness and $DNP$-completeness. Some possibility of doing this have already been ruled out. In particular, consider a reduction from some $NP$-hard problem $R'$ to a problem $(R, D)$ (where $D$ is a polynomial time samplable distribution) of the following kind: on input $x$ we reduce the problem of deciding whether $x \in R'$ to the problem of solving $(R, D)$ on, say, four instances $x_1, x_2, x_3, x_4$, where $\forall i\ x_i$ is distributed according to $D$, but they are *not* necessarily independent. [Feigenbaum & Fortnow] show that the existence of such a reduction implies that the $PH$ collapses.

There are at least two ways out of this:

- Consider a 'classical' Turing reduction.
- Try a reduction from some other complexity class, such as *Statistical Zero Knowlegde.*

## 22.2 Quantum Complexity

For a physicist, physics is the goal, while computers are the tool. For a computer scientist is true the opposite: computers are the goal, physics is the tool. In particular, a computer scientist tries to come up with an abstract model of computation, and then show (1) it is physically realizable and (2) it is the strongest possible.

Very influencing has been the

- *Turing-Church Thesis*: every physically realizable computing device can be simulated by a Turing machine.

What is really relevant to Complexity Theory, however, is the

- *Strong Turing-Church Thesis*: every physically realizable *efficient* computing device can be simulated by a Turing machine *with a polynomial time slow-down*.

There have been two main challenges to the Strong Turing-Church Thesis.

The first one comes from *Randomness*: some problems are efficiently solvable (with high probability) using randomness, but are not known to be solvable in deterministic polynomial time. This is the $BPP$ vs $P$ (open) question.

The more recent challenge comes from *Quantum Physics*. We start with an experiment: Consider a wall with two small holes $A$ and $B$. In front of the wall, at equal distance from $A$ and $B$, we put a source of light, and behind the wall we put a screen where we can measure the intensity of the received light.

If we close one hole $A$ or $B$, and let the light go through the other, we measure the same intensity on the screen. However, when opening both holes, the measured intensity is *not* the sum of the intensities previously measured. In particular, at the point on the screen at equal distance from $A$ and $B$ we observe *no light*, while there was some when opening just one hole.

This experiment can be turned in a computational problem as follows: given $n$ walls each with $n$ holes which can be open or closed, a source of light in front of all the walls and a point $P$ on a screen behind all of them: do we measure light in $P$? This problem was considered by Feynman in the 80's.

### 22.2.1 Quantum bits

We define the *state* of a system consisting of one *Quantum Bit* (*qubit*) as a vector $\in \mathbb{C}^2$ of unit length (for this lecture, you can think of it as being a vector with real components). We can think of a qubit as of a coin which has been tossed but has not landed yet. The components of the state represent the probability that this coin will land on head or tails.

Similarly, the *state* of a system consisting of $n$ qubits is a vector $\in \mathbb{C}^{2^n}$ of unit length. Again, we can think of each component of this vector as expressing the probability that the $n$ coins will land in some particular configuration.

What can we *see* of the qubits, and how can we *manipulate* them? These two aspects are referred to as Quantum Measurement and Quantum Evolution, respectively.

### 22.2.2 Quantum Measurement

Consider two qubits in the state $(\frac{1}{\sqrt{2}}, \frac{1}{2}, -\frac{1}{2}, 0)$. We use the following notation (useful because states are often sparse, i.e. most of their coordinates are 0):

$$\frac{1}{\sqrt{2}}|00> + \frac{1}{2}|01> - \frac{1}{2}|10> + 0 \cdot |11> .$$

This means that if we measure the state then with probability $\left(\frac{1}{\sqrt{2}}\right)^2$ we will see 00, with probability $\left(\frac{1}{2}\right)^2$ we will see 01 and so on.

We can also measure one qubit at a time. In this case we follow the rule of *conditional probability.* For example, if we observe the first (leftmost) qubit in the above example, then with probability $1/2 + 1/4$ we see a 0, and then we are in the state

$$\sqrt{\frac{4}{3}} \cdot \frac{1}{\sqrt{2}}|00> \; + \sqrt{\frac{4}{3}} \cdot \frac{1}{2}|01>,$$

while if we see a 1 then we are in the state

$$1|10> .$$

### 22.2.3 Quantum Evolution

Consider some state on $n$ qubits $v \in \mathbb{C}^{2^n}$. For every $2^n \times 2^n$ matrix $U$ over $\mathbb{C}$ which is *unitary* (i.e. $U \cdot U^H = I_{2^n}$, where $U^H$ is the transposed conjugate of $U$) the tranformation $v \to Uv$ is physically realizable.

For example, the matrix $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ negates a single qubit, the matrix $\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}$ flips one (the value of the bit is preserved in the sign, so that we can unflip it), and the general form of these 1-qubit operations is $\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$.

A *quantum circuit* is a circuit where we have such matrices as gates. It can be proven that we can postpone every measurement at the end paying only a little overhead. Rather than computing some function, quantum circuits *sample* from some distribution, and in particular they can sample from some distributions which are not known to be polynomial time samplable.

A *quantum Turing machine* is similar to a classical Turing machine, but the transition function is given by a unitary matrix, representing the quantum evolution of the state. [Bernstein & Vazirani] show that a quantum Turing machine can be initialized in such a way that it can simulate every given circuit. In particular, we can get $\epsilon$ close to the distribution sampled by the circuit with only polynomial time slow-down. Note we cannot ask for the quantum Turing machine to sample *exactly* the same distribution sampled by the circuit, because the circuit may have gates with values (e.g. $\sqrt{3}$) not included in the definition of the quantum Turing machine.

# Chapter 23

| **6.841/18.405J: Advanced Complexity Theory** | May 8, 2002 |
|---|---|
| Lecture 22 | |
| *Lecturer: Madhu Sudan* | *Scribe: Benjamin Morse* |

In this lecture we will cover Quantum Information, Manipulation, Circuits, and Computers. First we will close the topic of Average-case Complexity with some words about unanswered questions in the field.

## 23.1 Average-Case Complexity

Average-case Complexity is for the most part still not understood. Even the following simple problem, which is fundamental to the study of average-case complexity, is still unanswered.

Given a random SAT formula $\phi$ in $n$ variables, with a number of 3-CNF clauses equal to $\Delta n$ for some predetermined constant $\Delta$, where the clauses are picked independantly randomly from a uniform distribution, is the formula satisfiable or not?

From experimental evidence, it's been determined that:

· if $\Delta > 6$, the probability that $\phi \in SAT$ approaches 0.

· if $\Delta < 3$, the probability that $\phi \in SAT$ approaches 1.

Between these two constants, it's harder to tell whether a given formula is satisfiable, and the threshold is around $\Delta = 4.2$. At this threshold, one would imagine that the problem is hard on average, but we have no formal way of showing this.

In practice, there are good heuristics for SAT, but no proofs - if a heuristic algorithm fails, we don't know if it's because there's no satisfying assignment, or if the heuristic simply can't cope with that particular formula. Also, for formulae that take a long time, there isn't a way of knowing if they're taking polynomial or exponential time. There's a lot of work that needs to be done to link theory and practice.

Ideally, we would want some way of relating DNP-completeness and NP-completeness. Feigenbaum and Fortnow have pointed out that one of the more obvious ways of doing this isn't going to work.

Let's assume, as we might like to, that a reduction exists from an NP-hard problem $R'$ to a DNP problem $(R, D)$, where $D$ is a polynomial-time sampleable distribution, such that the randomized reduction of an instance $x$ of $R'$ produces (for example) four random instances $x_1, x_2, x_3, x_4$ of $(R, D)$ such that all $x_i$ are solvable. Each $x_i$ is distributed according to $D$, but they don't need to be necessarily independant of each other. If this is possible, then the polynomial hierarchy collapses.

(Is it necessary that this distribution even exists, though? This is a tricky question - it does exist, but it's possible that it might not be sampleable.)

There are two ways out of this situation: We can follow a generic Turing reduction instead, or we could try using a problem that's complete for a class between P and NP (possibly BPP or 'statistical zero knowledge'), which could still give us an interesting result.

## 23.2 Physics vs. Computation

To the view of the physicist, physics is the goal, and computation is a tool that he can use to use to analyze physics and extract relationships. To the computer scientist, computers are the goal, and physics is a tool she can use to build machines that perform computation. (A fanciful example is using bent wire and a soap solution to form a surface - this solves an extremely complicated differential equation naturally.) The computer scientist comes up with a mathematical model, and tries to prove that it is both physically realizable, and that it's the strongest possible - that it can model all physical processes.

The Turing-Church hypothesis states that every physically realizable computing device can be simulated by a TM. The stronger version of this hypothesis also claims that a TM can do this with only polynomial slowdown. The hypothesis has ben challenged in the past by the introduction of randomness (which Turing machines can't predict). Randomness is a resource that can speed up a process, and challenge the strong hypothesis (i.e. $\exists L \in BPP$ that we only know of an exponential-time deterministic algorithm for). And you can buy computer chips that will use physical processes to give you random numbers, so this could be an effective challenge to the strong version of the hypothesis.

However, the more recent challenge to the strong Turing-Church hypothesis has come with the introduction of quantum computation.

### 23.2.1 The two-slit experiment

We have a wall that can measure the intensity of light hitting it. In front of this wall we place a screen with two slits, and in front of that, a light source. When either of the slits is covered, the wall is illuminated with a fairly predictable Gaussian-like intensity behind the open slit. However, if we uncover both slits, the intensity of the light is banded because of interference, with a node of no light at all between the two slits, instead of the straightforward superposition we would expect. We can describe the 1-screen case with differential equations, but Feynman posed the problem of $n$ screens with $n$ holes in series. Calculating the amount of light that hits that hits the wall at any point is in EXP. This is a case of TMs failing to simulate physical processes with polynomial slowdown, so computer scientists create a new model of computation, with Quantum Turing Machines.

### 23.2.2 Quantum information

A Quantum bit (qubit) can be described by a vector in $\mathbb{C}^2$ of unit length. This differs from a probabilistic bit (which is a real number $p \in [0, 1]$). The two components of this state represent the probabilities of the two possible outcomes - a qubit can, when observed, resolve to either a 0 or a 1.

The state of an $n$-qubit system can be represented by a vector in $\mathbb{C}^{2^n}$. The reason why it can't be represented as a $\mathbb{C}^{2n}$ vector is because of quantum entanglement. Each of the $2^n$ components represent the probabilities that the system will resolve to each of the $2^n$ different possible outcomes.

### 23.2.3 Quantum operations

There are two main operations we can perform on a quantum system. We can manipulate it (which is referred to as Quantum Evolution) or we can observe part of, or all of the system (which is referred to as Quantum Measurement).

Consider a 2-qubit system with state $(\frac{1}{\sqrt{2}}, \frac{1}{2}, \frac{1}{2}, 0)$. This state is described in standard notation as

$$\tfrac{1}{\sqrt{2}}|00> + \tfrac{1}{2}|01> + \tfrac{1}{2}|10> + 0|11>$$

The coefficients to the left of the states are known as the amplitudes.

Using quantum measurement, we can look at part of the system. For example, we can look at the first bit and see how it comes out. The probability we'll get a 1 is $\frac{3}{4}$, and the probability we'll get a 0 is $\frac{1}{4}$. If we get a 0, that leaves the system in the state

$$\sqrt{\tfrac{4}{3}} \cdot \tfrac{1}{\sqrt{2}}|00> + \sqrt{\tfrac{4}{3}} \cdot \tfrac{1}{2}|01>$$

If we observe a 1, that collapses the system to the state

$$1|10>$$

### 23.2.4 Quantum evolution

Consider another $n$-qubit quantum system with state described by $v \in \mathbb{C}^{2^n}$, and a $2^n \times 2^n$ matrix $U$ over $\mathbb{C}$ such that $U$ is unitary (i.e. $U \cdot U^H = I_{2^n}$. $U^H$ is the Hermetian transpose of $U$, which involves transposing the matrix and taking the complex conjugate of all cells.) Quantum evolution can be modelled by taking the state $v$ and calculating $U \cdot v$ for the new state. A helpful unitary matrix for a single qubit is one of the form

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

, which rotates a qubit according to $\theta$. This can be used to negate or flip a single qubit at a time.

### 23.2.5 Quantum circuits

A quantum circuit consists of a $n$-input, $m$-output circuit that has has these unitary matrices as gates. These gates can take any number of qubits as inputs and output the same amount of qubits. The full generalization of quantum circuits allows measurement to take place at any point in the system, but it can be shown that for any circuit there is an equivalent one that postpones measurement until the end with a small overhead.

Quantum circuits do not compute functions per se, but essentially sample from a distribution $D : \{0,1\}^n \to \{0,1\}^m$. These distributions are not necessarily polynomial-time sampleable - there are $D$ that are not poly-size circuits, but are poly-size Q-circuits.

### 23.2.6 QTMs

Does there exist a generalized machine that can simulate any gate arrangement? A QTM is roughly analogous to a deterministic Turing machine. Recall the formal definition of a Turing machine as the tuple

$$\begin{gathered} (Q, \Sigma, \Gamma, \delta, q_0, F) \\ \delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, stay\} \\ q_0 \in Q, F \subseteq Q \end{gathered}$$

A QTM has a very similar definition, except that $\Sigma$ and $\Gamma$ are qubits, and $\delta$ is a unitary matrix. For every quantum circuit $C$ that samples from a distribution $D$, there is a QTM $M$ that simulates it to within any arbitrary $\epsilon$. The reason we can only attain $\epsilon$ closeness is because there can be irrational constants like $\sqrt{3}$ that appear in the gates of $C$, but don't show up in the QTM. We can approximate these constants, though. [Bernstein & Vazirani]

# Chapter 24

| **6.841/18.405J: Advanced Complexity Theory** | May 13, 2002 |
|---|---|
| Lecture 23 | |
| *Lecturer: Madhu Sudan* | *Scribe: Rebecca Hitchcock* |

## Algorithms in Quantum Computing

This lecture covers two algorithms that make use of the quantum computational structures that were set up last lecture. We begin with a review of the Quantum Computing Model.

## 24.1 Recap: Quantum Computing Model

### 24.1.1 Quantum Circuits

A quantum circuit is s circuit with $n$ wires in and $n$ wires out. A $k$-nary gate in this circuit is a $2^k \times 2^k$ matrix which is a map on vectors of size k. At the output of the circuit, we observe some the state by sampling some subset of the output wires.

A quantum circuit can compute any function that can be computed by a classical circuit. We wish to determine what is a sufficient collection of gates to compute any function using a quantum circuit. There are three such gates:

1. The Hadamard Gate $\mathrm{H}_2 = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$. This is a unitary gate which means that $\mathrm{H}_2$ times the complex conjugate of $\mathrm{H}_2^T$ is I, the identity matrix. This is the gate that gives us the power of quantum computing by introducing randomness. If we input $|0\rangle$ into this circuit we get $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. Measuring the qubit will therefore show 0 with probability $\frac{1}{2}$ and 1 with probability $\frac{1}{2}$.
2. A classical negation gate. This gate has two inputs $a$ and $b$ and shows on its output $a$ on the first wire and $a \oplus b$ where $\oplus$ is XOR. The negation is obtained by inputting 1 for $a$ and the input to be negated as $b$.
3. A classical AND gate. This gate has three inputs, $a$, $b$, and $c$. The output of the first two inputs are unchanged, that is they output $a$ and $b$ respectively. The third output gives $c \oplus (a \wedge b)$.

Using these gates we can simulate and polynomial sized classical circuit with a polynomial sized quantum circuit.

### 24.1.2 Quantum Turing Machines

Quantum Turing machines are represented very similarly to classical turing machines. The differences are that the tape has qubits on it instead of regular bits, and the transition function is given by a local unitary matrix. This model leads us to desire a definition of quantum polynomial time. There are two definitions that appear in the literature, BQP and EQP, which are analogous to BPP and ZPP respectively; BQP represents computation with bounded error and EQP represents computation with zero error.

There are two ways to formalize these complexity classes. The first is in the expected way, defining completeness and soundness in the usual ways and allowing a polynomial number of steps on a quantum turing machine.

The second way uses quantum circuits as described above:

**Definition 24.1** **EQP** $= \{L | \exists$ *a family of quantum circuits* $\{Q_n\}_n$*, where* $Q_n$ *takes n inputs, and a classical polynomial time turing machine* $M$ *that on input* $1^n$ *outputs* $Q_n$ *such that:*

$$w \in L \Leftrightarrow Pr\left[Q_{|w|}(w) = 1\right] = 1$$

$$w \notin L \Leftrightarrow Pr\left[Q_{|w|}(w) = 1\right] = 0\ \}$$

**Definition 24.2** **BQP** $= \{L | \exists$ *a family of quantum circuits* $\{Q_n\}_n$*, where* $Q_n$ *takes n inputs, and a classical polynomial time turing machine* $M$ *that on input* $1^n$ *outputs* $Q_n$ *such that:*

$$w \in L \Leftrightarrow Pr\left[Q_{|w|}(w) = 1\right] \geq \frac{2}{3}$$

$$w \notin L \Leftrightarrow Pr\left[Q_{|w|}(w) = 1\right] \leq \frac{1}{3}\ \}$$

This second set of definitions is generally easier to work with. It is important to remember however that the definition requires not only a polynomial sized circuit, but also the ability to build that circuit in classical polynomial time. This enforces a measure of uniformity, which is necessary because as we recall from our study of classical circuits there exist undecidable languages for which there are polynomial sized circuits.

## 24.2 Simon's algorithm

There are three main algorithms that exploit the power of quantum computing. These are Simon's Algorithm, Grover's Algorithm, and Shor's algorithm. Grover's algorithm is for NP search and we will not cover it in this lecture. Both Grover's and Simon's algorithm deal with relativized promise problems.

### 24.2.1 Simon's Problem

Simon was interested in a problem inspired by cryptography. In this problem we are given an oracle for a function $f : \{0,1\}^n \longrightarrow \{0,1\}^n$ and we wish to determine whether or not it is a one-to-one function. He sets up his problem in the form of a promise problem:

**$\Pi_{YES}$:** $\exists s \in \{0,1\}^n/0^n\ s.t\ \forall x\ f(x+s) = f(x)$

$\Pi_{NO}$: $f$ is a one-to-one function

In order to approach this problem classically, you could imagine a world divided into two subsets, x, and x+s. But the space is so large that we are unlikely to see two vectors that map to the same output because there are $2^n$ vectors. This problem is in Promise-NP, but Simon's algorithm gives a method for solving it more efficiently.

In order to discuss Simon's algorithm we need a little more information on the unitary Hadamard Transform.

### 24.2.2 The Hadamard Transform Revisited

Consider a number of Hadamard gates in parallel. If we have six wires coming in, with an input $x = 1010111$ what do we see on the output? Each output wire has equal probability of being one of two choices. The output is the product of the states, so we know the magnitude of the output is $(\frac{1}{\sqrt{2}})^6$. The expression for the general case is as follows:

$$\frac{1}{2^{\frac{n}{2}}} \sum_{y \in \{0,1\}^n} (-1)^{\langle x,y \rangle} |y\rangle$$

Where $\langle x, y\rangle$ represents the inner product modulo two. This operation is very important to Simon's algorithm.

### 24.2.3 Simon's Algorithm

The main idea is to compute a non-trivial function on the $2^n$ inputs. Unlike a trivial function such as the average, we cannot compute this in classical probabilistic polynomial time. Notice here that we are trying to discover properties of the oracle, so we can set the input to whatever we like. Here we set it to $|0^n 0^n\rangle$. Consider the following sequence of transformations(gates):

$$|0^n 0^n\rangle \xrightarrow{H_2^n} \frac{1}{2^{\frac{n}{2}}} \sum_{x \in \{0,1\}^n} |x 0^n\rangle \xrightarrow{f(x)} \frac{1}{2^{\frac{n}{2}}} \sum_{x \in \{0,1\}^n} |x f(x)\rangle \xrightarrow{H_2^n} \frac{1}{2^{\frac{n}{2}}} \sum_{x \in \{0,1\}^n} \frac{1}{2^{\frac{n}{2}}} \sum_{y \in \{0,1\}^n} (-1)^{\langle x,y \rangle} |y f(x)\rangle$$

The first arrow simply gives us the Hadamard transform of the first $n$ qubits. No signs have been introduced at this point because our input was zero, and if we were to go through another Hadamard transform we would get $|0^n 0^n\rangle$ back. The second transformation is possible because of a relativized version of the idea that we can perform and classical computation using a quantum circuit. The details of this are beyond the scope of this lecture. The third arrow is another Hadamard transform on the first $n$ qubits.

Something nontrivial has happened at this point. Consider what happens if we observe the tape in both the yes and no cases:

**NO case:** In this case you see every possible combination of strings $\langle x, y\rangle$. The state of the system is:

$$\frac{1}{2^n} \sum_{y,z} (\pm 1) |y, z\rangle$$

There are no collisions and we are left with a uniformly distributed random sample.

**YES case:** In this case we look at the terms of the sum $2^{-n}(1)^{\langle x,y \rangle}|y, f(x)\rangle$ and $2^{-n}(1)^{\langle (x+s),y \rangle}|y, f(x+s)\rangle$ Since $f(x) = f(x+s)$ we know that these terms are equal, with the possible exception of the $(\pm 1)$. There

are two cases. If $\langle y, s\rangle = 1$ then these two terms will have opposite signs and cancel out. In this case, the corresponding $|y, f(x)\rangle$ will not appear in the output. In the other case, $\langle y, s\rangle = 0$, and they will have the same sign and the resulting coefficient will be $\pm 2 \cdot \frac{1}{2^n}$.
Thus in the yes case we see there are $2^{2n-2}$ possible vectors $\langle y, f(x)\rangle$, and each appears with equal probability.

In order to distinguish these two cases we take $2n$ samples of the circuit, and focus on the $y$ portion. If the set of vectors $y_1 ... y_{2n}$ is of rank $n$, then we know we are in the NO case. Only if we had a 1:1 function could these vectors have full rank. If these vectors are of rank $n-1$, then we know we are in the YES case and using linear algebra we can find a candidate vector for $s$. The algorithm has zero error and the $s$ will be correct with high probability.

## 24.3 Shor's Algorithm

Simon's algorithm gives us exponential speedup over classical algorithms, given the assumption of relativized quantum computations. Shor's algorithm gives us this same speedup on a specific classical problem (factoring), and doesn't rely on this idea. The main idea of the algorithm is inspired by Simon's method.

### 24.3.1 Intuitive Idea

Shor's Algorithm is based on a classical result from the 1970's which says for a factoring a number N reduces to finding the order of a number $a$ mod N. The order of $a$ is the place in the sequence $a^0, a^1, a^2, ..., a^r$ where $a^r \equiv 1(\text{mod N})$.This reduction is based on factoring $a^r - 1 \equiv 0(\text{mod N})$ to $(a^{\frac{r}{2}} - 1)(a^{\frac{r}{2}} + 1) \equiv 0(\text{mod N})$. With high probability, r is even and neither of the terms of the product are zero.

Since Simon's Algorithm seems to compute periods, Shor felt that it might be possible to use the ideas in it to find order. We can find an $s$ s.t.

$$\forall x\; f(x) = f(x \cdot s)$$

Where the $\cdot$ can be any group operation. We can phrase the problem of finding the order of $a$ in the same language.

$$f(i) \stackrel{\triangle}{=} a^i$$

$$\forall i\; f(i) = f(i + r)$$

### 24.3.2 A New Unitary Operation

We introduce here a new operation which we claim to be unitary for all choices of $q$. This operation computes the complex Fourier transform and its expression is given below:

$$|j\rangle \longrightarrow \frac{1}{\sqrt{q}} \sum_{k=0}^{q-1} e^{\frac{2\pi i}{q} \cdot j \cdot k} |k\rangle$$

Although this operation is always unitary, it may not always be local. We ignore this problem for now, and return to it briefly at the end of the lecture.

### 24.3.3 Shor's Algorithm

Fix $a$, N, and some large $q$. Consider the following sequence of transformations (gates). The operation described above is notated as "O" and the input is two zero vectors from $\mathbb{Z}_q$

$$|00\rangle \xrightarrow{O} \frac{1}{\sqrt{q}} \sum_j |j0\rangle \overset{\text{modular exp.}}{\longrightarrow} \frac{1}{\sqrt{q}} \sum_j |jf(j)\rangle \xrightarrow{O} \frac{1}{q} \sum_j \sum_k e^{\frac{2\pi i}{q} \cdot j \cdot k} |kf(j) \longrightarrow \text{Observe State}$$

Note that this is very similar to the progression made in Simon's Algorithm. In the second step, however, we do not need to rely on relativized arguments to see that this can be computed using a quantum circuit. It is simply a matter of converting a classical circuit into a quantum one.

**Claim 24.3** *$k$ is very close to a multiple of $\frac{q}{r}$, thus, $\frac{q}{k}$ gives a good approximation of $r$.*

Assume that $q$ is a multiple of $r$. Then we have:

$$\sum_{j_1=0}^{q-1} \sum_{j_2=0}^{r-1} \sum_k e^{\frac{2\pi i}{q}(rj_1+j_2)k} |kf(j_2)\rangle = \sum_k \sum_{j_2} |kf(j_2)\rangle \cdot e^{\frac{2\pi i}{q} j_2 k} (\sum_{j_1=0}^{\frac{q}{r}-1} e^{\frac{2\pi i}{q} r j_1 k})$$

Consider the last term of the product, with $m$ set equal to $\frac{q}{r}$.

$$\sum_{j_1}^{m-1} (e^{\frac{2\pi i}{m} k})^{j_i}$$

We see some $m^{th}$ root of unity, depending on the value of $k$. If $k$ is a multiple of $m$, we get $m$ for the value of the sum and if not we get 0. We find $r$ by observing many samples and then using a gcd calculation to find $\frac{q}{r}$ from a number of multiples of $\frac{q}{r}$.

### 24.3.4 Remaining Details

There are two main ideas that we glossed over in this discussion of Shor's Algorithm.

1. **q is not necessarily a multiple of r**. We handle this using analysis such that $[kr]_q$ is a very small contribution. We are then left with a problem that can be solved using an integer program in two variables.

2. **A q-ary Fourier Transform is not always local** If we pick $q$ to be a power of 2, then in this case we can construct a small quantum circuit implementing $q$-ary FT.

# Chapter 25


**6.841/18.405J: Advanced Complexity Theory** May 13, 2002

Lecture 23

*Lecturer: Madhu Sudan* *Scribe: Alice Chan*


## 25.1 Today

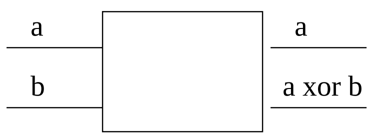
Recap of quantum computing model, Simon's algorithm, Shor's algorithm for factoring.

### 25.1.1 Quantum circuits

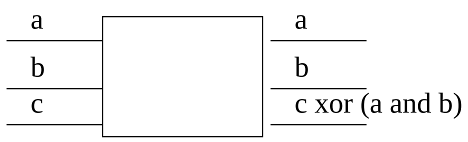
These are circuits with $n$ different wires combined using quantum gates. A quantum gate is a map from $2^k \to 2^k$. The Hadamard transform,"negation" and "and" form a sufficient collection of gates.

The Hadamard transform, $H_2$: $\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$,

"negation"




"and"




The transition function for a quantum TM is a transition matrix and the tape consists of q-bits.

"Quantum polynomial-time"

BQP is, in some sense, an extension of BPP and EQP an extension of ZPP. BQP is the class of languages that can be solved with a polynomial number of steps on a quantum TM, with completeness and soundness as defined before. An equivalent definition is that BQP is the class of languages that can be solved by a polynomial-sized quantum circuit which is constructible in classical polynomial-time.

### 25.1.2 Simon's algorithm

This algorithm is for a promise problem where we are trying to decide whether a function $f$ is 1-1.

The oracle is the function $f : \{0,1\}^n \to \{0,1\}^n$.
A YES instance is the case when $f$ is not 1-1, ie. $\exists s \in \{0,1\}^n - \{0\}^n$ s.t. $\forall x f(x+s) = f(x \oplus s) = f(x)$. If such an $s$ exists, $f$ is at approximately 2-1.
A NO instance is the case that $f$ is 1-1.
Suppose $x = |010111>$ and apply $H_2$ to each of the bits. The outcome is $\frac{1}{2^{\frac{n}{2}}} \sum_{y\in\{0,1\}^n} (-1)^{<x,y>}|y>$. Each of the $2^6$ possible outcomes has equal probability of occuring.

**Simon's algorithm:**
Initialize the quantum circuit to $|0^n, 0^n>$.
Apply $H_2$ to the first $n$ bits and get $\frac{1}{2^{\frac{n}{2}}} \sum_{x\in\{0,1\}^n} |x, 0^n>$.
Set the machine to $\frac{1}{2^{\frac{n}{2}}} \sum_{x\in\{0,1\}^n} |x, f(x)>$
(as classical computation can be simulated in the quantum world).
Undo the Hadamard computation.
(If the Hadamard computation was undone at the stage with $\frac{1}{2^{\frac{n}{2}}} \sum_{x\in\{0,1\}^n} |x, 0^n>$ we get $|0^n, 0^n>$.
But with $\frac{1}{2^{\frac{n}{2}}} \sum_{x\in\{0,1\}^n} |x, f(x)>$, the result is $\frac{1}{2^{\frac{n}{2}}} \sum_{x\in\{0,1\}^n} \frac{1}{2^{\frac{n}{2}}} \sum_y (-1)^{<x,y>}|y, f(x)>$. )
Observe the tape.

In the NO instance, every string $<y, z>$ is observed in $|y, f(x)>$ since $f$ is 1-1. So the state, $\frac{1}{2^n} \sum_{y,z} (\pm 1)|y, z>$, is a uniformly distributed random sample.
In the YES case, $f$ is approximately 2-1, so $f(x)$ will only take on $2^{n-1}$ values.
If $<y, s>= 1$, then

$$(-1)^{<x,y>}|y, f(x)> + (-1)^{<x+s,y>}|y, f(x+s)>= (-1)^{<x,y>}(|y, f(x)> + (-1)|y, f(x)>)$$

as $f(x+s) = f(x)$.
If $<y, s>= 0$, then all possible $2^{2n-2}$ vectors $|y, f(x)>$ are seen with equal amplitude. (There are $2^{2n-2}$ possibilities because half of the vectors are ruled out since $f$ is 2-1 and half of the remaining are ruled out because $<y, s>= 0$.)
Sampling from this circuit $2n$ times and writing the results $y_1, \ldots, y_{2n}$ as a matrix, either we get $y_1, \ldots, y_{2n}$ of rank $n$ in the NO case or we get a rank of $n-1$ for the YES case.

### 25.1.3 Shor's algorithm

Intuition: Given $n$, pick a random $a \in \mathbb{Z}_n^*$. Then factoring $n$ reduces to computing the order of $a \bmod n$ (finding $r$ such that $a^r - 1 \equiv 0 \bmod n$). Simon's algorithm seems to compute periods of functions so perhaps it can be used to compute the period of the order function $f(i) = a^i$, ie. it can find $r$ such that $f(i+r) = f(i)$.
Fix $a, n$ and some $q$. Let $j \in \mathbb{Z}_q$ and define a unitary operator $|j> \mapsto \frac{1}{\sqrt{q}} \sum_{k=0}^{q-1} e^{\frac{2\pi i}{q} j*k}|k>$, similar to a complex Fourier transform.

**Shor's algorithm:**
Initialize the state to $|0, 0>$.
Apply the unitary operator above to the first half and get $\frac{1}{\sqrt{q}} \sum_j |j, 0>$.
Set the machine state to $\frac{1}{\sqrt{q}} \sum_j |j, f(j)>$, where $f$ is the order function.
Apply the unitary operator to get $\frac{1}{q} \sum_j \sum_k e^{\frac{2\pi i}{q} j*k}|k, f(j)>$.
Observe state.

Claim: $k$ is very close to a multiple of $[\frac{q}{r}]$.
(Proof omitted.)

Assume $q = mr$ for some $m$.

Writing out $\frac{1}{q}\sum_j \sum_k e^{\frac{2\pi i}{q} j*k}|k, f(j) >$ as

$$\frac{1}{q}\sum_{j_1=0}^{\frac{q}{r}-1}\sum_{j_2=0}^{r-1}\sum_k e^{\frac{2\pi i}{q}(rj_1+j_2)*k}|k, f(j_2) >= \sum_k \sum_{j_2} |k, f(j_2) > e^{2\pi*i*j_2*k}(\sum_{j_1=0}^{\frac{q}{r}-1} e^{\frac{2\pi i}{q} r*j_1*k})$$

$$= \sum_{j_1=0}^{m-1}(e^{\frac{2\pi i}{m}k})^{j_1} = \begin{cases} m & \text{if } k \text{ is a multiple of } m, \\ 0 & \text{otherwise.} \end{cases} \tag{25.1}$$

Major issues:
1) $q$ is not a multiple of $r$:
Get $k$ such that $[kr]_q$ is very small contribute (handled by extending analysis and applying integer programming in $O(1)$ variables).
2) $q$-ary Fourier transform is not always local:
In the case where $q$ is a power of 2, can construct a small quantum circuit implementing any $q$-ary FT.

# Chapter 26

| **6.841/18.405J: Advanced Complexity Theory** | May 15, 2002 |
|---|---|
| Lecture 24 | |
| *Lecturer: Madhu Sudan* | *Scribe: Johnny Chen* |

## 26.1 Quantum Computing Wrap-Up

Recall that in the last lecture we saw Simon's algorithm and Shor's factoring algorithm. Today we consider some other issues and offer concluding remarks.

### 26.1.1 Grover's Algorithm

Suppose we have oracle access to a function $f : [N] \to \{0, 1\}$ (here $[N] = \{1, \ldots, N\}$), and we want to know if there exists a $x$ such that $f(x) = 1$. This is a prototypical NP-complete problem, so we do not hope to solve in $O(\log N)$ time with a classical model of computation. But what about quantum computers? It turns out that a quantum algorithm needs $o(\sqrt{N})$ sequential queries to decide this problem [Bennett, Brassard, Bernstein, Vazirani]. Grover proved this bound was tight by giving an algorithm using $O(\sqrt{N})$ time, so it seems that $\mathrm{NP}^f \subsetneq \mathrm{BQF}^f$. The algorithms of Simon, Shor and Grover therefore demonstrate that quantum computing is definitely an interesting *model* to consider.

### 26.1.2 Problems and Pitfalls

It remains to be seen if a quantum computer can actually be built. Here we will consider some challenges.

**Error Correction**
Suppose we measure bits from a circuit by taking 0 to be -5V and 1 to be +5V. For small perturbations we can still take accurate measurements: for example, we take +4.9V to be a 1. However, these small perturbations accumulate, making the final measurements unreliable. In the classical model of computation two things save us and allow for error correction. First is the transistor, which in our example naturally pushes output close to $\pm 5V$. The second is a simple mechanism in which we encode 0s and 1s by repetition, so that 0 becomes $0 \cdots 0$ for example. As some of the 0s are corrupted, we can correct these errors by periodically considering the corrupted strings and changing them to $0 \cdots 0$ or $1 \cdots 1$ via majority vote. While this seems rather basic, it is crucial to classical computation, and not possible in quantum computation! We cannot measure periodically for self correction because the states will then collapse to a single one, destroying linear superposition. Also, since processes must be reversible, we cannot destroy the string.

Shor gave an elementary quantum error-correcting code that corrects errors without destroying or measuring the message. We will not discuss it, but it suffices to say results followed [AharanovBen-Or], [Shor], [Kitaev] that consider what it means to build a fault-tolerant quantum computer.

**Gates with Low Error**
One assumption permeating quantum computing is that quantum gates can be built with error probability at most $10^{-4}$. With this assumption we can assemble millions of such gates and perform real computations. As yet, no one has been able to build even one gate with this accuracy.

**Initializing and Resetting**
In our description of quantum computation model, we have assumed that we can reset the input to a gate to 0s (erasing the work tape). But how is this done in a quantum computer? It turns out that this is a nontrivial problem.

Physicists would say, it depends on the model. One particular model uses *Nuclear Magnetic Resonance* (NMR), where cooling the circuit has a "cleaning" effect. However, if we wish to have 0 with probability $1-p$ and 1 with probability $p$, the cost of cooling grows *exponentially* with $\frac{1}{p}$. So to set $n$ bits to 0 requires $\Omega(2^n)$ energy; this erases the advantage of quantum computing. A proposed solution was given by Schulman and Vazirani. They give an efficient procedure where circuits are cooled, and through preprocessing bits are separated into "very cold" (pure 0s) and "very hot" to maintain entropy.

**Other Work**
The areas of quantum communication, quantum information theory, quantum cryptography and quantum complexity are all quite active.

## 26.2 Complexity Wrap-Up

What have we learned this semester? Madhu's view falls into two broad categories.

### 26.2.1 Lower Bounds

Some techniques we have seen are:

- Diagonalization
- Circuit Complexity
- Communication Complexity

Admittedly, lower bounds were not discussed so much this semester, because complexity theorists have not been able to prove much (compared to the types of questions still open).

### 26.2.2 Identifying Computational Themes

We discussed some applications of Complexity theory.

- We abstracted some notions of games like Go, chess or Mahjongg and asked, for example, how does playing Go against a master compare to computer Mahjongg?
- A repeating theme was to identify a computational resource, define a complexity class and find a complete problem for this class.
- We explored some notions of proof, notably PCPs.

### 26.2.3 Topics Not Covered

**Proof Complexity: Resolution**
If we are given an unsatisfiable formula $\phi(x_1, \dots, x_n)$, how can we prove that it is unsatisfiable? This is not an implausible task: consider the formula

$$\phi(x_1, x_2, x_3, x_4) = x_1 \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee x_3 \vee x_4) \wedge \overline{x_3} \wedge \overline{x_4}.$$

From the first two clauses, $x_2$. Combining this with the third clause gives $x_3 \vee x_4$. With the fourth clause, we have $x_4$, contradicting the last clause. Therefore $\phi$ is not satisfiable. Such an argument is called a *resolution derivation*.

Certainly we do not expect a short resolution proof on all instances, since this would imply co-NP=NP. An active area of research involves proving lower bounds for resolution. The first such papers appeared in the 80s [Haken85].

**Randomness**
Complexity Theory has been quite successful at answering questions relating to randomness. The main consideration in this area is the difference between *actual randomness*, measured statistically, and *apparent randomness*, i.e., randomness required for computation.

Physicists believe randomness is everywhere. But how expensive is it? Consider three "worlds":

1. **Randomness is cheap**
   In this world, we can build randomized algorithms everywhere, and BPP rules.

2. **Randomness is "somewhat" expensive**
   In this world, randomness comes at the cost of polynomial slowdown. Therefore we need to reduce and recycle randomness. This area has had some nice results. For example, suppose that with a BPP algorithm $A$, $n$ random bits gives us error probability $1/4$. How many random bits do we need for error $1/16$? An immediate upper bound is $2n$, but it turns out that even with $A$ as a black box, $n+2$ random bits suffice. Indeed, for error $1/2^k$ we need $n + O(k)$ bits, where the constant has been pushed to $2 + \epsilon$ for some $\epsilon$.

3. **Randomness is infeasibly expensive**
   In this world, we must consider *pseudorandomness* - generating efficient objects with little or no randomness that are computationally indistinguishable from truly random objects. An example of a result in this area is the following

   **Theorem 26.1 (Impagliazzo, Widgerson)** *If every $f \in \boldsymbol{E} = DTIME(2^{O(n)})$ that has circuit complexity $2^{\Omega(n)}$ then there exists a pseudorandom generator ($\Rightarrow$ BPP=P).*

   A related question in this area is, what form does randomness take? We usually think of a random source as being a sequence of random, independent coin flips. What if the source is "dirtier"? Specifically, for a random variable $X$ on a set $\Omega$ define the min-entropy $H_\infty = \min_x \log_2 \Pr[X = x]^{-1}$. We will think of a source $X$ as "dirty" if we just know that $H_\infty = k$. The question becomes, can we produce pure random bits from a dirty source? It turns out that we cannot - but if use a purely random seed, we can. These objects are called extractors. One of the best extractor constructions uses $n$ bits from the source, a seed of length $\log n$ and outputs $m = \epsilon k$ purely random bits (again, $k$ is the min-entropy of the source). There turns out to be a fascinating connection between extractors and pseudorandom generators [Trevisan].

**Knowledge**
How do we formalize the notion of knowledge? Consider a message from Bob to Alice saying "I flipped a coin, and it landed heads". Intuitively, we know that Alice has received no knowledge from this statement: she has learned of the existence of a coin that can land heads up, a fact she could have verified herself. But

when can we say knowledge has been exchanged in interaction? Indeed, the precise notion of knowledge was a barrier for those designing cryptographic protocols. The seminal work of Goldwasser, Micali and Rackoff [GMR] defined knowledge and zero-knowledge proofs (ZKP). The idea is that Alice has not learned anything from Bob's statement because she could have simulated it herself. While she may not be able to simulate it exactly, she can simulate some kind of distribution.