CS 15-855 Computational Complexity Theory **Fall 2000**

Scribe Notes

Professor: Steven Rudich *TA: Ke Yang*

CS 15-855 Computational Complexity Theory **September 13, 2000**

Lecture 1: The Formal Foundations of Complexity

Lecturer: Rudich *Scribe: Steven Rudich / Editor: Steven Rudich*

Synopsis: Definitions of one-tape and multi-tape Turing machines, their computation and operation. Time and space classes. Linear-speedup theorem. Simulation and Diagonalization: the hierarchy theorems.

1 What is Complexity?

Computational complexity theory is the formal study of what can and *what can't* be computed using bounded resources. The resources which will receive the most attention in this class will be space and time. Other resources include parallel-time, time-space product, random bits, interaction, volume of communication, and advice from god.

2 Basic Definitions

For completeness, we start at the beginning. For many this will be a fast review of the formal foundations of Turing machine time and space complexity classes. For others, this material will be new. In which case, the lecture will be too quick to absorb and should be carefully studied out of class. Practical and philosophical considerations which motivate our choices will be deliberately postponed until the third lecture.

2.1 Turing Machines

The Turing machine (TM) was first defined by Alan Turing in his landmark 1937 paper[2]. There are many variations on the details of its definition. Let us agree on what we will mean for the duration of the class. A *one-tape Turing machine* is a computing device with a finite state control, a tape, and a tape head which points to (scans) a given cell on the tape. The tape is an infinite sequence of cells starting

at the left. We initialize the machine by putting the finite control into a special start state q_{start} and writing our input to the machine on the tape (starting from the leftmost cell). All other cells on the tape will contain the special blank symbol $\Box$. At each time step the machine makes a move which is a function of the state of the finite control and of the symbol being scanned by the tape head: it changes its control state, overwrites the symbol on the cell being scanned, and moves the head one to the right or left. If the machine ever enters the special state q_{accept} (q_{reject}) it halts and is considered to have accepted (rejected) the input.

More formally, a Turing machine M is a 3-tuple (Σ, Q, δ). Σ contains the blank symbol ($\Box$) in addition to some finite alphabet of tape symbols. Q contains q_{start}, q_{accept}, q_{reject}, and a finite set of control states. δ is the transition function for M. δ is a function from $\Sigma \times Q$ to $\Sigma \times Q \times \{\rightarrow, \leftarrow\}$. A *configuration* C of M consists of the tape contents up to the rightmost non-blank symbol, the state of the control, and the position of the head. A configuration is represented as a string of symbols from $\Sigma \cup Q$: $\sigma_1\sigma_2\sigma_3 \ldots \sigma_i q \sigma_{i+1} \ldots \sigma_l$, where $\sigma_1\sigma_2 \cdots \sigma_l$ are the tape contents, q is the state of the finite control, and σ_{i+1} is the symbol being scanned by the tape head.

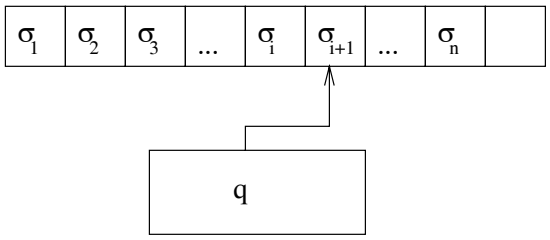


Figure 1: The Turing machine state represented by $\sigma_1\sigma_2\sigma_3 \ldots \sigma_i q\ \sigma_{i+1} \ldots \sigma_n$

We can now define how the TM M *computes* on an input string $x = x_1x_2x_3 \ldots x_n$. M is initialized to the configuration $C_0 = q_{start}x_1x_2 \cdots x_n$. At the ith time step the TM applies δ to C_{i-1} to derive the configuration C_i: δ is applied to the current state and the symbol being scanned. The result is a 3 tuple $\langle \sigma, q, d \rangle$: σ is written on the current cell, the finite control is changed to be in state q, and (unless it will fall off the left) the head is moved one cell in direction d. If at the kth step the machine enters q_{accept}(or q_{reject}), we say that the machine halts and accepts (or rejects) input x in k steps. The *computation* of the machine on x is the sequence of configurations $\langle C_0, C_1, \ldots, C_k \rangle$.

Example: Let **PALINDROMES** be the set of strings over $\{a, b\}$ that read the same backwards and forwards. We can build a TM to accept exactly the strings in this set. $\Sigma = \{a, b, +, \Box\}$. $Q = \{q_{start}, q_{accept}, q_{reject}, q_a, q_{a'}, q_b, q_{b'}, q_l\}$. We now have to

define δ on all its values:

$$\begin{array}{lll}
\delta(a, q_{start}) = (+, q_a, \rightarrow) & \delta(b, q_{start}) = (+, q_b, \rightarrow) & \delta(\square, q_{start}) = (\square, q_{accept}, \rightarrow) \\
\delta(a, q_a) = (a, q_a, \rightarrow) & \delta(b, q_a) = (b, q_a, \rightarrow) & \delta(\square, q_a) = (\square, q_{a'}, \leftarrow) \\
\delta(a, q_{a'}) = (\square, q_l, \leftarrow) & \delta(b, q_{a'}) = (b, q_{reject}, \rightarrow) & \delta(+, q_{a'}) = (+, q_{accept}, \leftarrow) \\
\delta(a, q_b) = (a, q_b, \rightarrow) & \delta(b, q_b) = (b, q_b, \rightarrow) & \delta(\square, q_b) = (\square, q_{b'}, \leftarrow) \\
\delta(a, q_{b'}) = (a, q_{reject}, \leftarrow) & \delta(b, q_{b'}) = (\square, q_l, \leftarrow) & \delta(+, q_{b'}) = (+, q_{accept}, \leftarrow) \\
\delta(a, q_l) = (a, q_l, \leftarrow) & \delta(b, q_l) = (b, q_l, \leftarrow) & \delta(+, q_l) = (+, q_{start}, \rightarrow)
\end{array}$$

The machine starts by changing the leftmost symbol to a "+" and moving the head to the rightmost symbol and changing it to a blank (if the symbols fail to match, the machine rejects). It moves back to the right of the plus and repeats the process until all symbols have been matched. We leave as an exercise to work out the sequence of 11 configurations the machine enters during its computation on input *aba*.

A *multi-tape TM with input and output tapes* operates on the same basic principle as the one-tape TM, but has a few differences. It has a special *input only tape* which is read-only and a special *output only tape* which is write only. For work space, it has some constant number of *work tapes* just like the tape on the one-tape machine. The input is, of course, initially written on the input tape. At each time step the machine makes a move depending on all the (constant) number of symbols being scanned by the tape heads and the state of it finite control. Each head overwrites the cell it is scanning and moves left or right one cell. Termination conditions are the same. The formal description is analogous to the one-tape case. The δ function now depends on tuples of alphabet symbols as well as the current state.

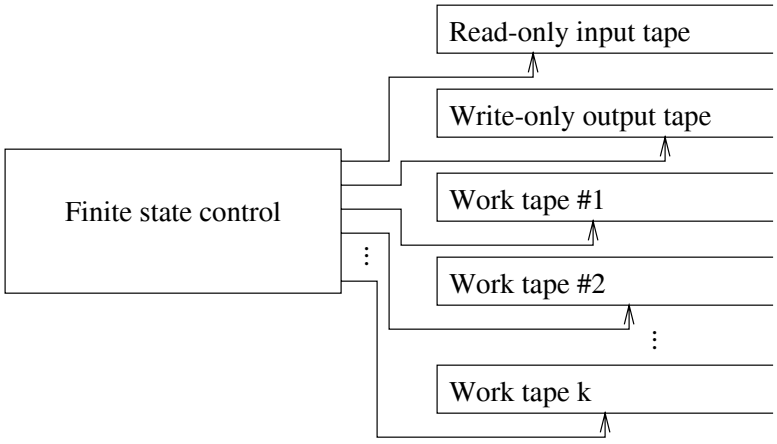


Figure 2: A k tape multi-tape Turing Machine

Convention: Unless we say otherwise, when we say Turing machine we will mean MULTI-TAPE TM WITH INPUT AND OUTPUT TAPES.

Definition 2.1 *A set (language) S is* **recognized (or decided)** *by a TM M if M accepts every string in S and rejects every string not in S.*

Definition 2.2 *A set (language) S is* **accepted** *by a TM M if M accepts every string in S.*

Notice that, unlike a recognizer, an acceptor for a set might not halt on all inputs.

Definition 2.3 *A set S is* **recursive (or decidable)** *if there exists a TM which recognizes it.*

Definition 2.4 *A set S is* **recursively enumerable (or r.e.)** *if there exists a TM which accepts it.*

2.2 Time and Space

Let f be any function from $\mathbb{N}$ to $\mathbb{N}$. The notation $|x|$ means the length of the string x. We will define our time and space classes in terms of the sets which can be recognized using no more than a specified amount of a resource.

Definition 2.5 *$A \in \mathsf{TIME}(f(n))$ if there is a multi-tape TM which recognizes A and uses no more than $f(|x|)$ steps on input x.*

Definition 2.6 *$A \in \mathsf{SPACE}(f(n))$ if there is a multi-tape TM which recognizes A and uses no more than $f(|x|)$ cells of its work tapes during its computation on input x. A cell is considered used if it is scanned by the tape head.*

We can define time and space for function computation as well.

Definition 2.7 *$r \in \mathsf{FTIME}(f(n))$ if there is a TM which on input x writes $r(x)$ on its output tape within $f(|x|)$ steps.*

Definition 2.8 *$r \in$ **FSPACE**$(f(n))$ if there is a TM which on input x writes $r(x)$ on its output tape and uses no more than $f(|x|)$ cells of its work tapes.*

Example: **PALINDROMES** is in $\mathsf{TIME}(2.5n)$ and is also in $\mathsf{SPACE}(3 \log n)$. It is in $\mathsf{TIME}(2.5n)$ because a TM can copy the input onto its work tape (time n), send the input head back to the beginning (time n), and then compare the two sides of the string as the input head marches forward and the work tape head marches backwards (time $n/2$). It is in $\mathsf{SPACE}(3 \log n)$ because we can store 3 pointers into the input string. One pointer will always contain the current head position. One pointer will start at 1. We run our input head to the end and set the third pointer to the position of the last input symbol. We then run the head back and forth keeping our last two pointers up to date concerning which pair of symbols is being checked.

Question: Can **PALINDROMES** be recognized by a machine which runs simultaneously in linear time and logarithmic space? Neither of the two algorithms above has this property. We will be able to answer this question in the next lecture.

3 Basic Theorems

3.1 Linear Speedup and Compression

The next two theorems show us that multiplicative constants don't matter when we're talking about complexity classes.

Theorem 3.1 (Linear Speedup) *Suppose $M(x)$ runs in $f(|x|)$ steps. For any $\epsilon > 0$, we can build an equivalent machine M' that runs in $\epsilon f(n) + n$ steps.*

Proof: Fix k as large as desired. M' has one more work tape than M (and a much larger finite control and tape alphabet). M' copies its input onto its extra work tape in compressed form: one tape-symbol for every k-tuple of M's tape symbols (thus the need for a much larger tape alphabet). This initial copying and compression takes n steps. M' will then simulate M so as to maintain a compressed representation of all its work tapes.

M' will make 6 moves to simulate k of M's moves. By moving each of its tape heads one right, two left, and one right again while storing those symbols in its finite control,

M' now knows (in its control) every symbol within radius k of any of M's tape heads. Thus, M' knows M's next k moves. By using 2 additional moves, M' can modify its compressed symbols so as to represent the configuration of M after those k moves. ■

Corollary 3.2 *If A can be recognized in cn time for $c > 1$, then for any $\epsilon > 0$, $A \in \mathsf{TIME}((1+\epsilon)n)$.*

Corollary 3.3 *Suppose $n = o(f(n))$. If A can be recognized in $\mathrm{O}(f(n))$ time, then $A \in \mathsf{TIME}(f(n))$.*

Theorem 3.4 (Linear Compression) *Suppose $M(x)$ runs in $f(|x|)$ space. For any $\epsilon > 0$, we can build an equivalent machine M' that runs in $\epsilon f(n) + 2$ steps.*

Corollary 3.5 *If A can be recognized in $\mathrm{O}(f(n))$ space, then $A \in \mathsf{SPACE}(f(n))$.*

The proof for the case of space is left as an exercise.

3.2 Simulation

3.2.1 Proper Complexity Functions

Before speaking about the simulation of a given machine for $f(n)$ steps, we will address a technical point. If we are going to simulate a machine for $f(n)$ steps, we had better be able to calculate $f(n)$. Since we will want our simulations to be efficient, it would be counter-productive if the calculation of $f(n)$ required more than $f(n)$ time or space. The *reasonable* $f(n)$ (e.g., $\log n$, $n^{\log n}$, 2^n) that people actually use as bounds for complexity classes don't pose any problem. This consideration motivates a formal definition of which functions are appropriate to use as resource bounds.

Definition 3.1 *$f(n)$ is a* proper complexity function *if $f(n) \geq f(n-1)$ and there is a TM M which for every input x outputs a string of length exactly $f(|x|)$ and runs in time $\mathrm{O}(|x| + f(|x|)$ and space $\mathrm{O}(f|x|)$.*

Why do all the normal functions have this property? The following exercise will help to explain why. Show that if $f(n)$ and $g(n)$ are proper complexity functions then so are $f(g(n))$, $f(n)+g(n)$, $f(n)g(n)$, and $2^{g(n)}$.

Proposition: If $f(n)$ is a proper complexity function then languages recognized in time $f(n)$ are the same as those accepted in time $f(n)$.

Proof left as exercise.

Convention: Unless we say otherwise, ALL COMPLEXITY FUNCTIONS CONSIDERED IN THIS CLASS WILL BE ASSUMED TO BE PROPER COMPLEXITY FUNCTIONS.

3.2.2 Representing TM's as Strings

Any string or number can be viewed as a TM and vice versa. This is a simple point, but important enough to explain. Let's fix a convention for how a TM (Σ, Q, δ) is represented as a string over $\{0, 1, \#\}$. Assign a distinct binary sequence of length k to each symbol in Σ and Q. k is chosen as the least number that will allow us to do this. The assignment is otherwise arbitrary except that q_{start}, q_{accept}, and q_{reject} are assigned the sequences 0^k, $0^{k-1}1$ and $0^{k-2}10$, respectively. By "list Σ" we will mean listing the binary codes for the elements of Σ separated by #'s (similarly for "list Q"). A transition rule in δ such as $\delta(a, b, c, q) = (d, \leftarrow, e, \leftarrow, f, \rightarrow, q')$ can be listed as:

$$a\&\#b\&\#c\&\#q\&\#d\&1\#e\&1\#f\&0\#q'\&$$

where $x\&$ means the binary code for x. To list the entire TM description we just list Σ, place ##, list Q, place a ##, list δ, and place an end-marker ##. Notice that given the string, we can unambiguously reconstruct the TM which it represents. Also notice that since every such string is a number in base 3, we have a method to calculate the precise correspondence between i and the ith Turing machine. Of course, for most i the ith Turing machine is not even syntactically well formed, but it can be view as a Turing machine which does nothing. (Formally, we can equate it with whichever particular TM we wish.)

3.2.3 The Simulation Theorem

The following is the formal version of the fact that we can easily write an efficient interpreter (such BASIC or ML) that runs any given program-input pair for a fixed

number of steps.

Convention: In situations where we give a TM M as input to another machine, it should be understood that we mean the string which represents M.

Also notice that because of the endmarker in our representation of TM M, we can feed Mx to another TM and expect it to be able to parse it into the M portion and the x portion.

Theorem 3.6 (Clocked Simulation) *For every proper complexity function $f(n)$, there exists a TM S_f running in time $f^3(|Mx|)$ such that on input Mx, S_f accepts iff $M(x)$ accepts within $f(|x|)$ steps.*

Proof sketch: S_f will have 4 work tapes. On one, it ticks off $f(|x|)$ cells to make a "clock". Notice that because $f(n)$ is a proper complexity function we know that this will not take very long. Another tape will be devoted to the contents and head positions of M's work tapes. A third tape will store M's program and the current state of its finite control. Keep the forth tape for work space.

S_f scans the entire tape holding M's work tapes. As it does so, it writes the information about which heads are scanning which symbols on its scratch tape. Now S_f can scan M's program and determine the result of the next move. S_f makes the updates and removes one "tick" from the clock tape.

If the clock ever runs out or if M rejects, then S_f rejects. If M ever accepts, then S_f does too.

The time spent simulating each of the $f(|x|)$ steps of M could be as much as $\mathrm{O}(f(|Mx|)^2)$ because during an update the information on the tape doing an update might have to be moved over as many as $\Omega(f(|x|)$ cells. The total simulation can be compressed as in Theorem 3.1 from $\mathrm{O}(f^3)$ to f^3 steps. ∎

In fact, by being much more clever about the details of the simulation it is possible to show a better simulation theorem[1]. We state it without proof.

Theorem 3.7 (Clocked Simulation) *For every proper complexity function $f(n)$ and $g(n) = \omega(f(n) \log f(n)$, there exists a TM S_f running in time $g(|Mx|)$ such that on input Mx, S_f accepts iff $M(x)$ accepts within $f(|x|)$ steps.*

The optimal version for space is much easier and is left as an exercise.

Theorem 3.8 (Bounded Space Simulation) *For every proper complexity function $f(n)$, there exists a TM S_f running in space $f(|Mx|)$ such that on input Mx, S_f accepts iff $M(x)$ accepts using only $f(|x|)$ space.*

3.3 Diagonalization and the Hierarchy Theorems

The brilliant technique Cantor used to argue the uncountability of the real numbers is called *diagonalization*. It was also used to show the undecidability of the halting problem and Gödel's incompleteness theorem. We can import the technique into complexity theory. The idea is quite simple. Suppose that a machine M in time class A can simulate any machine in time class B. We will build a diagonalizer machine D. On input x, D will use M to simulate $x(x)$ and do the opposite of what it does. (This is called "diagonalizing against x".) It follows that if x is in time class B,[1] M gave a correct simulation of x, and hence $D(x) \neq x(x)$. The language recognized by D, which is in time class A, is different on at least one string from any language in time class B.

Theorem 3.9 (Time Hierarchy) *For any proper complexity function $f(n)$ and $g(n) = \omega(f(n) \log f(n))$, $\mathsf{TIME}(f(n))$ is properly contained in $\mathsf{TIME}(g(2n))$.*

Proof: Let S_f be the simulator from Theorem 3.7 which runs in $g(n)$ steps on inputs of length n. Make a "diagonalizer" machine D that runs the simulator and does the opposite, i.e., $D(M)$ accepts iff $S_f(MM)$ rejects. The language A recognized by D is clearly in $\mathsf{TIME}(g(2|M|))$. For any N which runs in $f(|M|)$ time for all inputs M, $N(N) \neq D(N)$ because $S_f(N, N) = N(N)$ by theorem 3.7. Hence, no machine N can decide A in $f(|M|)$ steps on all inputs M.

Alternative ending: Suppose that $N(M)$ runs in $f(|M|)$ steps and recognizes A. $N(N)$ accepts implies that $S_f(N, N) =$ reject. $N(N)$ rejects implies that $S_f(N, N) =$ accept. Contradiction. ∎

Corollary 3.10 *$\mathsf{TIME}(n^k)$ is properly contained in $\mathsf{TIME}(n^{k+1})$, for all k.*

[1] If it is not, what M does might be meaningless, but it does not hurt the argument.

Theorem 3.11 (Space Hierarchy) *For any proper complexity function $f(n)$ and $g(n) = \omega(f(n))$, $\mathsf{SPACE}(f(n))$ is properly contained in $\mathsf{SPACE}(g(2n))$.*

The proof is left as an exercise.

4 Worthy of Mention

I will state two theorems that are worth mentioning. The first shows that our restriction to proper complexity functions is not a mere technicality; without this restriction very strange gaps appear in the complexity classes. The second warns us that there is not always an optimal time bound for a given problem.

Theorem 4.1 (Trakhtenbrot-Borodin[2] Gap Theorem) *Given any total recursive function[3] $g(n) \geq n$, there exists a total recursive function $S(n)$ such that*

$$\mathsf{SPACE}(S(n)) = \mathsf{SPACE}(g(S(n)).$$

The gap theorem has a similar version for time.

Theorem 4.2 (Blum's Speedup Theorem) *Let $r(n)$ be any total recursive function. There exists a recursive language L such that for any TM M recognizing L, there is a faster TM M' recognizing L. If M runs in time $t(n)$, M' will run in time less than $r(t(n))$ for all but finitely many inputs.*

The proofs of these theorems can be found in *Introduction to Automata Theory, Languages, and Computation*, by Hopcroft and Ullman (published by Addison-Wesley).

References

[1] F. C. Hennie and R. E. Stearns. Two-tape simulation of multitape machines. *J. ACM.*, 13:4, pp 533-546, 1966.

[2] A. M. Turing. On computable numbers, with an application to the *Entscheidungsproblem. Proc. London Math. Society*, 2, 42, pp. 230-265, 1936.

[2] Trakhtenbrot published his proof in 1964, Borodin, unaware of his work, published his proof in 1972.

[3] That means any function computable by a TM and defined for every value.

CS 15-855 Computational Complexity Theory **September 15, 2000**

Lecture 2: B.C. Complexity Theory— Palindromes

Lecturer: Rudich *Scribe: Patrick Riley / Editor: Håkan Younes*

Synopsis: The lecture first develops some results from the area of communication complexity, specifically about the equality function. Those results are used to prove very tight bounds on the computational complexity of recognizing palindromes with a multi-tape Turing machine and a one tape Turing machine. The three primary results in this lecture are: a time-space tradeoff for a multi-tape machine (time $\times$ space $= \Omega(n^2)$), a lower bound for the space needed by a multi-tape machine ($\log n$), and a lower bound on the time needed by a one-tape machine (n^2).

5 The Palindromes Problem

This lecture gives a taste of complexity theory Before Cook (before reductions and completeness were the focus of an energetic young complexity theorist's labors). In particular, we will focus on machines that *recognize* the PALINDROMES language:

$$\text{PALINDROMES} = \{XX^R \mid X \in \Sigma^*\} \quad \text{(where } X^R \text{ represents the reverse of } X\text{)}$$

In the previous lecture we saw that (for the multi-tape TM we use) PALINDROMES $\in$ $\mathsf{TIME}(3n)$ and PALINDROMES $\in$ $\mathsf{SPACE}(3 \log n)$.

This lecture will answer the following three questions:

1. Is there a space-time tradeoff for palindromes, or can a single algorithm be simultaneously fast and space efficient?
2. How fast can a one-tape Turing Machine recognize PALINDROMES?
3. Can PALINDROMES be recognized in $o(\log n)$ space?

We can look at item 1 intuitively

Intuition: Consider the input string with substrings X on the right and Y on the left, both having length $n/3$ and separated by a sequence of $n/3$ zeros (fig. 3).

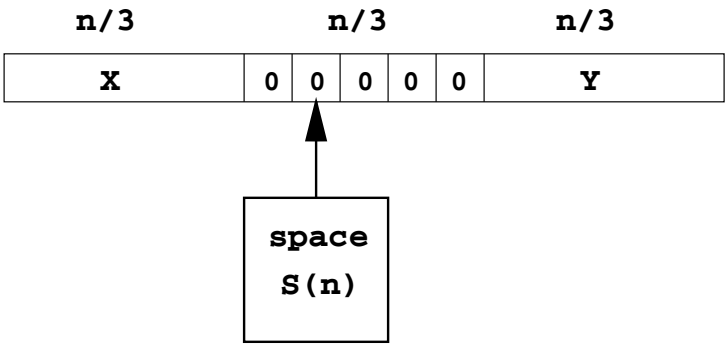


Figure 3: Carrying information across the sea of zeros ...

Suppose that a TM uses at most $S(n)$ space. Whenever it moves from X to Y or back, it can only carry $O(S(n))$ bits of information across the 0's. Thus it needs to make at least $\Omega(n/S(n))$ trips across the section of zeros. Each such trip takes at least $n/3$ steps. Therefore, the total number of steps is at least $T(n) = \Omega(n^2/S(n))$. From this we get the space-time trade-off

$$T(n) \cdot S(n) = \Omega(n^2).$$

Of course, all this is just an intuitive explanation of what yet has to be formally demonstrated.

6 Communication Complexity

In order to prove some interesting things about PALINDROMES, we are going to use some tools from communication complexity. Naturally, we first need to understand a little bit of this theory.

6.1 Basic Definitions

The basic setup is as follows. Two parties have a number that only they know. Call these numbers X and Y (and the respective parties the X-player and Y-player).

The players want to compute a function $f(X, Y) \to \{0, 1\}$ so that they both know the value. The trouble is that communication is very expensive, so they want to minimize the number of bits they have to communicate. We will allow each player to have infinite computational resources.

Example: Consider the function

$$f(x, y) = (x + y) \mod 2 \qquad x, y \in [0 \ldots n]$$

Dumb protocol: Player X sends $\log n$ bits of x to player Y. Player Y sends back one bit of result. The cost is $(\log n) + 1$ bits to communicate.

Smart protocol: Player X sends $x \mod 2$. Player Y sends back the answer. The cose is 2 bits to communicate.

Definition 6.1 *A **protocol** is a binary tree where each internal node v is labeled $\langle X\text{-player}, f_v \rangle$ or $\langle Y\text{-player}, g_v \rangle$ where $f_v \colon X \to \{0, 1\}$ and $g_v \colon Y \to \{0, 1\}$. Each leaf node is labeled 0 or 1.*

The label on the node determines whose turn it is to speak. The functions f_v and g_v determine what bit the player should send, and which branch of the tree to go down. Here is the formal algorithm for when the X-player has input x and the Y-player has input y.

```
current_node = root;
while (current_node is not a leaf)
    if (current_node is X-player)
        X-player computes b = f_v(x)
    else
        Y-player computes b = g_v(y)
    The player announces b
    if (b=0)
        current_node = left_child(current_node)
    else
        current_node = right_child(current_node)
f(x,y) = contents of leaf current_node
```

There are a couple of points to note:

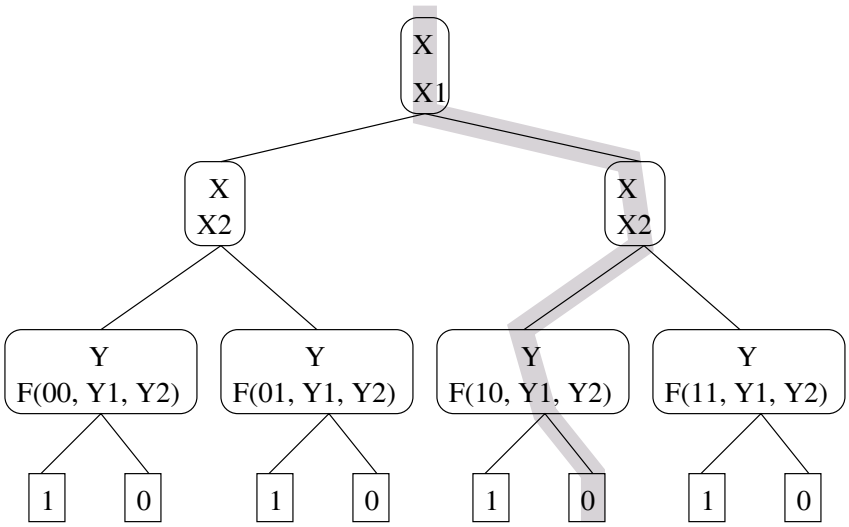


Figure 4: An Example Protocol

- We assume that communication is completely reliable.
- There must be a different protocol for every set X and Y.
- A *path* in the protocol represents a conversation between X and Y

As an example, consider the protocol shown in Figure 4 for computing whether the two bit strings $x = x_1x_2$ and $y = y_1y_2$ are the same. The function F is defined as follows

$$F(a,b) = \begin{cases} 0 & \text{if } a = b \\ 1 & \text{if } a \neq b \end{cases}$$

Note the order of the 0 and 1 leaves!

In particular if $x = 10$ and $y = 11$, the protocol would follow the path highlighted.

Definition 6.2 *The **cost of protocol** P **on input** $\langle \boldsymbol{x,y} \rangle$, denoted $\mathsf{COST}_{\langle x,y \rangle} P$ is the length of the path in P taken on input $\langle x, y \rangle$.*

Definition 6.3 *The **cost of protocol** P, denoted $\mathsf{COST}P$ is the length of the longest path in P.*

Note that

$$\mathsf{COST}P = \max_{\langle x,y\rangle} \mathsf{COST}_{\langle x,y\rangle}P$$

Definition 6.4 *The* **COMMUNICATION_COMPLEXITY**(f) *is the minimum over all protocols P computing f of COSTP.*

Note in particular that if a function f takes inputs of maximum size x and y, COMMUNICATION_COMPLEXITY$(f) \leq \max(x, y) + 1$. Any function f can be computed by one player sending all of their bits to the other, who then computes f and sends the result back.

Definition 6.5 *The* **AVERAGE_COMPLEXITY**(f) *of a set S is the minimum over all protocols computing f (over the set S) of the average cost of computing f. In other words*

$$\mathit{AVERAGE\_COMPLEXITY}(f) = \min_{P \text{ computing } f} \frac{\sum_{\langle x,y\rangle \in S} \mathit{COST}_{\langle x,y\rangle}P}{|S|}$$

An important note here is that while we only look at the cost of P on S, P must be a protocol computing f on all inputs, note just those in S.

6.2 Path Braiding Lemma

Fix a protocol P. Let $C_{\langle x,y\rangle}$ be the path (a.k.a. conversation) taken on input $\langle x, y\rangle$.

Lemma 6.1 ***Path Braiding***

$$\left(C_{\langle x,x\rangle} = C_{\langle y,y\rangle}\right) \implies \left(C_{\langle x,x\rangle} = C_{\langle x,y\rangle}\right)$$

Proof: We will proceed by induction on the path $C_{\langle x,x\rangle}$. The base case (of a 0 length path) is trivial. Suppose that $C_{\langle x,x\rangle}$ is the same as $C_{\langle x,y\rangle}$ up to a node v. Consider two cases:

- v belongs to X-player

  The paths $C_{\langle x,x\rangle}$ and $C_{\langle x,y\rangle}$ must take the same child of v here because the X-player has the same input in both cases.

- v belongs to Y-player

  The paths $C_{\langle y,y\rangle}$ and $C_{\langle x,y\rangle}$ must take the same child of v here because the Y-player has the same input in both cases. We know that $C_{\langle x,x\rangle} = C_{\langle y,y\rangle}$

■

Note that while the statement of the lemma just mentions $\langle x, y\rangle$, by simple variable renaming we have

$$\left(C_{\langle x,x\rangle} = C_{\langle y,y\rangle}\right) \implies \left(C_{\langle x,x\rangle} = C_{\langle x,y\rangle} = C_{\langle y,x\rangle} = C_{\langle y,y\rangle}\right)$$

6.3 The Equality Function

We will now use the Path Braiding Lemma to prove interesting (but perhaps not surprising) things about the equality function. Intuitively, the equality function should have high complexity because (in the worst case), *every* bit of both players input must be compared.

In particular, we will compute the function $f\colon \{0,1\}^n \to \{0,1\}$ with

$$f(x,y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$$

Throughout the rest of this section, let P be any protocol for f.

Lemma 6.2 *For every x, $C_{\langle x,x\rangle}$ in P must be distinct.*

Proof: Assume not; that is $\exists x, y$ with $x \neq y$, such that $C_{\langle x,x\rangle} = C_{\langle y,y\rangle}$. By the Path Braiding Lemma (Lemma 6.1), $C_{\langle x,x\rangle} = C_{\langle x,y\rangle}$. However, $f(x,x) \neq f(x,y)$, so P must give the wrong answer on one of those inputs. $\Rightarrow\Leftarrow$ ■

Theorem 6.3 $\mathit{COMMUNICATION\_COMPLEXITY}(f) = n + 1$

Proof: There are 2^n distinct paths of the form $C_{\langle x,x\rangle}$ (see Lemma 6.2). Clearly, at least one of them must have length n. However, if they all have length n, then

there are only 2^n leaves (which must then be all labeled 1). We know that f must sometimes return 0, so there must be a path a length $n+1$.

We know that COMMUNICATION_COMPLEXITY$(f) = n+1$ because of the trivial algorithm of X sending all her bits to Y, who then sends back the answer. ∎

Theorem 6.4 *AVERAGE_COMPLEXITY*$(f)_{\{\langle x,x\rangle | x\in\{0,1\}^n\}} > n - \log n - c$ *where* c *is a constant.*

Proof: Let us consider how many strings (out of the 2^n possibilities) have cost less than $n - \log n$. By looking at the size of a complete binary tree, we see that there can be at most $2^{n-\log n+1} = \frac{2^{n+1}}{n}$ such strings. Therefore, at least a $\frac{n-2}{n}$ fraction of the strings contribute at least $n - \log n$ to the average. Let's conservatively say that all of those strings contribute $n - \log n$ to the average and that the rest contribute 0.

$$\text{AVERAGE\_COMPLEXITY}(f)_{\{\langle x,x\rangle | x\in\{0,1\}^n\}} > \frac{n-2}{n}(n - \log n) =$$
$$n - \frac{n-2}{n}(\log n) - 2 > n - \log n - 2$$

∎

In the proofs below, we're going to use a variant of the equality function:

$$f'(x,y) = \begin{cases} 1 & \text{if } x = y^R \\ 0 & \text{if } x \neq y^R \end{cases}$$

It should be obvious that this function has the same communication complexity bounds shown above.

7 Time-Space Tradeoffs for Palindromes

We will now use all this communication complexity to prove some results about computational complexity in our model. We will look at the time it takes to recognize palindromes that have a "large" number of 0's in the middle of the string.

Theorem 7.1 *Let* M *be a multi-tape Turing machine running in time* $T(n)$ *and space* $S(n)$ *on inputs of length* $m = 3n$*; rejecting* $\{x0^n y \mid |x| = |y| = n, x \neq y^R\}$ *and accepting* $\{x0^n y \mid |x| = |y| = n, x = y^R\}$*. Then,*

$$\textit{COMMUNICATION\_COMPLEXITY}(f') = \mathrm{O}\left(\frac{T(n)S(n)}{n}\right)$$

Proof: We are going to construct a protocol for computing f' from the machine M. Imagine the two players X and Y interactively simulating the machine M. X simulates M until the read head gets to the Y portion of the string (past the 0s in the middle). At that point, X communicates to Y the current state and the contents of the work tapes, $\mathrm{O}(S(n))$ bits. The Y player then simulates M until the read head crosses into the X portion of the input tape, at which point Y communicates the current state and the contents of the work tape to X.

This continues until the machine halts, at which time whichever player has control of the machine sends the 1 bit answer to the other player.

The only communication occurs when the simulation is handed between players. Each hand off must take at least n time steps because the "sea" of 0s in the middle of the input must be crossed. Therefore, there are at most $\mathrm{O}(T(n)/n)$ communications of $O(S(n))$. Therefore,

$$\mathsf{COMMUNICATION\_COMPLEXITY}(f') = \mathrm{O}\left(\frac{T(n)S(n)}{n}\right)$$

■

Corollary 7.2 $T(n) \cdot S(n) = \Omega(n^2)$

Proof: From Theorem 6.3, we know that $\mathsf{COMMUNICATION\_COMPLEXITY}(f') = n + 1$. Therefore,

$$n + 1 = \mathrm{O}\left(\frac{T(n)S(n)}{n}\right)$$
$$T(n) \cdot S(n) = \Omega(n^2)$$

■

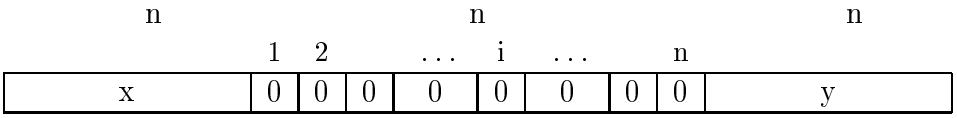
8 One-Tape TM for Palindromes

Now we are going to try and apply the communication complexity results to the one tape Turing machine model. This is a little more challenging because the machine is allowed to write over symbols on the input.

The proof this time will work by defining a set of protocols and using our bounds on average communication complexity (Theorem 6.4) to get a bound on the time for the Turing machine.

Note that I have switched the usage of n and m here from the lecture slides to be consistent with the previous section. In general it doesn't matter because they are only a constant factor different.

Consider a one-tape TM M which recognizes palindromes in time $T(m)$. We will consider inputs of the form $\{x0^n y \mid |x| = |y| = n\}$, where $m = 3n$. Pictorially



Let Δ_i be the ith cell from the left in the 0-portion of the string.

Now we will define communication protocol P_i. This is similar to the interactive simulation used in the proof of Theorem 7.1. The X player simulates M while the read head is to the left of Δ_i. The Y player simulates M at Δ_i and to the right of it. When they hand off the simulation, they communicate $\mathrm{O}(1)$ bits (just the current state of the machine).

Lemma 8.1 *For all inputs x,*

$$\sum_{i=1}^{n} \mathit{COST}_{\langle x,x\rangle} P_i = \mathrm{O}(T(m))$$

Proof: We'll prove this by looking at the path of the read head during the computation of the machine. $\mathsf{COST}_{\langle x,x\rangle} P_i$ is a lower bound on the amount of time the read head spends pointing to the symbol at Δ_i (up to a multiplicative constant). $\mathsf{COST}_{\langle x,x\rangle} P_i$ does not count cases where the read head comes from the right and then returns to the right. The sum of the costs is therefore a lower bound on the time of the machine. ∎

Theorem 8.2 *For a one-tape Turing machine recognizing* PALINDROMES, $T(n) = \Omega(n^2)$

Proof: By using Lemma 8.1 we get the following:

$$\sum_{x \in X} \sum_{i=1}^{n} \mathsf{COST}_{\langle x,x \rangle} P_i = 2^n \mathrm{O}(T(m))$$

$$\sum_{i=1}^{n} \sum_{x \in X} \mathsf{COST}_{\langle x,x \rangle} P_i = 2^n \mathrm{O}(T(m))$$

If we have an upper bound on the sum of a group of elements, we know that at least one of the elements must be at least as small as the average. Therefore, $\exists i$ such that:

$$\sum_{x \in X} \mathsf{COST}_{\langle x,x \rangle} P_i \leq 2^n \mathrm{O}\left(\frac{T(m)}{n}\right) = 2^n \mathrm{O}\left(\frac{T(n)}{n}\right)$$

$$\therefore \frac{\sum_{x \in X} \mathsf{COST}_{\langle x,x \rangle} P_i}{2^n} \leq \mathrm{O}\left(\frac{T(n)}{n}\right)$$

$$\therefore \mathsf{AVERAGE\_COMPLEXITY}(f') = \mathrm{O}\left(\frac{T(n)}{n}\right)$$

We know from Theorem 6.4 that $\mathsf{AVERAGE\_COMPLEXITY}(f') = \Omega(n)$. Therefore, $T(n) = \Omega(n^2)$. ∎

9 Lower Space Bound for Palindromes

In order to answer our third question, we will use the very useful technique of counting the number of configurations a machine can have.

Theorem 9.1 *A Turing machine M on input x of length n can have at most $n \cdot 2^{kS(n)}$ configurations (where k is a constant).*

Proof: Let consider how many bits are needed to describe a configuration. $\log n$ bits are needed to indicate the read head position. $\mathrm{O}(S(n))$ bits are needed to describe what is on the work tapes. $\mathrm{O}(1)$ bits are needed to describe the state. The number of configurations will be bounded by

$$2^{\log n + \mathrm{O}(S(n))} = n \cdot 2^{\mathrm{O}(S(n))}$$

∎

Corollary 9.2 *A recognizer using $S(n)$ space does not use more that $n \cdot 2^{kS(n)}$ time.*

Theorem 9.3 *No Turing machine recognizes* PALINDROMES *in* $o(\log n)$ *space.*

Proof: Assume there was a machine M that recognizes PALINDROMES in $o(\log n)$ space. By Corollary 9.2,

$$T(n) \leq n2^{ko(\log n)} \leq nn^{\frac{1}{2}} = n^{\frac{3}{2}}$$
$$T(n)S(n) = n^{\frac{3}{2}}o(\log n) \ll n^2$$

The last statement contradicts our space time tradeoff, Theorem 7.1. ∎

10 Robust Complexity Classes

Now it's time to take a step back and understand what we have done. We proved some very good and exact bounds on the computational complexity of recognizing PALINDROMES. However, the results depend on the details of our machine model. In particular, having two tapes is faster than 1. If we had random access to the tape, we could use time n and space $\log n$.

10.1 Some Robust Classes

For the rest of the course, we will focus on complexity results which are:

- Robust: if we make "reasonable" changes to the machine model the result still holds.
- Capable of classifying interesting problems: Most people would consider PALINDROMES a solved problem. We are interested in the difficulty of problems like primality, graph isomorphism, VLSI layout, linear programming, and factoring.

Here are some classes which seem to be robust:

$$\begin{aligned}
\mathsf{L} &= \mathsf{SPACE}(\log n) \\
\mathsf{P} &= \bigcup_k \mathsf{TIME}\left(n^k\right) \\
\mathsf{PSPACE} &= \bigcup_k \mathsf{SPACE}\left(n^k\right) \\
\mathsf{EXP} &= \bigcup_k \mathsf{TIME}\left(2^{n^k}\right)
\end{aligned}$$

There are a few obvious results:

- $\mathsf{L} \subset \mathsf{P}$

  A $\log n$ space recognizer uses no more that $n2^{k \log n}$ time (Corollary 9.2).

- $\mathsf{P} \subset \mathsf{PSPACE}$

  An n^k time machine is an n^k space machine.

- $\mathsf{PSPACE} \subset \mathsf{EXP}$.

  An n^k space recognizer uses no more than $n2^{\mathrm{O}(n^k)}$ time (Corollary 9.2).

- $\mathsf{L} \neq \mathsf{P} \vee \mathsf{P} \neq \mathsf{PSPACE}$

  We know from the Space Hierarchy Theorem that $\mathsf{L} \neq \mathsf{PSPACE}$.

- $\mathsf{P} \neq \mathsf{PSPACE} \vee \mathsf{PSPACE} \neq \mathsf{EXP}$

  We know from the Time Hierarchy Theorem that $\mathsf{P} \neq \mathsf{EXP}$.

Further, here are some of the best known problems and the complexity classes to which we know they belong:

- L: multiplication, many statistical tests, boolean formula evaluation
- P: linear programming, GCD, Gauss's quadratic residue mod$p$ algorithm
- NP: SAT, factoring
- PSPACE: games

10.2 Invariance Theses

Alonzo Church believed that than any effective mechanistic procedure can be simulated on a Turing machine. This gives rise to several "Invariance Theses"

- Time: Any reasonable machine can be simulated by a Turing machine with only a polynomial slow down.
- Space (for $\geq \log n$): Any reasonable machine can be simulated by a Turing machine using the same space.
- Time and Space: Any reasonable machine can be simulated by a Turing machine using the same space and only a polynomial slow down.

Peter Shor (at FOCS 94) gave algorithms for a "Quantum Computer" which can not (obviously) be efficiently simulated by a Turing machine! This is certainly a danger to the invariance thesis, but a quantum computer is not yet a reasonable model of computation. The invariance theses are scientifically falsifiable hypotheses. When a new machine model comes around, the hierarchy of complexity classes will have to be shuffled around.

If the invariance theses are true, then any space class larger than $\log n$ is robust and any time class with bounds closed under multiplication is robust. In particular P is the smallest *robust* time class containing $\mathsf{TIME}(n)$.

10.3 Lower Bounds

In the lower bounds presented here, we exploited a weakness in the machine model, namely that it is difficult to get information from one side of the input tape to the other. If we wanted to prove some lower bounds for *robust* classes, we have to exploit a weakness in *all* the machine. Our intuition begins to fail us as we look for weaknesses in all machines. That's one of the things that seems to make complexity theory such a challenging area.

CS 15-855 Computational Complexity Theory **September 18, 2000**

Lecture 3: Non-determinism, Completeness, Central ?s

Lecturer: Steven Rudich *Scribe: Håkan Younes / Editor: Dominic Mazzoni*

Synopsis: Non-determinism. Definition of non-deterministic Turing machines. Definition of non-deterministic time and space. NP, NL, and co-classes. "P = NP?" and other central questions. Reducibility: Turing reducible, Karp reducible, logspace reducible. Completeness and hardness. The Cook-Levin theorem (SAT is NP-complete).

11 Non-determinism

So far we have dealt with deterministic processes. We will now see what happens if we introduce non-determinism into our models.

11.1 Non-deterministic Turing Machines

Recall from the first lecture that a (deterministic) Turing machine M, formally defined, is a 3-tuple (Σ, Q, δ) where Σ is a finite alphabet of tape symbols, Q is the control states, and δ is the transition function for M. The function δ takes a configuration C_i of M, and returns the configuration C_{i+1} following C_i.

Instead of having a function, we can allow δ to be a relation. In other words, for a given configuration C_i, δ would return a set of configurations that are allowed to follow C_i. It is not specified which path is taken by the machine, but rather we say that the configuration following C_i is chosen **non-deterministically** from the elements in $\delta(C_i)$. Hence, we name this beast a **non-deterministic Turing machine** (NTM).

A computation path in an NTM is a sequence $C_0, C_1, \ldots, C_k$ such that $C_{i+1} \in \delta(C_i)$. A configuration C_i is reachable if there exists a computation path from the start configuration C_0 to C_i. The non-deterministic computation of an NTM on a given input x is a directed graph with vertices representing the reachable configurations, and with an edge from C_i to C_j whenever $C_j \in \delta(C_i)$.

Definition 11.1 *An NTM M* **accepts input** x **in time** t *if there is a computation path of length less than or equal to t from the start configuration C_0 to an accepting configuration.*

Definition 11.2 *An NTM M* **accepts a set** S *if for every $x \in S$, M accepts x, and for every $x \notin S$, M does not accept x.*

A non-deterministic computation is often thought of as a tree (Figure 5), where each node represents a configuration, and branches indicate non-deterministic choices.

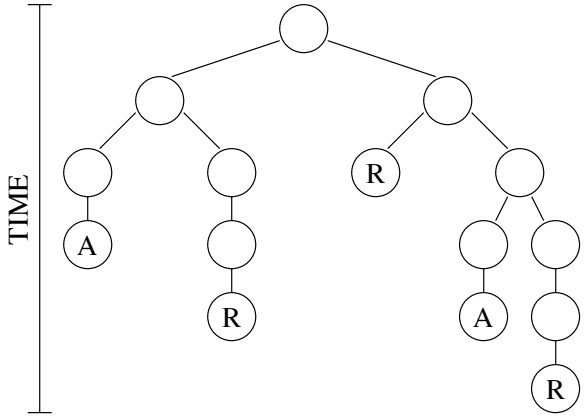


Figure 5: A non-deterministic computation, viewed as a tree.

11.2 Time and Space

We can now define a few more time and space classes. As before, $f(n)$ is a function from $\mathbb{N}$ to $\mathbb{N}$.

Definition 11.3 *$A \in \mathsf{NTIME}(f(n))$ if there is an NTM M accepting A such that on input x, the computation of M contains no path longer than $f(|x|)$.*

Definition 11.4 *$A \in \mathsf{NSPACE}(f(n))$ if there is an NTM M accepting A such that on input x, no configuration in the computation of M uses more than $f(|x|)$ cells of its work tapes.*

We can program an NTM to solve satisfiability of boolean formulas (SAT) as follows:

1. Guess an assignment to the variables.
2. Substitute the guessed assignment into the input formula and evaluate it.
3. If the formula evaluates to true on the guessed assignment, accept; otherwise reject.

If the number of variables in a formula ϕ is n, it requires $O(n)$ steps to guess a value for all the variables. The substitution of the variables with the guessed values, and the subsequent evaluation of the formula can be performed in $O(|\phi|)$ steps. The computation will consist of 2^n paths, all with length $O(|\phi|)$ (really $O(|\phi| + n)$, but the number of variables cannot exceed the length of the formula). Thus, according to our previous definition (and the equivalent of the linear speedup theorem for non-deterministic machines), SAT $\in \mathsf{NTIME}(n)$.

11.2.1 Non-deterministic Polynomial Time (NP)

We now define the complexity class NP as follows:

$$\mathsf{NP} = \bigcup_k \mathsf{NTIME}(n^k)$$

Since we proved SAT is in $\mathsf{NTIME}(n)$, clearly SAT is in NP. Other problems also in NP are, for example, determining whether two graphs are isomorphic, and determining whether a graph is 3-colorable.

We can define a path simulator Turing machine S, which takes as input an NTM M, an input vector x, and a path P, and returns the configuration C that M would reach by following path P. P is specified as a sequence of numbers, with each number bounded by the max fan-out of the relation δ. The path simulator can work with no time overhead, or no space overhead.

Without loss of generality, we can simulate any NTM M as follows:

1. Guess a path P.
2. Run $S(M, x, P)$.

There is only a constant factor loss in time.

Instead of thinking about paths, an alternative way of thinking about NP is that we first guess a solution and then verify it. If A is in NP, and we want to know if

$x \in A$ is true, we can guess a "proof" of the statement, and then verify the proof. The verification must be efficient (it must run in polynomial time). We can therefore think of NP as the class of problems for which a solution can be **verified** in polynomial time, while for problems in P a solution can be **generated** in polynomial time. We will return to this issue later.

By using the path simulator S, we can show that NP $\subset$ PSPACE. Simply loop through all the paths and run S on each path. We are simply doing depth first search on the computation tree of the NTM. This requires space proportional to the length of the longest path, but this is the running time of the NTM which is polynomial in the size of the input. Thus, the depth first search can be done in PSPACE.

11.3 Non-deterministic Logarithmic Space (NL)

Non-deterministic logarithmic space NL is defined as NSPACE($\log n$). A set is in NL if it can be recognized by an NTM using only $O(\log n)$ work tape cells.

Any directed graph $G = (V, E)$ can be represented as a list of node pairs, with each pair (v_1, v_2) representing a directed edge from v_1 to v_2. STCONN is the set of all directed graphs in which there is a directed path from node s to node t. We can recognize STCONN using the following algorithm:

Read input to count the number of edges, and let n be this number
$current\_node \leftarrow s$
$counter \leftarrow 0$
repeat until ($counter = n$ OR $current\_node = t$)
    Read an edge (v_1, v_2) from the input
    if $v_1 = current_node$ **then**
        choose non-deterministically
            $current\_node \leftarrow v_2$
            $counter \leftarrow counter + 1$
        or
            do nothing
if $current\_node = t$ **then**
  ACCEPT
else
  REJECT

Note that in the algorithm above, we implicitly assume that the NTM skips back to the beginning of its input if it gets to the end and hasn't exited the loop yet.

Anyway, an NTM can be programmed to run the algorithm. It is clear that it only uses logarithmic space, since all we need to keep on the work tapes at each time step is a counter and a node name. This means that STCONN is in NL.

A computation for a non-deterministic NTM, using only $O(\log n)$ space, is a directed graph of size $O(n)$. We can determine if there is a path from the start configuration C_0 to an accepting configuration C_{final} in polynomial time. Thus, we have $\mathsf{NL} \subset \mathsf{P}$.

11.4 Co-classes

If we can recognize the complement $\overline{A}$ of a set A in $\mathsf{NTIME}(f(n))$, then we say that A is in $\mathsf{coNTIME}(f(n))$. In particular, if $\overline{A}$ is in NP (NL), then A is in coNP (coNL). The set of satisfiable logical formulas (SAT) is in NP, so we have that the set of unsatisfiable logical formulas (UNSAT) is in coNP. Another member of coNP is the set of prime numbers (PRIMES). A member of coNL is of course the set of graphs such that there is no directed path from a given node s to another node t. Is $\mathsf{NP} = \mathsf{coNP}$, or $\mathsf{NL} = \mathsf{coNL}$? The former is still an open question, while the answer to the latter will be given in a later lecture.

We can also ask if there are any sets in known to be in $\mathsf{NP} \cap \mathsf{coNP}$. As we mentioned above, PRIMES is in coNP. In homework 2, we showed that PRIMES is also in NP, so there we have one member of $\mathsf{NP} \cap \mathsf{coNP}$.

12 Central Questions

Clearly, all sets recognizable in deterministic polynomial time are recognizable in non-deterministic polynomial time. Consequently, $\mathsf{P} \subset \mathsf{NP}$. We might then ask ourselves if there are more sets in NP than in P. Does the non-determinism buy us anything substantial, or can an NTM be simulated by a deterministic TM with only polynomial slowdown? This is the question "$\mathsf{P} = \mathsf{NP}$?" that has haunted theoreticians for several decades (at least since 1971, but Kurt Gödel had already posed the question in 1956 in a letter to von Neumann). Although the problem is still open, there are strong reasons to believe that the answer is "No!".

Table 1: Examples of recognition tasks and generation tasks.

Recognition	Generation
Audience	Composer
Appreciating jokes	Being a comedian
Verifying that p and q are factors of pq	Factoring pq
Understanding a mathematical argument	Being a research mathematician

12.1 Recognition vs. Generation

We mentioned above that P can be seen as the complexity class with members that can be generated in polynomial time, while NP contains sets for which membership can be verified (recognized) in polynomial time. If $\mathsf{P} = \mathsf{NP}$, then generation would be just as easy as recognition. Table 12.1 gives examples of recognition tasks and generation tasks.

Remember that SAT is in NP. It is easy to verify if a given variable assignment satisfies a formula ϕ, but it seems harder to come up with a satisfying assignment only given ϕ. If $\mathsf{P} = \mathsf{NP}$, we would be able to find a satisfying assignment in polynomial time. In general, given a relation $R(x, y)$ computable in polynomial time, and given an x and a y, we can verify these values indeed make $R(x, y)$ true. Suppose now that $\mathsf{P} = \mathsf{NP}$. Then we can generate an x and a y that satisfy $R(x, y)$ (use self reducibility as in homework problem 1.1).

We could let R be the relation between a formula F in first order logic, and a proof that the F is true. With $\mathsf{P} = \mathsf{NP}$, we could generate, not only a proof, but the **shortest** proof for F in time polynomial in the length of the proof. This means you could solve all mathematical problems in time polynomial in the length of the answer! This would of course leave all mathematicians without work, but $\mathsf{P} = \mathsf{NP}$ has far wider consequences. It is, for example, easy to verify if a design for a cold fusion reactor works. Well, then use the same technique as before to generate a design for a cold fusion reactor. But why stop there? A society with access to this wonderful algorithm could produce optimal airplanes, bridges, buildings, rockets, etc. All of this sounds just incredible, and perhaps that is why a majority of theoreticians believe P just cannot be equal to NP.

Table 2: Relations between complexity classes and concepts.

Complexity Classes	Concepts	Example
$\mathsf{P} = \mathsf{NP}$?	Recognition vs. Generation	SAT $\in \mathsf{P}$?
$\mathsf{NP} = \mathsf{coNP}$?	Existential vs. Universal	TAUTOLOGY $\in \mathsf{NP}$?
$\mathsf{P} = \mathsf{L}$?	Sequential vs. Parallel	CVAL $\in \mathsf{L}$?
$\mathsf{L} = \mathsf{NL}$?	Recognition vs. Generation	STCONN $\in \mathsf{L}$?
$\mathsf{P} = \mathsf{NP} \cap \mathsf{coNP}$?	Computation vs. Proof	FACTORING $\in \mathsf{FP}$?
$\mathsf{P} = \mathsf{PSPACE}$?	Time vs. Space	$n \times n$ GO $\in \mathsf{P}$?
$\mathsf{EXP} = \mathsf{NEXP}$?	Recognition vs. Generation	$n \times n$ TILING $\in \mathsf{EXP}$?

12.2 Questions Related to Other Complexity Classes

We have just argued that the question "$\mathsf{P} = \mathsf{NP}$?" can be related to the question "Is there a qualitative difference between the ability to recognize and the ability to generate?". Similar questions can be related to other complexity classes. Table 12.2 summarizes this. Figure 6 shows the world picture we have constructed so far, assuming none of the complexity classes are the same. If the answer is "yes" to any of the questions in Table 12.2, we would have to revise the world picture accordingly.

13 Reducibility

An important concept in complexity theory is **reducibility**. We often want to convert recognition of a set A into recognition of a set B. If we know something about the complexity of recognizing elements in A, this will give us information about the complexity of recognizing elements in B. There are different flavors of reducibility:

Definition 13.1 *A set A is* **Cook (Turing) reducible** *to B ($A \leqslant_T B$) if there exists a polynomial time oracle M such that M^B decides A.*

Definition 13.2 *A set A is* **Karp (Levin, or Many-One) reducible** *to B ($A \leqslant_m B$) if there exists a function $f \in \mathsf{FP}$ such that $x \in A$ if and only if $f(x) \in B$.*

Definition 13.3 *A set A is* **logspace reducible** *to B ($A \leqslant_\ell B$) if there exists a function $f \in \mathsf{FSPACE}(\log n)$ such that $x \in A$ if and only if $f(x) \in B$.*

The reducibility relations are transitive. Furthermore, logspace reducibility is the strongest—i.e. $A \leqslant_\ell B \implies A \leqslant_m B \implies A \leqslant_T B$.

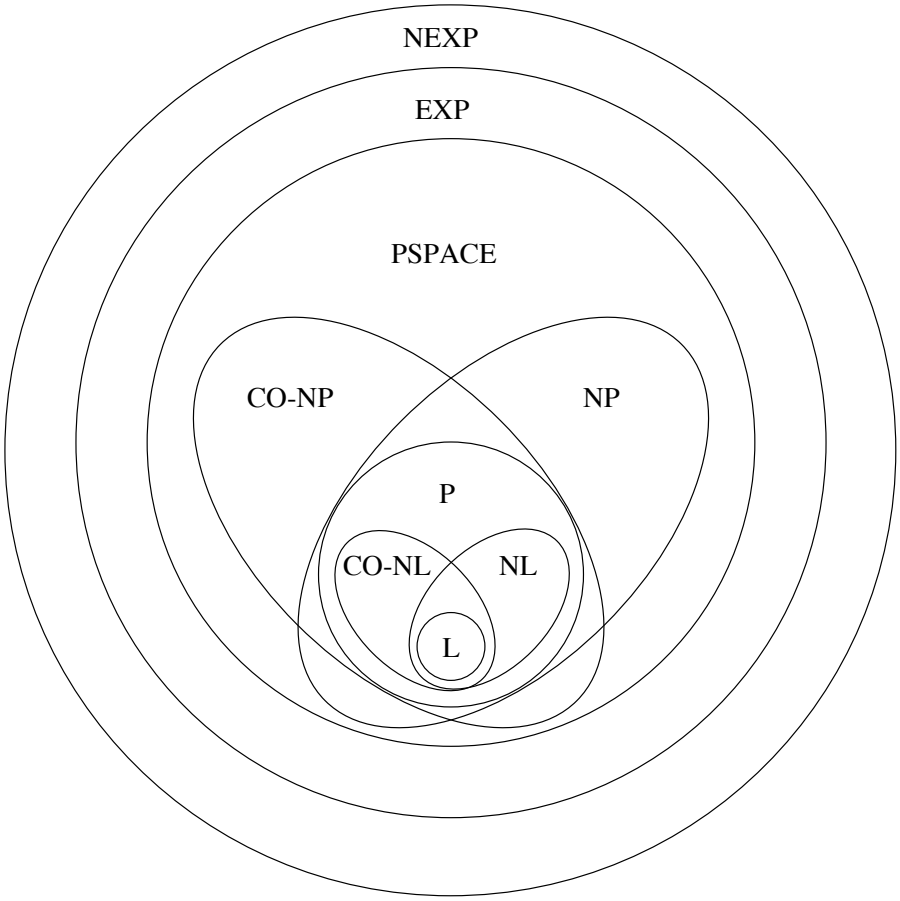


Figure 6: The world picture so far.

Knowing $A \leqslant_T B$ already tells us that if we can solve B, then we can solve A. Why bother making sure the reduction can be performed in logspace? The less resources needed for the reduction, the closer is the relation between A and B. For example, UNSAT $\leqslant_T$ SAT because we can have a Turing machine M that simply negates the result of a machine recognizing SAT. If, however, UNSAT $\leqslant_m$ SAT then NP equals coNP. Whenever possible (which is almost always), logspace reductions are used.

14 Completeness

Definition 14.1 *A set A is* **NP-hard** *with respect to logspace reductions if for every $B \in NP$, $B \leqslant_\ell A$.*

Definition 14.2 *A set A is* **NP-complete** *with respect to logspace reductions if A is NP-hard, and $A \in NP$.*

Completeness is about expressiveness. An NP-complete set can, in some sense, express any other set in NP. If we can find an NP-complete set A, and we can show $A \in \mathsf{P}$, then we know $\mathsf{P} = \mathsf{NP}$. We can substitute any other complexity class in place of NP in the definitions above. Thus we can talk about EXP-hard, or PSPACE-complete sets.

Theorem 14.1 (Cook-Levin) *SAT is NP-complete with respect to logspace reductions.*

Proof: We know from before that SAT is in NP. Now, let A be an arbitrary set in NP. We must show that a logspace f exists such that $x \in A \iff f(x) \in \text{SAT}$.

Let M be an NTM recognizing A running in time n^k. For a given input x, $x \in A$ if and only if there exists an $|x|^k \times |x|^k$ tableau for $M(x)$ such that:

1. $C_0 = q_{start}x$
2. $C_i \stackrel{M}{\to} C_{i+1}$ for every step of the computation.
3. q_{accept} appears in the tableau.

We will build a formula $\phi(\vec{t})$ that will verify a tableau. The input to ϕ is a bit vector $\vec{t}$, with bits $t_{i,j,s}$ that are 1 if and only if the symbol s is written on cell (i,j) of the tableau. The formula $\phi(\vec{t})$ will be a conjunction of four different parts, each verifying a specific aspect of the tableau:

- $\phi_{syntax}(\vec{t})$, which is true if and only if there is exactly one symbol in each cell:

$$\phi_{syntax}(\vec{t}) = \bigwedge_{i,j} \left(\bigvee_{s} t_{i,j,s} \right) \wedge \bigwedge_{i,j,s \neq s'} (\neg t_{i,j,s} \vee \neg t_{i,j,s'})$$

  The first part verifies that every cell has at least one symbol, and the second part that no cell has more than two symbols.

- $\phi_{init}(\vec{t})$, which is true if and only if $q_{start}x$ is written on the first line of the tableau:

$$\phi_{init}(\vec{t}) = t_{1,1,q_{start}} \wedge t_{1,2,x_1} \wedge \ldots \wedge t_{1,n+1,x_n} \wedge t_{1,n+2,\square} \wedge \ldots$$

- $\phi_{local}(\vec{t})$, which is true if and only if all 2×3 regions in the tableau are consistent with the transition relation δ:

$$\phi_{local}(\vec{t}) = \bigwedge_{r \in R} \Delta(r)$$

  Here, R is the set of all 2×3 regions, and Δ is a function that checks consistency for such regions. For example, Δ would be true for the region $\begin{array}{|c|c|c|}\hline c & q & a \\ \hline q' & c & b \\ \hline \end{array}$ if and only if $(b, q' \leftarrow) \in \delta(a, q)$.

- $\phi_{accept}(\vec{t})$, which is true if and only if q_{accept} occurs in the tableau:

$$\phi_{accept}(\vec{t}) = \bigvee_{i,j} t_{i,j,q_{accept}}$$

As defined, $\phi(\vec{t})$ is in conjunctive normal form and has size polynomial in $|x|$. We can construct $\phi(\vec{t})$ in logspace. Furthermore, there is a one to one correspondence between accepting paths in $M(x)$ and satisfying assignments to $\phi(\vec{t})$. Thus, every set $A \in \mathsf{NP}$ can be reduced to SAT. ∎

15 Open Questions

We have introduced non-determinism, and the complexity class NP. We have shown that a number of natural mathematical problems, such SAT and PRIMES, reside in NP. In the previous section, we proved that SAT not only is in NP, but that it is NP-hard. PRIMES, on the other hand, is not. The fact that there exist efficient randomized algorithms for determining primality of a number suggests that PRIMES might be in P. This is possible without having P = NP. Other open questions are if factoring can be done in polynomial time, and if graph isomorphism can be determined in polynomial time. Although many cryptographic schemes would break if factoring could be done in polynomial time, the hierarchy of complexity classes that we have built up so far could still be valid since we do not know that factoring is NP-complete.

CS 15-855 Computational Complexity Theory **September 20, 2000**

Lecture 4: Nondeterminism applied to SPACE

Lecturer: Rudich *Scribe: Maverick Woo / Editor: Luis von Ahn*

Synopsis: In this lecture, we are going to continue building our world picture of complexity classes. In particular, we are going to consider the computation power of different nondeterministic space classes such as NL. We will also prove Savitch's Theorem $\mathsf{NSPACE}(s(n)) \subseteq \mathsf{SPACE}(s^2(n))$ and the Immerman-Szelepscényi Theorem $\mathsf{NSPACE}(s(n)) = \mathsf{coNSPACE}(s(n))$.

16 Recap

Remember that last time we started building our world picture of complexity classes. We are interested to know about, for example, $\mathsf{NP} \stackrel{?}{=} \mathsf{coNP}$ and $\mathsf{P} \stackrel{?}{=} \mathsf{L}$.

We also defined the notion of log-space reducibility $\leq_l$. For two languages $A, B \in \Sigma^*$, we say $A \leq_l B$ iff there exists a log-space reduction function $f\colon A \to B$ such that $x \in A \iff f(x) \in B$. With the notion of reducibility, we were able to give definitions to hardness and completeness. We say A is Δ-hard w.r.t. $\leq_l$ if $\forall B \in \Delta, B \leq_l A$. We say A is Δ-complete if A is Δ-hard and also $A \in \Delta$.

Of the many complete problems in different complexity classes, here are some examples that we will see in this course. CVAL is P-complete as we see in homework 2.1b. The problem of s-t connectivity will be shown to be NL-complete in this lecture. As we all know, SAT is NP-complete. Finally we will show QSAT is PSPACE-complete.

17 NL

17.1 In the beginning, there is s-t connectivity ...

The problem of s-t connectivity (STCONN) can be stated as follows: given a directed graph $G = \langle V, E \rangle$ and two nodes $s, t \in V$, is there a path going from s to t? It

turns out that STCONN is a very important problem because, as we will see, it's NL-complete in a very natural way.

In 1970, Savitch showed that STCONN is in $\mathsf{SPACE}(\log^2 n)$, which immediately implies that PSPACE= NPSPACE. This shows that the effect of using nondeterminism in space classes is not as dramatic as what we believe for time classes as in the $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ question.

A real surprise came in the late 1980's when Immerman (1988) and Szelepscényi (1987), each working independently, discovered that s-t **non**-connectivity is also in NL, thus NL= coNL. That concludes that nondeterminism and co-nondeterminism are equally powerful in space classes. On the other hand, the relationship between nondeterminism and co-nondeterminism for time classes remains open as in the $\mathsf{NP} \stackrel{?}{=} \mathsf{coNP}$ question.

17.2 s-t connectivity is NL-complete

Theorem 17.1 STCONN $\in$ *NL*

The intuition behind this proof is to do a "drunk-pub-crawling"—we first get drunk, and then start walking from pub s on the street. If we ever arrive at pub t before we get rolled over by a car driven by another drunk fellow, then we know there is an s-t path ...

Proof: First read the input graph $G = \langle V, E \rangle$ and nodes s and t. Let $n = |E|$. We start with node s and repeatedly guess the next node among the reachable nodes from the current node and update the current node. We will continue doing this until either we arrive at node t, or we have already guessed n times. If we arrive at node t, then we accept. Otherwise reject.

Observe that the counter can be kept with a $\log n$-bit counter and we can find out the reachable nodes of a given node by simply reading through E once each time. Thus, STCONN $\in$ NL. ∎

Theorem 17.2 STCONN *is NL-complete*

Proof: Since we already have theorem 17.1, all we need to prove is that for any $A \in \mathsf{NL}$, $A \leq_l$ STCONN.

For any $A \in \mathsf{NL}$, let M_A be the NL machine for A using $S(n) = \mathrm{O}(\log n)$ space. WLOG, we assume M_A hs exaclty one accepting configuration C_{accept}.

Let's count the number of possible configurations of M_A. Let the number of states that M_A has be Q, which is a constant. Then there are n input head positions and $2^{\mathrm{O}(\log n)} = \mathrm{O}(n)$ work-tape contents. So there are only $Q \times n \times \mathrm{O}(n) = \mathrm{O}(n^2)$ configurations.

Observe that we only need a $\mathrm{O}(\log n)$ space counter in base-4 to count up to $\mathrm{O}(n^2)$. So here is an algorithm that can generate the configuration graph of M_A:

MAKE-CONFIGURATION-GRAPH(M_A, x)
1 $n \leftarrow length[x]$
2 $V \leftarrow \emptyset$
3 $E \leftarrow \emptyset$
4 **for** $i \leftarrow 1$ **to** n^2
5 **do** $V \leftarrow V \cup C_i$
6 **for** $j \leftarrow 1$ **to** n^2
7 **do** $V \leftarrow V \cup C_j$
8 **if** $C_i \rightarrow_1 C_j$
9 **then** $E \leftarrow E \cup \langle C_i, C_j \rangle$
10 $s \leftarrow C_0$
11 $t \leftarrow C_{accept}$
12 **return** $\langle \langle V, E \rangle, s, t \rangle$

The correctness of this algorithm follows immediate by observing that we have enumerated all possible pairs of configuration. Thus, if $M_A(x)$ accepts, there must be a path from s to t, which means $x \in A \iff$ MAKE-CONFIGURATION-GRAPH$(M_A, x) \in$ STCONN. ∎

The technique of producing the entire computaion graph of M, using little space, generalizes to arbitrary M.

Lemma 17.3 *Let M be a nondeterministic machine using space $f(n) \geq \log n$. There is a deterministic space $f(n)$ machine H that given x outputs the entire configuration graph of $M(x)$.*

17.3 Savitch's Theorem

Theorem 17.4 STCONN $\in$ *SPACE*$(\log^2 n)$

Before we prove this theorem, we will first define a very useful notion that we will be using again in the course. We say, with respect to a graph G, that the predicate PATH(x, y, i) holds if there is a path from x to y in G of length at most 2^i.

Proof: Obviously, if we are able to compute PATH$(s, t, \lceil \log n \rceil)$, then we can decide STCONN. The trick is to re-use space smartly. Consider the following algorithm (with G declared implicitly):

```
PATH(x, y, i)
 1  if i = 0
 2     then if ⟨x, y⟩ ∈ E
 3             then return TRUE
 4             else  return FALSE
 5     else  z ∈ V if PATH(x, z, i − 1) and PATH(z, y, i − 1)
 6                    then return TRUE
 7                    else  return FALSE
```

To see that this algorithm only takes $O(\log^2 n)$ space, we specifically lay out how to implement the recursion stack here: whenever PATH(a, b, l) is called, we place the activation record (a, b, l) on work-tape stack. When the call returns with a value, we remove the triple from the stack and use the returned value to resume work on PATH(a', b', l') where (a', b', l') is now at the top of stack. Observe that PATH$(s, t, \lceil \log n \rceil)$ never has more than $\lceil \log n \rceil$ activation record on the stack, and each activation record is just $O(\log n)$ bits long. Thus, the total space required is $O(\log^2 n)$.

The correctness of this algorithm follows immediate from the fact that if node y is reachalbe from node x in at most 2^i steps, then there must be another node z that is reachable from x in at most 2^{i-1} steps and at the same time can reach y in at most 2^{i-1} steps from z. ∎

17.4 Immerman-Szelepscényi Theorem

Theorem 17.5 ST-NON-CONN $\in$ *NL*

The technique that we will use in this proof may be called "iterate count accounting". First we define $R(i)$ to be the number of nodes reachable from node s in at most i steps, then we show how to compute $R(i)$ from $R(i-1)$. Note that computing $R(i)$ only depends on $R(i-1)$ and nothing before, thus we can compute $R(n)$ iteratively, starting from $R(0)$. This allows us to re-use space efficiently.

Proof: Here is an algorithm that computes $R(i)$ using the value of $R(i-1)$. Note that we iterate on i in the main function and use the value of $R(i-1)$ as an input in the helper function. We adopt the convention that if a function returns NIL that means the function crashes (never returns).

```
COMPUTE-R-HELPER(G, i, R_{i-1})
 1  ⟨V, E⟩ ← G
 2  v_count ← 0
 3  v ∈ V v_flag ← FALSE
 4  u_count ← 0
 5  u ∈ V if guess u is reachable in at most i − 1 steps
 6          then if guess path of at most i − 1 steps from s to u
 7                  then u_count ← u_count + 1
 8                       if u = v or u →_1 v
 9                          then v_flag = TRUE
10                  else return NIL
11  if R_{i-1} ≠ u_count
12     then return NIL
13  if v_flag
14     then v_count ← v_count + 1
15  return v_count
```

```
COMPUTE-R(G, i)
1  ⟨V, E⟩ ← G
2  n ← |V|
3  R_prev ← 1
4  for i ← 1 to n
5  do R ← COMPUTE-R-HELPER(G, i, R_prev)
6     R_prev ← R
7  return R
```

After showing how we can compute $R(n)$, we have a simple algorithm to compute whether node t is unreachable:

T-UNREACHABLILTY(G, s, t)
1 $\langle V, E\rangle \leftarrow G$
2 $n \leftarrow |V|$
3 $V' \leftarrow V \backslash \{t\}$
4 **if** $R_V(n) = R_{V'}(n-1)$
5 **then return** FALSE
6 **else return** TRUE

The correctness of this algorithm can easily be established if we can prove that COMPUTE-R-HELPER indeed computes $R(i)$ from the value of $R(i-1)$. To analyze COMPUTE-R-HELPER, let's focus on the return value, namely v_{count}. We can see that v_{count} is incremented only when u_{count} contains the value $R(i-1)$ (line 13) and v_{flag} is set (line 15).

We observe that the value of $R(i-1)$ by definition is the number of nodes reachable from s in at most $i-1$ steps. By the guards in lines 7 and 8, u_{count} will only get incremented when we find a path from s to any node u in at most $i-1$ steps. By the virtue of nondeterminism, this will imply that at some computation path u_{count} will indeed contain the value of $R(i-1)$. (The guard at line 13 is to make sure we kill the other "branches". Recall what it means by a nondeterministic TM to compute a function.)

It should be obvious that v_{flag} will only be set when u_{count} is incremented, which means u is reachable from s in at most $i-1$ steps, and when either $u = v$ or v is reachable from u in exactly one step. Either way, that means v is reachable from s in at most i steps. Thus, v_{count} indeed contains the value of $R(i)$.

The space requirement of this algorithm is clearly $O(\log n)$ because all we need to keep is a couple of counters around while each of them are bounded by n. ∎

17.5 Deterministic vs. Nondeterministic Space Classes

By Savitch's Theorem, we know STCONN $\in$ SPACE$(\log^2 n)$ and a straight-forward corollary follows:

Corollary 17.6 *NSPACE*$(\log n) \subseteq$ *SPACE*$(\log^2 n)$

Furthermore, we have a similar corollary for any function $f(n)$ greater than $\log n$:

Corollary 17.7 $\mathit{NSPACE}(f(n)) \subseteq \mathit{SPACE}(f^2(n))$

Proof: Let M be a NTM accepting $A \in \mathsf{NSPACE}(f(n))$. There exists a $\mathsf{SPACE}(f(|x|))$ DTM H such that $H(x)$ is the configuration graph of $M(x)$ with C_0 and C_{accept} marked and $|H(x)| = c^{f(|x|)}$ for some constant c. Let G be the $\log^2 n$ space STCONN machine. $G(H(x))$ accepts A in $\log^2(c^{f(|x|)}) = \mathrm{O}(f^2(|x|))$ space. ∎

Also, by the Immerman-Szelepscényi Theorem, we know that ST-NON-CONN $\in \mathsf{NL}$ and a corollary simply follows:

Corollary 17.8 $\mathit{NL} = \mathit{coNL}$

Another corollary whose proof is similar to corollary 17.7 is (also only for functions bigger than $\log n$):

Corollary 17.9 $\mathit{NSPACE}(f(n)) = \mathit{coNSPACE}(f(n))$

18 PSPACE

TQBF (or QSAT) is a variant of SAT. Formally, we define

$$\text{TQBF} := \{\Delta \mid \Delta \text{ is a True Quantified Boolean Formula}\}.$$

A QBF is a formula of the form $\exists x_1 \forall x_2 \exists x_3 \cdots Q_n x_n \, \phi(x_1, x_2, x_3, \ldots, x_n)$ where x_i's are boolean variables, ϕ is a CNF formula on x_i's and Q_n is $\forall$ if n is even and $\exists$ if n is odd.

The main result in this section is that QSAT is actually PSPACE-complete.

18.1 QSat is PSPACE-complete

Theorem 18.1 QSAT $\in \mathit{PSPACE}$

Proof: Given input x, observe that we can easily check if the input is of the right format with a $\log |x|$ bit counter and reject if the format is not right. From now on,

let x be $Q_1x_1Q_2x_2\cdots Q_nx_n\phi(x_1,x_2,\ldots,x_n)$. Here is an algorithm to check if x is true:

CHECK$(Q_1x_1Q_2x_2\cdots Q_nx_n\phi(x_1,x_2,\ldots,x_n))$

```
1  if n = 0
2     then return φ(x_1, x_2, ..., x_n)|_{x_1,x_2,...,x_n}
3     else  if Q_1 = ∃
4              then return CHECK(Q_2x_2 ··· Q_nx_nφ(x_2, x_3, ..., x_n)|_{x_1=0})
5                     or CHECK(Q_2x_2 ··· Q_nx_nφ(x_2, x_3, ..., x_n)|_{x_1=1})
6              else  return CHECK(Q_2x_2 ··· Q_nx_nφ(x_2, x_3, ..., x_n)|_{x_1=0})
7                     and CHECK(Q_2x_2 ··· Q_nx_nφ(x_2, x_3, ..., x_n)|_{x_1=1})
```

Let $S(n)$ be the space used by CHECK$(Q_1x_1Q_2x_2\cdots Q_nx_n\phi)$. We observe that recurrence relation is $S(n) = \mathrm{O}(1) + S(n-1)$ and $S(0) = \log n$. Solving it we get $S(n) = \mathrm{O}(n)$. Thus, CHECK can be done in linear space. ∎

Theorem 18.2 QSAT *is PSPACE-complete*

Before we prove this theorem, here is an observation. Let M be a PSPACE machine. $M(x)$ accepts iff there exists a valid tableau of dimension $|x|^k \times c^{|x|^k}$ which is clearly too big if we have to use a variable for the content of each cell. Thus, the Cook-Levin proof can't be adapted for QSAT.

We will instead try to encode Savitch's space-efficient recursion as a short QBF. In paritcular, we will re-use the PATH(a,b,i) notation again to mean that configuration b is reachable from configuration a in 2^i steps. Clearly, our aim here is to check PATH(C_0, C_{accept}, n^k) for n being the length of the input.

A quick yet flawed attempt would be the encode PATH$(a,b,0) = 1$ iff $a \to_1 b$ and PATH$(a,b,i) = \exists z(\text{PATH}(a,z,i-1) \wedge \text{PATH}(z,b,i-1))$. The problem is our formula's length will get doubled every time i goes down by 1, thus creating an exponential length formula in the end.

Proof sketch: It should be easy to verify that PATH(a,b,i) can be encoded as

$$\exists z \forall x \forall y[((x = a \wedge y = z) \vee (x = z \wedge y = b)) \implies \text{PATH}(x,y,i-1)].$$

Let's check the length of the formula. Denote $|\text{PATH}(a,b,i)|$ as $L(i)$, we see that $L(i) = \mathrm{O}(n) + L(i-1)$ and $L(1) = \mathrm{O}(n)$. Solving this recurrence, we get $L(n) = \mathrm{O}(n^2)$.

Here are a sketch on how to get the resulted formula into QBF: first we can move all the quantifiers into the front of the formula by transforming it into prenex normal form to get $Q_1 x_1 Q_2 x_2 \cdots Q_n x_n[\phi(x_1, x_2, \ldots, x_n)]$. Then we can transform the formaula further by transforming $\phi(x_1, x_2, \ldots, x_n)$ to $\phi'(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m)$ as in SAT $\leq_l$ 3SAT. Finally, we can make sure the quantifiers alternate and start with a $\exists$ by adding dummy variables. ∎

The reader is referred to Papadimitriou page 456 for a very detailed proof on how to massage the formula into the right form and the variable accounting.

18.2 Games

It turns out that there is an interesting way to think of QSAT: for any QBF formula ψ, we can think of it as a game between the $\exists$ player (E) and the $\forall$ player (F). We can see that E wins the game if there exists a choice x_1 for E, such that for all choices x_2 for F, such that there exists a choice x_3 for E, such that ... that $\phi(x_1, x_2, \ldots, x_n)$ is true. Thus, E has a winning strategy iff $\psi \in$ QSAT.

We may define the notion of a "reasonable" game in a PSPACE flavor:

- if n is the nubmer of bits required to describe the state of the game, such as the board configuration, then the maximum number of moves a game could take is bounded by a polynomial in n, assuming at least one of the players is playing optimally.
- when any player gets a winning position, the first player can prove this fact with a proof of length at most polynomial in n.

If we consider any resaonable full-information game, we can use $\exists \vec{y} \phi(\vec{x}, \vec{y})$ to denote that $\vec{x}$ is a win for the first player. Then, $\exists x_1 \forall x_2 \cdots Q_{n^k} x_{n^k} \exists \vec{y} \phi(\vec{x}, \vec{y})$ is in QSAT iff the first player has a win. Using the QSAT oracle, we can say that P^{QSAT} can play any "reasonable" full-information game optimally.

Fact 1 *Nothing less than a PSPACE-complete set could be universal for a two-player, full-information game.*

Here we give an example of a simple PSPACE-complete two-person game called Geography. We assume that there is a n word dictionary accessible to both players and there is a designated starting word. The rules of the game are:

- player one starts and says the designated starting word
- players alternate and must use a word that begins with the last letter of the previous word
- no word can be repeated
- the player who gets stuck loses

An example game using real-life geographic words maybe: Athens, San Francisco, Ohio, Oakland ...

To see that Geography is PSPACE-complete, we observe that it can easily be used to express a QSAT formula by encoding special "words" into the dictionary as the following figure shows:

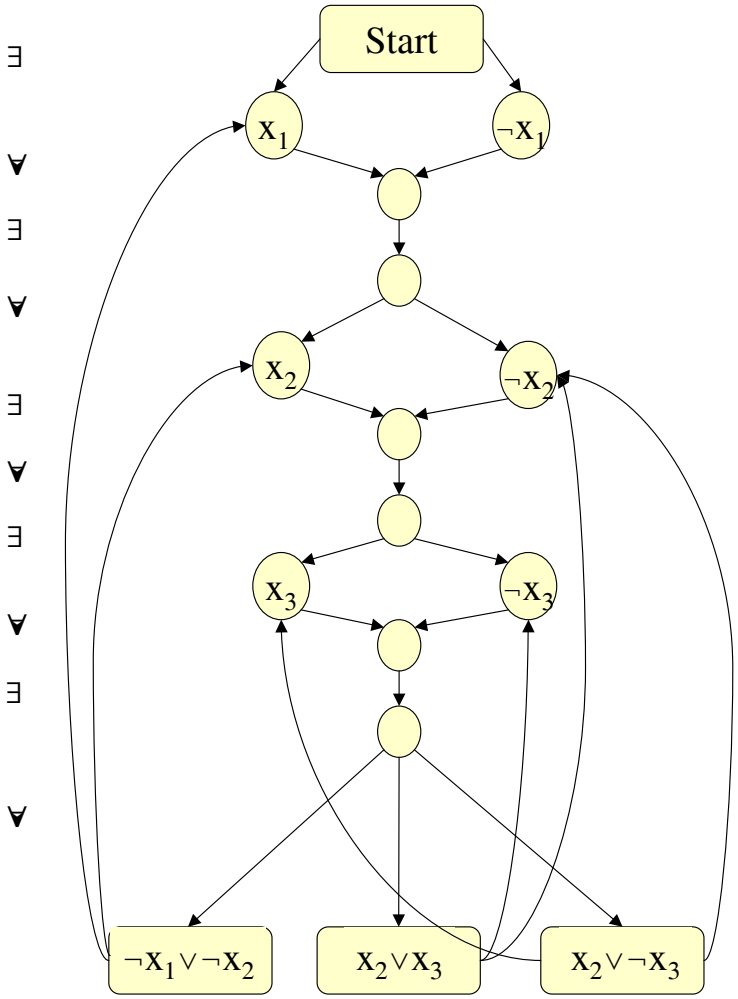


Figure 7: Geography of $\exists x_1 \forall x_2 \exists x_3 [(\neg x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \neg x_3)]$

Furthermore, Geography is not the only real-life game that can be generalized into PSPACE-complete. In fact, the generalization of many real-life games like $n \times n$ Go and $n \times n$ Checkers are actually PSPACE-complete. (For Chess, it's a little bit unnatural to generalize into $n \times n$.) Thus, if SAT is the patron saint of mathematicians, then QSAT is the patron saint of game players. And if NP $\neq$ PSPACE, as what we expect, then it would mean that there is no short way to describe an optimal strategy for every given game. However, we should also note that there are short interactive proofs that has the power of PSPACE, as we will see later in the course.

There are other extensions of games too. For example, solitare games like getting a piece of furniture through a twisting hallway can be PSPACE-complete. (Maverick: Sokoban is a fun puzzle game that is also PSPACE-complete.) Games against random opponents like stochastic SAT can be PSPACE-complete. While PSPACE-complete sounds so good for a lot of games, a three person gams with randomness and hidden information (like some generalized poker game) can be as difficult as NEXP-complete.

CS 15-855 Computational Complexity Theory **September 22, 2000**

Lecture 5: Alternating Quantifiers

Lecturer: Steven Rudich *Scribe: Umut A. Acar / Editor: Bryan Clark*

Synopsis: Oracles, the polynomial hierarchy, polynomially balanced relations, algorithm-oracle paradox, $QSAT$, $PSPACE$.

19 State of Affairs

By now, we have proved that $NSPACE(f(n)) \subset SPACE(f^2(n))$ and $NSPACE(f(n)) = coNSPACE(f(n))$. Whether $L = NL$ is an open and interesting open question. In this lecture, we study $PSPACE$ further by introducing the polynomial hierarchy.

20 Oracles

Much like an algorithm invoking another, generally "simpler", algorithm to solve a problem, we can imagine a Turing Machine invoking another Turing Machine. This worths studying because it enables us to study the problems that can be solved given a solution to some other problem.

Formally, we define a *Turing Machine with an oracle* or an *Oracle Turing Machine (OTM)*, $M^?$ as a multi tape deterministic Turing Machine that has a special tape called *query tape* and three states $q_?$, q_{yes}, and q_{no} that are called the *answer states*. Any language can be plugged in for '?'.

Let A be a language, then a computation of an Oracle Turing Machine with an oracle A proceeds like a Turing Machine, except for the transitions involving the states where the oracle is queried. The Oracle Turing Machine is in $q_?$ state whenever it queries the oracle with a question of the form "is $y \in A$", where y is on the query tape with the head pointing to the beginning of the query. The Oracle Turing Machine then moves to q_{yes} or q_{no} if $y \in A$ or $y \notin A$ respectively. We denote the computation of $M^?$ with oracle A on input x as $M^A(x)$. A nondeterministic Turing Machine with an oracle can be defined similarly.

We define the time complexity of an Oracle Turing Machine as the time complexity of the Turing Machine $M^?$ corresponding to the Oracle Turing Machine. In $M^?$, an oracle query takes one step. We further define complexity classes with oracles based on this notion of time complexity. Let C be a complexity class then C^A is the complexity class of languages that are decided (accepted) by a Turing Machine with oracle A that has the same sort and time bound as in C.

Example:

$$P^A = \{L \mid \textit{There exists an OTM } M^A \textit{ that decides } L \textit{ in polynomial time.}\}$$

Example:

$$NP^A = \{L \mid \textit{There exists an OTM } M^A \textit{ that decides } L \textit{ in nondeterministic polynomial time}\}.$$

We can further apply a complexity class as an oracle. In this case, a language in the oracle complexity class can be used as an oracle throughout a computation.

Example:

$$P^{NP} = \{ L \mid \exists\, A \in NP \textit{ and there is a deterministic, polynomial} - \textit{time Turing Machine with oracle } A \textit{ that decides } L.\}$$

Example:

$$NP^{\Delta} = \{ L \mid \exists\, A \in \Delta \textit{ and there exists a nondeterministic, polynomial} - \textit{time Turing Machine that decides } L.\}$$

Example:$P^{SAT} = P^{NP}$. Indeed, $P^{SAT} \subseteq P^{NP}$, we simply choose, SAT as the oracle language. To show that $P^{NP} \subseteq P^{SAT}$, we observe that every $A \in NP$ can be reduced to SAT in logarithmic space, and thus, in polynomial time.

Here are a few more examples with oracles.

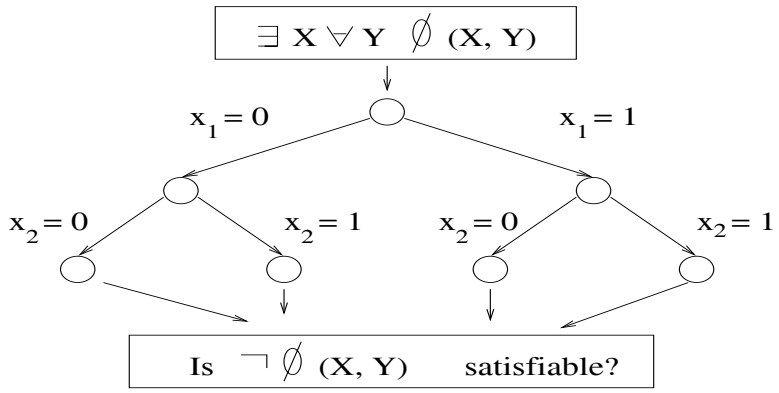


Figure 8: A Turing Machine for finding an X satisfying $\forall Y \Phi(X, Y)$.

Example:Is $P^{QSAT} = NP^{QSAT}$? The answer is yes. First note that $QSAT$ is $PSPACE$ complete. Hence $PSPACE \subseteq P^{QSAT}$ and trivially, $P^{QSAT} \subseteq NP^{QSAT}$. Furthermore, $NP^{QSAT} \subseteq NPSPACE$. Finally, by Savitch's theorem $NPSPACE \subseteq PSPACE$. Thus we have, $PSPACE \subseteq P^{QSAT} \subseteq NP^{QSAT} \subseteq PSPACE$, and hence, $P^{QSAT} = NP^{QSAT} = PSPACE$.

Example:Is $P^{SAT} = NP^{SAT}$? If $P = NP$ then these two classes are equal. Otherwise, we believe that it is highly unlikely that these two a classes are equal. As an example, consider a problem of the form "Is there an x such that $\forall Y \Phi(X, Y)$?" This problem can be solved by an OTM with a SAT oracle , M^{SAT} as shown in Figure 8. The machine M guesses an assignment for X, X_0 and then asks its oracle whether $\neg\Phi(X_0, Y)$ is satisfiable for some Y. If the oracle says "yes" then M does not accept, otherwise, it accepts. As we will see later in the lecture, this problem is unlikely to be in P^{SAT}.

21 The Polynomial Hierarchy

The polynomial hierarchy is the sequence of classes $\Delta_i, \Sigma_i, \Pi_i$ such that, $\Delta_0 = \Sigma_0 = \Pi_0 = P$ and $\Delta_i = P^{\Sigma_{i-1}}$, $\Sigma_i = NP^{\Sigma_{i-1}}$, and $\Delta_i = coNP^{\Sigma i-1}$. We define the *polynomial hierarchy* as the class PH such that $PH = \bigcup_{i \geq 0} \Sigma_i$.

The first level of the hierarchy is the familiar, $P, NP, coNP$ classes. The second level is the P^{NP}, NP^{NP}, and its complement $coNP^{NP}$. There are important problems such as the minimum circuit problem in the second level. Figure 9 depicts the first

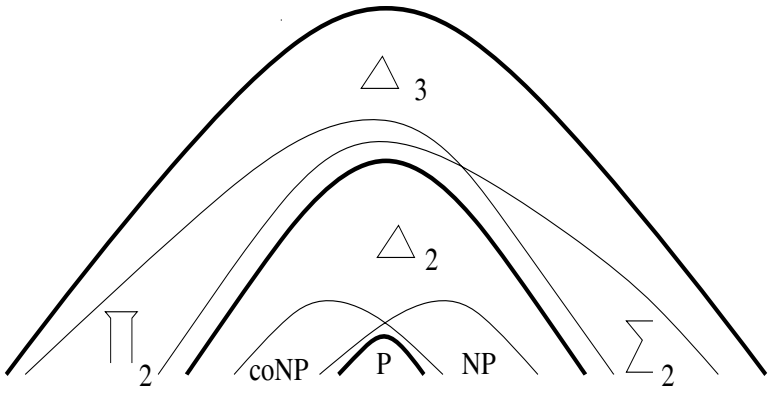


Figure 9: The first three levels of the polynomial hierarchy.

three levels of the polynomial hierarchy.

21.1 Polynomially-balanced Relations

A language $L \in NP$ can be described in terms of a relation R as follows: $L = \{x \mid \exists y\ R(x,y)\}$, where $|y|$ is polynomial in x and R is polynomial-time checkable relation. We call y a certificate for x. Indeed, if a language has this form, then a nondeterministic Turing Machine can guess y and verify it in polynomial time. Similarly, if a language is in NP, then the tableau of the accepting path of the computation on input x is a certificate for x. Similarly, a language L is in $coNP$ if and only if there is a polynomial-time checkable relation R such that $L = \{x \,|\, \forall y\, R(x,y)\}$, where $|y|$ is polynomial in $|x|$. Indeed, for such an L, the complement of L, $L^c = \{x \,|\, \exists y \neg R(x,y)\}$. Since R is polynomial checkable, $\neg R$ is as well. Thus, $L^c \in NP$ and therefore, L is in $coNP$. To prove the sufficiency part, observe that the complement of $L = \{x \mid \exists y\ R(x,y)\}$ have the appropriate form.

A *polynomial-time checkable relation* $R(x, y_1, y_2, ..., y_n)$, where the length of $y_1, ..., y_n$ are polynomially bounded by the length of x is called a *polynomially balanced relation.* Earlier, we expressed NP and $coNP$ in terms of polynomially balanced relations. In the rest of our discussion, all the relations that we mention are polynomially balanced; we will state otherwise. The following theorem demonstrates that every class in polynomial hierarchy can be represented with polynomially balanced relations along with a list of alternating quantifiers.

Theorem 21.1 *A language L is in Σ_i if and only if there is a polynomially balanced*

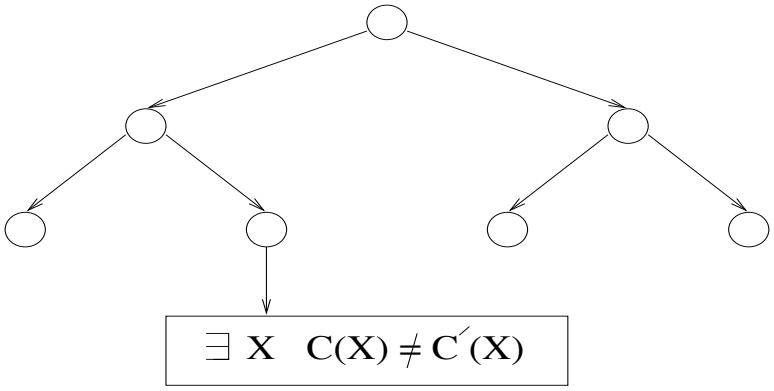


Figure 10: A Turing Machine for the MINIMUM_CIRCUIT problem.

relation R such that

$$L = \{x \mid \exists Y_1 \forall Y_2 \exists Y_3 ... Q_i Y_i \ R(x, Y_1, ..., Y_i)\},$$

where Q_i is a $\forall$ if i is even and an $\exists$ otherwise.

As a corollary observe that a language in $\Pi_i = co\Sigma_i$ can be expressed as

$$L = \{x \mid \forall Y_1 \exists Y_2 \forall Y_3 ... Q_i Y_i \ R(x, Y_1, ..., Y_i)\}.$$

For example the $MINIMUM\_CIRCUIT$ problem which is in Π_2 has the following form:

$$\{c \mid \forall c' \exists x \ c(x) \neq c'(x)\}.$$

The $MINIMUM\_CIRCUIT$ problem can be stated as follows: Given a boolean circuit C, is it true that there is no circuit with smaller gates that gives the same result as C on every input (i.e., is C minimal)? The $MINIMUM\_CIRCUIT$ problem is in Π_2 if one can generate all smaller circuits and verify that each such smaller circuit computes a different function. This can be done by nondeterministically creating all smaller circuits and asking whether $C(X) \neq C'(X)$ for all such circuits. It is currently not known whether $MINIMUM\_CIRCUIT$ is Π_2 complete or not.

Polynomial hierarchy captures the expressibility of a problem in logic. For example, consider the $UNIQUE\_OPTIMAL\_TSP$ problem. The language $UNIQUE\_OPTIMAL\_TSP$ contains all graphs that have a unique optimal tour. The $UNIQUE\_OPTIMAL\_TSP$ language can be expressed as follows:

$$\exists T \forall T' [tour(T) \cap tour(T') \cap (T \neq T') \Rightarrow |T'| > |T|],$$

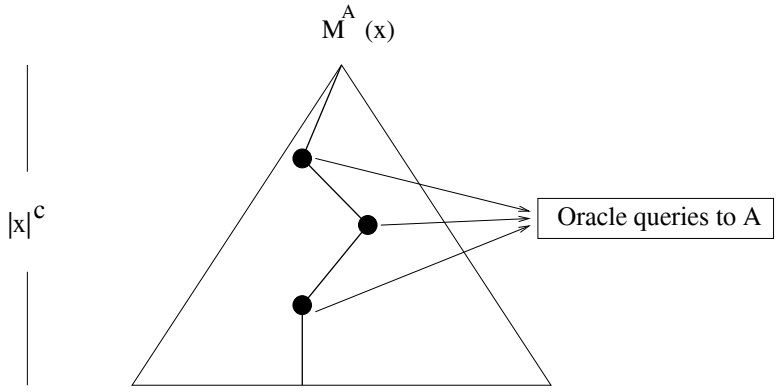


Figure 11: The computation on x with Oracle Turing Machine.

where $|x|$ denoted the length of a tour and $tour(x)$ evaluates to true if and only if x is a tour. This expression of $UNIQUE\_OPTIMAL\_TSP$ tells us that it is in Σ_2. Now, consider an alternative way to express $UNIQUE\_OPTIMAL\_TSP$:

$$\forall T \forall T' \exists T''[tour(T) \cap tour(T') \cap (T \neq T') \Rightarrow (|T''| < |T|) \cup (|T''| < |T'|).$$

In this expression, we used two $\forall$ quantifiers of the same type in a row. Note that, $\forall T \forall T' \equiv \forall (T, T')$.

By now, we have studied decision problems. Decision problems, however, are not a natural way of expressing optimization problems. Optimization problems, when expressed in a more natural way, helps us to understand other important problem classes. As an example, consider the Traveling Salesperson Problem, TSP, and its various characterizations. The decision version of TSP, TSP_D is stated as " Is there a tour of length at most l?" The exact TSP problem, TSP_E is stated as "Is there an optimal tour of length l?" The TSP cost problem, TSP_{COST} is stated as "What is the length of the optimal tour?" And finally, the TSP problem is stated as "What is the optimal tour?" The versions of TSP problem gets harder in the order we stated them, that is, $TSP_D \leq_l TSP_E \leq_l TSP_{COST} \leq_l TSP$. We know that the TSP_D problem is NP-complete. The TSP_E problem is DP-complete - a language is in DP if it is the intersection of two languages, one in NP and the other in $coNP$. The TSP_{COST} and the TSP problems are both FP^{NP}-complete.

The TSP_E problem is in DP because it is the intersection of TSP_D, a problem in NP, and the complement of TSP_D, a problem in $coNP$. Note that DP is not defined as $NP \cap coNP$ but as the intersection of languages rather than classes.

Theorem 21.2 *A language L is Σ_i if and only if there is a polynomially balanced*

relation $S(x, y)$ *such that* $L = \{x \mid \exists y\, S(x, y)\}$, *where* $\{(x, y) \mid S(x, y)\} \in \Pi_{i-2}\}$.

Proof: The proof is by induction on i. For the base case, we need to show that a language $L \in NP$ if and only if $L = \{x \mid \exists y\, R(x, y)\}$ for some polynomially balanced relation R. We have shown this earlier in this section.

For the general case, assume that the statement holds for all integers up to $i-1$. For the sufficiency part, we want to show that L can be recognized by a nondeterministic Turing Machine with a Σ_{i-1} oracle. The nondeterministic Turing Machine simply guesses a y and asks the Σ_{i-1} oracle whether $\neg S(x, y)$; since $S(x, y) \in \Pi_{i-1}$, $S(x, y) \in \Sigma_{i-1}$.

For the sufficiency part, we would like to show that such a polynomially balanced relation $S(x, y)$ exists for a language $L \in \Sigma_i$. Since L is in Σ_i, there is a nondeterministic Turing Machine with oracle $A \in \Sigma_{i-1}$, M^A, that decides L (see Figure 11). The computation with the Turing Machine M^A on an input string $x \in L$ has a polynomial-time accepting path. The Turing Machine M^A makes calls to the oracle A along this path and receives either a "yes" or a "no" answer. Since the language A is in Σ_{i-1}, there is a relation T, such that $T(w, z)$ is true for $w \in A$, i.e., z is a certificate for w. Now, consider the computation tableau for the accepting computation together with the certificate of each query to the oracle A that results in a "yes" answer. We define the tableau of an accepting computation on x together with oracle certificates as the certificate for x. Thus, $S(x, y)$ holds if and only if y is the certificate for x.

Now, we show that $S(x, y)$ is polynomial time checkable. First we need to verify that each move on y is valid, which we do in polynomial time by using the definition of the Turing Machine M^A. Second, we need to verify that each "yes" answer from the oracle is valid. Since, we have a certificate z for each such query for w and $T(w, z) \in \Pi_{i-1}$, this is in Π_{i-1} for each query. There are at most a polynomial number of them and thus, the verification of all "yes"' s is in Π_{i-1}. Likewise, we need to verify for each "no" answer for a query of w that $w \notin A$. Since $A \in \Sigma_{i-1}$, $A^c \in \Pi_{i-1}$ and thus, we ask for whether $w \in A^c$. Since, there are only a polynomial number of "no" answers, all of them can be checked in Π_{i-1}. This completes the proof. ∎

21.2 The Algorithm-Oracle Paradox

Given an algorithm for SAT, we can make a polynomial algorithm for $MINIMUM\_CIRCUIT$. Given an oracle for SAT, however, it is remains unclear how to solve the $MINIMUM\_CIRCUIT$

problem. Many people think that it is not possible. Given these, we have an apparent paradox: An oracle for SAT has the same input-output behavior as an algorithm for SAT, it cannot possibly be less useful in solving problems!

21.3 Quantified SAT

We define PH as the union of the classes in the polynomial hierarchy. Formally, $PH = \sum_{i=0}^{i=\infty} \Sigma_i$. It is now known whether $\Sigma_i = \Sigma_{i+1}$ for some i. However, it is believed that this is not the case. The following theorem exhibits a complete language, $QSAT_i$ for Σ_i. The language $QSAT_i$ is defined as follows

$$QSAT_i = \{X | \exists X_1 \forall X_2 \exists X_3 ... Q_i \; X_i \; \Phi(X_1, X_2, ..., X_i), \; and \; , X = X_1 X_2 ... X_i\},$$

where Φ is a $3 - SAT$ formula and each X_j is a string of bits. Clearly, $QSAT_i \in \Sigma_i$.

Theorem 21.3 *$QSAT_i$ is Σ_i-complete.*

Proof: A language $L \in \Sigma_i$ has the form:

$$\{x \mid \exists Y_1 \forall Y_2 ... Q_i Y_i \;\; R(x, Y_1, Y_2, ..., Y_i)\}.$$

Since, R is polynomially checkable, by Cook-Levin, there are Φ_1 and Φ_2 such that

$$\begin{aligned} R(X, Y_1, ..., Y_2) &\Leftrightarrow \exists Z \; \Phi_1(X, Y_2, ..., Y_i, Z), and \\ R(X, Y_1, ..., Y_2) &\Leftrightarrow \exists Z \; \Phi_2(X, Y_2, ..., Y_i, Z). \end{aligned}$$

If the quantifier Q_i is an $\exists$, then

$$L = \{x \mid \exists Y_1 \forall Y_2 ... \exists Y_i \exists Z \;\; \Phi(x, Y_1, Y_2, ..., Y_i, Z)\},$$

or equivalently

$$L = \{x \mid \exists Y_1 \forall Y_2 ... \exists Y_i' \;\; \Phi(x, Y_1, Y_2, ..., Y_i')\},$$

where $Y_i' = (Y_i, Z)$.

If the quantifier Q_i is a $\forall$, then

$$\begin{aligned} L &= \{x \mid \exists Y_1 \forall Y_2 ... \forall Y_i \ R(x, Y_1, Y_2, ..., Y_i, Z)\} \\ &\equiv \{x \mid \neg(\forall Y_1 \exists Y_2 ... \exists Y_i \ \neg R(x, Y_1, Y_2, ..., Y_i, Z)\} \\ &\equiv \{x \mid \neg(\forall Y_1 \exists Y_2 ... \exists Y_i \exists Z \ \Phi_2(x, Y_1, Y_2, ..., Y_i, Z)\} \\ &\equiv \{x \mid \exists Y_1 \forall Y_2 ... \forall Y_i \forall Z \ \neg\Phi_2(x, Y_1, Y_2, ..., Y_i, Z)\} \\ &\equiv \{x \mid \exists Y_1 \forall Y_2 ... \forall Y_i' \ \neg\Phi_2(x, Y_1, Y_2, ..., Y_i')\}, \end{aligned}$$

where $Y_i' = (Y_i, Z)$. This completes the proof. ∎

The following proposition states that PH is a subset of $PSPACE$. Indeed, a polynomially balanced relation can be checked in polynomial space given a given set of assignment for the variables of the relation. Since, the variables are of polynomial length in the length of the input, this can be done in polynomial space. It is not known whether $PH = PSPACE$. However, if this is the case, then all the polynomial hierarchy collapses to a some layer.

Proposition: $PH \subset PSPACE$.

Remark: PH is the set of all graph theoretic properties that can be expressed in second order logic.

A first order quantification is a quantification over nodes of a graph. A second order quantification is over relations defined over nodes, such as $EDGE_G(u, v)$.

CS 15-855 Computational Complexity Theory **September 25, 2000**

Lecture 6: The Complexity of Counting

Lecturer: Steven Rudich *Scribe: Shuchi Chawla / Editor:*

Synopsis: Complexity classes for counting. Sharp-P, Sharp-P completeness. parsimonious reducibility. # SAT, # Perfect Bipartite Matching, # Cycle Cover and Permanent of a matrix as # P complete problems.

22 Counting Classes

In the last lecture, we studied problems of the form $\exists x_1 \forall x_2 \ldots Q_n x_n\ \phi$.

We know that sets of the form $\{y \mid \exists x \phi(x, y)\}$ are in NP, sets of the form $\{y \mid \exists x_1 \forall x_2 \phi(y, x_1, x_2)\}$ are in Σ_2, and sets of the form $\{y \mid \exists x_1 \forall x_2 \ldots Q_n x_n \phi(y, x_1, x_2, \ldots, x_n)\}$ are in PSPACE.

A main thrust of this lecture is to consider the following question:

For a given ϕ, how many x are there such that $\phi(x)$ holds?

Such questions regarding the number of solutions to a problem are classified into **Counting Classes**, for example, $\#P$.

23 Sharp-P

$\#P$ or **Sharp P** (also known as *Number* P or *Pound* P) is the class of functions which count the number of satisfying assignments to polynomial problems. Formally:

Definition 23.1 (Sharp P) . *A function f is said to be in $\#P$ (denoted $f \in \#P$) iff there exists a non deterministic Turing machine M (running in polynomial time), such that, $M(x)$ has $f(x)$ accepting paths for all inputs x.*

Example: Let $f(\phi) = \#$ of satisfying assignments to ϕ. Construct $M(\phi) \in$ NP, such that, M guesses an assignment X and accepts ϕ iff $\phi(X)$ is true. In other words, M

is the NP machine for SAT. Hence, $f \in \#P$. Such a function f can also be called $\#SAT$ or Sharp SAT.

Some other examples of counting problems are

Hamiltonian : Given a graph G, how many hamiltonian cycles does it have?

3 Color : How many 3-colorings does a graph G have?

Triangles : How many triangles does a graph G have?

Graph reliability : How many subgraphs of a graph G contain a path from 1 to n?

23.1 Position of #P in the World View

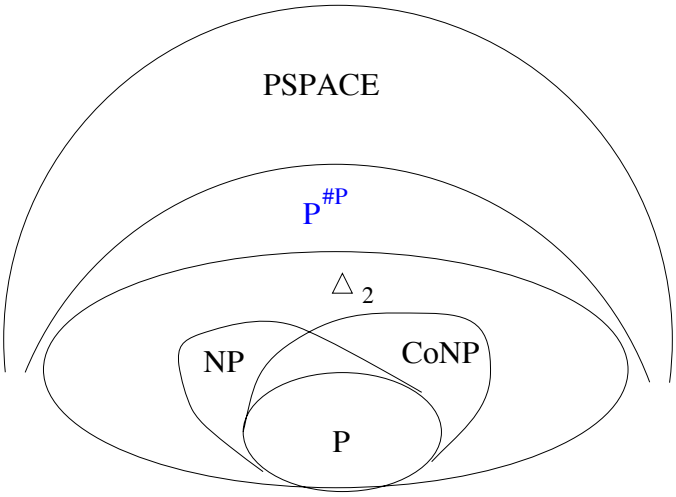


Figure 12: Modified World View

23.1.1 Decision versions of Counting classes

To observe the position of Counting classes in the hierarchy of complexity classes (*world view*), we consider decision versions of counting classes which are classes having access to counting oracles.

$P^{\#P}$ is a polynomial time machine which gets advice from a $\#P$ oracle. Similarly, $P^{\#SAT}$ is a polynomial time machine which gets advice from a $\#SAT$ oracle. Clearly, $P^{\#SAT} = P^{\#P}$, since a $\#SAT$ oracle can answer any question which any other $\#P$ oracle can answer.

Notice that $\mathsf{NP} \subset P^{\#SAT}$ since we can solve SAT using a $\#SAT$ oracle. Similarly, $\Delta_2 = P^{SAT} \subset P^{\#SAT}$. However, $P^{\#P} \subset \mathsf{PSPACE}$ since a machine in $P^{\#P}$ can be simulated deterministically in polynomial space as both $\#P$ and P can be simulated in PSPACE. Hence, our world view now looks like Figure 12.

Question: Is $\mathsf{FP} = \#P$?

If $\mathsf{FP} = \#P$, then $\#SAT \in \mathsf{FP}$, which implies that we can solve SAT in polynomial time (by solving $\#P$ and answering 1 if the solution is positive). This suggests, $P = NP$. Hence, most people believe that FP is strictly contained in $\#P$.

24 $\#P$ Completeness

Definition 24.1 *f is said to be* **#P-Hard** *iff* $\#P \subset FP^f$.

Definition 24.2 *f is said to be* **#P-Complete** *iff f is #P-Hard and $f \in \#P$.*

24.1 # SAT is $\#P$ complete

$\#P$ completeness of $\#SAT$ follows easily from NP-completeness of SAT. In the proof for NP-completeness of SAT, we use Cook-Levin reduction, which reduces the acceptance problem for any Nondeterministic Turing machine M and input x to an instance of SAT, such that each accepting path corresponds to exactly one assignment of the corrsponding SAT. Hence, the number of satisfying assignments to SAT is the same as the number of accepting paths in NTM M. Hence, we can reduce the problem solved by M to $\#SAT$. Such a reduction is known as **Parsimonious Reduction**.

Definition 24.3 *f is* **Parsimoniously reducible** *to g iff there exists $r \in FP$ such that for all x, $f(x) = g(r(x))$.*

By the above definition, Cook-Levin gives a Parsimonious reduction from any problem in $\#P$ to $\#SAT$. This is because Cook-Levin reduction gives a 1-1 map between

accepting paths in the NP-complete problem and satisfying assignments to the corresponding instance of SAT.

Note that parsimonious reducibility is an equivalence relation as it is symmetric and transitive. Accordingly, two functions are called **parsimoniously equivalent**, if one is parsimoniously reducible to the other.

24.2 Some more #P Complete problems

Definition 24.4 *Let $G = (U, V, E)$ be a bipartite graph, in which $|U| = |V|$ and $E \subset U \times V$. $M \subset E$ is a* **Perfect Bipartite Matching** *iff every node in U is incident with precisely one edge in M and every node in V is incident with precisely one edge in M. Equivalently, we may say that there exists a bijection $\pi : [1..n] \to \pi[1..n]$ so that $u_i v_{\pi(i)} \in M$ for each $i \in [1..n]$.*

We will shortly demonstrate how $\#SAT$ is reducible to # Cycle Cover, which is parsimoniously reducible to # Perfect Bipartite Matching and Permanent of a matrix. This will imply that $\#SAT$ is reducible to # Perfect Bipartite Matching which implies that the latter is $\#P$ Complete.

Contradiction!!

Notice, that if $\#SAT$ is *Parsimoniously Reducible* to # Perfect Bipartite Matching, then we can reduce SAT to Perfect Bipartite Matching. To see this, use the argument that if there exists a Perfect Bipartite Matching, then number of PBM's of the graph is greater than zero. By applying parsimonious reducibilty, this means that number of satisfying assignments to the corresponding SAT is greater than zero and hence the formula is satisfiable.

However, Perfect Bipartite Matching can be done in polynomial time. Hence, if SAT is reducible to PBM, then $\mathsf{P} = \mathsf{NP}$! This means that our argument has gone wrong somewhere.

The catch is that $\#SAT$ is *not* parsimoniously reducible to # PBM. $\#SAT$ is only Turing reducible to # PBM. As a result, the number of satisfying assignments to SAT are related to the number of PBMs of the corresponding graph, but the two are not equal.

Perfect Bipartite Matching is parsimoniously equivalent to # Cycle Cover and Permanent of a matrix.

24.2.1 # PBM is parsimoniously reducible to # Cycle cover

Definition 24.5 *Let G be a directed graph. A* **Cycle Cover** *of G is a set of Node disjoint cycles covering every node.*

Observation: Cycle Cover and Perfect Bipartite Matching are parsimoniously equivalent.

Proof: Let $G = (U, V, E)$ be a Bipartite Graph. Construct a graph $G' = (W, E')$ such that $|W| = |U| = |V|$ and $(w_i, w_j) \in E'$ iff $(u_i, v_j) \in E$. It is clear that a cycle cover of graph G' defines a PBM on G and every PBM on G can be expressed as a cycle cover on G'. Hence the two problems are parsimoniously equivalent. ∎

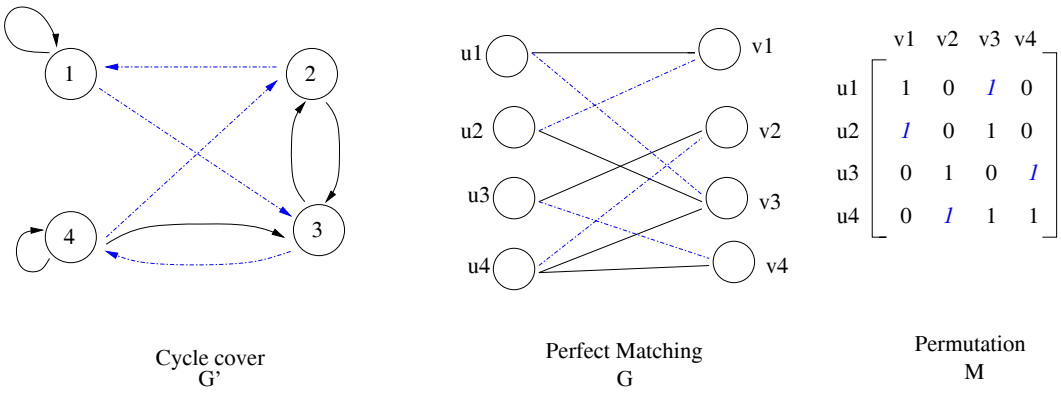


Figure 13: Equivalence of PBM, # cycle cover and permanent of a matrix

24.2.2 # PBM is parsimoniously reducible to Permanent of a matrix

Definition 24.6 *Let M be a $n \times n$ binary matrix. The* **Permanent** *of M is the sum of matrix values of all permutations on $[1..n]$. That is, if π is a permutation on $[1..n]$, $\prod_{i=1}^{i=n} m_{i,\pi(i)}$ is the value corresponding to that permutation, and Permanent of $M = \sum_{\pi} \prod_{i=1}^{i=n} m_{i,\pi(i)}$.*

Observation: Perfect Bipartite Matching and Permanent of a matrix are parsimoniously equivalent.

Proof: To reduce # PBM to Matrix permanent, for a bipartite graph $G = (U, V, E)$, define a matrix M as $m_{ij} = 1$ iff $(u_i, v_j) \in E$. (Notice that this is the adjacency matrix for G). Then, the value of a permutation in M is 1 iff all edges corresponding to this permutation are present in G. Hence, each permutation with value 1 corresponds to exactly 1 perfect bipartite matching in G. Thus the two problems are parsimoniously equivalent. ■

Example: Consider Figure 13. The dashed edges and italicised numbers demonstrate a PBM on graph G, the corresponding cycle cover on $G\prime$, and the corresponding permutation on Matrix M.

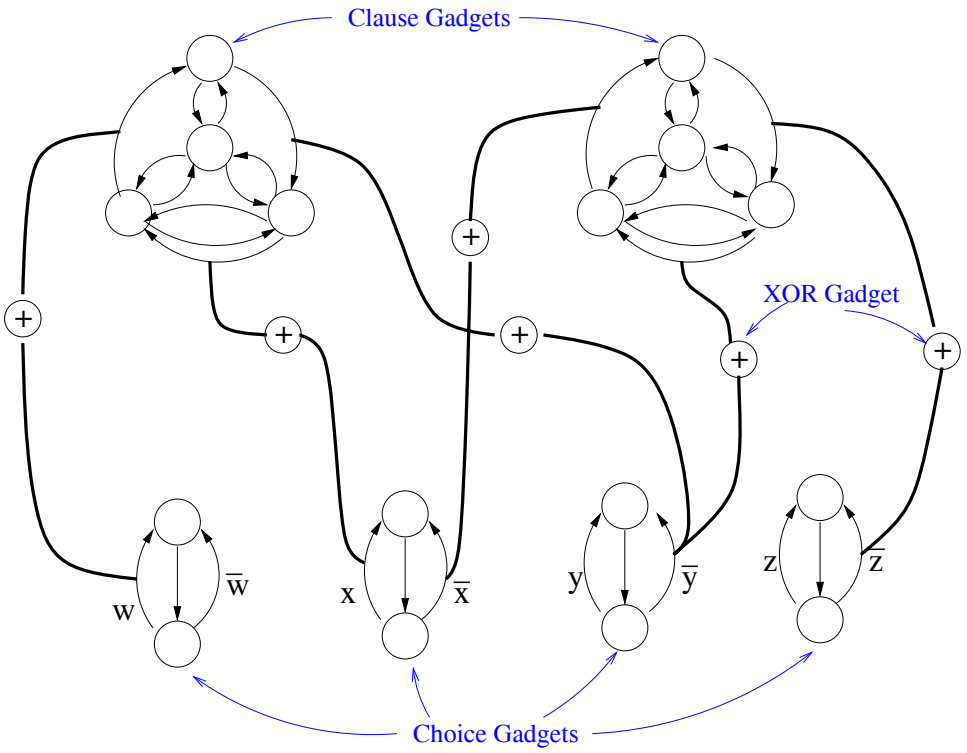


Figure 14: Graph for # 3SAT

24.3 # Cycle cover and related problems are #P Complete

Now we will show a Turing reduction from $\#3SAT$ to # Cycle cover, proving the $\#P$ completeness of the three problems mentioned above. (Notice that $\#SAT$ is reducible to $\#3SAT$ in the same way as SAT is reducible to $3SAT$.)

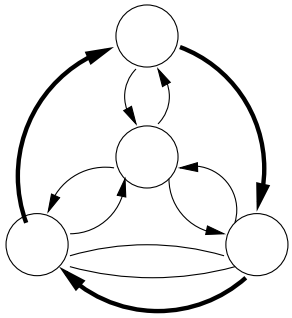


Figure 15: The Clause Gadget

Consider a 3CNF formula of the form $(A \vee B \vee C) \wedge \cdots \wedge (X \vee Y \vee Z)$, where, $A, B, \ldots, Z$ are all literals. In order to cnvert these into a graph, we will separately convert each clause into a subgraph called a *Clause Gadget*. Each variable will have a *Choice Gadget* associated with it, which will help us to decide the value of the literal in the formula. Further, *XOR Gadgets* will help us enforce that value into the formuls. The convertion of a formula into these gadgets and their functioning is demonstrated in Figure 14. We describe the working of the gadgets below.

The Clause Gadgets

Each bold edge in the Clause gadget (Figure 15) corresponds to one literal in the associated clause. Selection of a bold edge signifies value 0 for the corresponding literal. Notice that the clause gadget is so designed, that we cannot select all 3 bold edges in the cycle cover simultaneously, as in that case we will not be able to include the middle node in the cycle cover. Also, there is exactly 1 cycle cover for each of the seven ways to include to not include the bold edges.

The Choice Gadgets

The Choice Gadgets help us select an assignment for the corresponding variable. We should be able to associate this assignment with the choice of the corresponding edge on the Clause Gadgets. For this purpose we use the **XOR Gadgets**. The placement of this gadget between two edges implies that at most one of these edges will be included in a cycle cover. Construction of the XOR Gadget is discussed next.

24.3.1 The XOR Gadget

The XOR Gadget should be designed in a manner enforcing that any cycle cover must use exactly one of the edges to which the gadget is connected. However, recall that

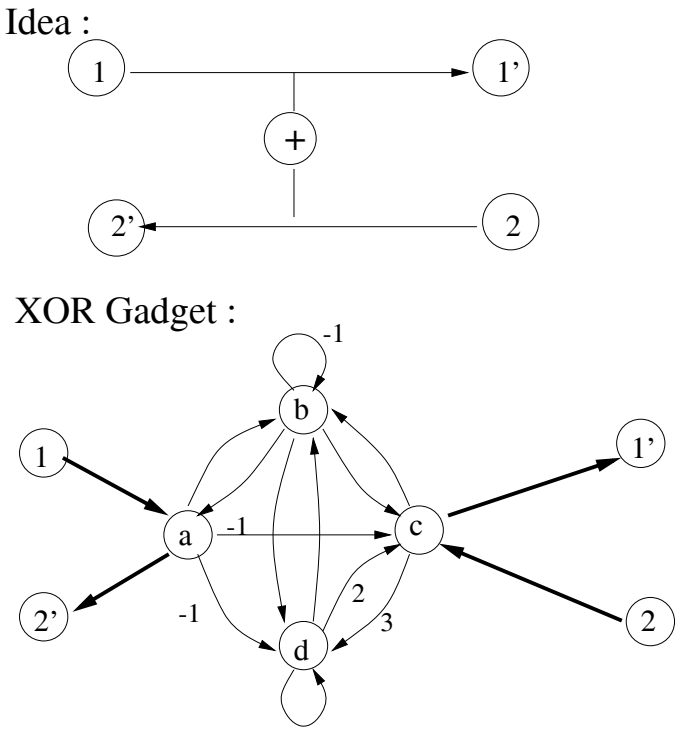


Figure 16: The XOR Gadget

if we are able to construct such a gadget, we will be able to parsimoniously reduce $\#3SAT$ to # cycle cover, which will prove $\mathsf{P} = \mathsf{NP}$ as discussed earlier. We will see that constructing such a gadget is not possible and instead we will construct a gadget which enforces that the number of cycle covers using exactly one of the edges gets multiplied by 4, while the number of cycle covers which dont use exactly one of the edges gets multiplied by a large number N. As a result the obtained reduction is not parsimonious.

In order to construct an XOR Gadget, we first relax the requirement that edges have no weight (or each edge has weight 1). Figure 16 demonstrates the construction of such a gadget.

We define the weight of a cycle cover as the product of its edge weights. Cycle cover count is defined as the sum of weights of all cycle covers. In the Figure, we can see that cycle cover count = 4.

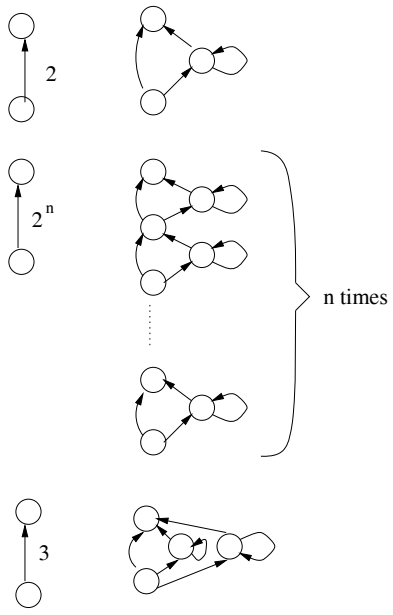


Figure 17: Simulating edge weights

24.3.2 Simulating edge weights

Since in the original problem, edges do not have weights, we need to simulate edge weights by constructing a subgraph such that the number of cycle covers in the original graph get multiplied by the edge weight if an edge is replaced by the appropriate subgraph. These subgraphs are shown in Figure 17. This takes care of all edges with positive weights. However, notice, we have edges with weight -1. In order to simulate this using a subgraph, we consider number of cycle cover modulo some sufficiently large number N, and replace the -1 edge by a subgraph of value $N - 1$.

Let m be the number of XOR Gadgets in the weighted graph case.

Then, each satisfying assignment corresponds to 4^m weight contribution to the cycle cover. Hence, the cycle cover count is at most $2^m \cdot 4^m$ (as number of satisfying assignments $\leq 2^m$). Therefore, the cycle cover count is at most 2^{3m}.

We choose $N = 2^{4m} + 1$ (Sufficiently larger than the possible number of cycle covers). Accordingly, we can replace each edge of weight -1 with a subgraph corresponding to value 2^{4m}.

24.3.3 Correspondence between $\#3SAT$ and # Cycle Cover

We reduced a $\#3SAT$ problem into a graph G and number m such that

The number of satisfying assignments to ϕ is $\frac{(\#\text{ of cycle covers of G}) \mod (2^{4m}+1)}{4^m}$.

Notice that this is not a parsimonious reduction. Due to the presence of 'mod' in the equation, we cannot reduce 3SAT to Cycle cover or PBM in a similar manner.

25 Related things

25.1 Another counting class : Sharp P_1 ($\#P_1$)

Definition 25.1 Sharp P_1. *A function f is said to be in $\#P_1$ iff there exists an NTM M such that for all n, $M(1^n)$ has $f(n)$ accepting paths.*

This class is not nearly as interesting as #P due to the absence of any interesting natural problems known to be complete in it. Valiant in 1977 demonstrated a combinatorial $\#\mathsf{P}_1$ complete problem.

CS 15-855 Computational Complexity Theory **September 29, 2000**

Lecture 8: What can be said about the NP-complete sets?

Lecturer: Rudich *Scribe: Cory Williams / Editor: Jason Crawford*

Synopsis: Mahaney's Theorem: NP-complete sets are not sparse. Isomorphism conjecture: All NP-complete sets are isomorphic to one another. Ladner's Theorem: P $\neq$ NP implies there exists a problem in NP which is not NP-complete or in P.

26 Sparseness

Definition 26.1 *A set S is said to be* **sparse** *if there exists a polynomial $P(n)$ such that the number of strings of length n in S is less than $P(n)$.*

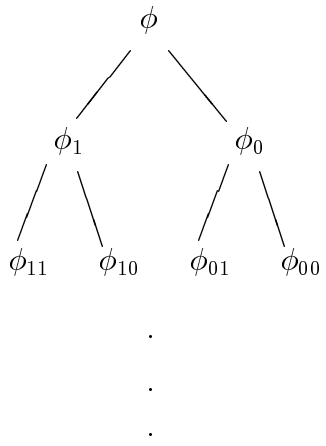
Sparseness is an intermediate step between normal sets and unary sets. As we will see, sparseness is a useful notion in terms of the relationship between P, NP, and NP-complete sets.

26.1 Mahaney's Theorem

We will require a few theorems and corollaries before expressing Mahaney's Theorem.

Theorem 26.1 *(Berman 1978) If S is a unary set and SAT $\leq_L S$, then P = NP.*

Proof: Let ϕ be a formula with variables $x_1, x_2, \ldots, x_n$ and let $\phi_{a_1 a_2 \ldots a_j}$ be ϕ with $x_1 = a_1, x_2 = a_2, \ldots, x_j = a_j$. Then we can define the tree T_ϕ to be the binary tree pictured below.



This can be thought of as a search tree to a satisfying argument.

Now let $r\colon \text{SAT} \to S$ be the reduction function. Then we may observe that

$$|\{r(\phi') \mid |\phi'| \leq |\phi|\}| < |\phi|^k$$

This follows from the fact that fact that there can be at most one element of a given length in S and that $r(\phi)$ can only have length polynomial in the size ϕ. We can also see that if $r(\phi) = r(\phi')$ then ϕ and ϕ' must both be satisfiable or both be unsatisfiable.

Now let us think of the possible values of $r(\phi)$ as colors. We can then think of painting each node of T_ϕ with its color. It is important to note that there are only a polynomial (in the length of ϕ) number of colors being used to color this tree.

Consider a depth first search of the tree T_ϕ. If we stop when we have found a satisfying argument to ϕ then we will have seen exactly $n+1$ satisfiable nodes in the tree, namely those along the path to the satisfying formula. All other nodes that we visited will have been unsatisfiable, otherwise we would have found a satisfying argument earlier. Using this knowledge, we can create a pruning rule for a depth first search of the tree T_ϕ. If we see a color more than $n+1$ times, then we know that the color must represent unsatisfiable formulas. Thus we can prune that branch of the tree and continue. If such a color represents satisfiable formulas, then we should have reached the bottom of the tree already since we have seen $n+1$ satisfying formulas. If we let c be the number of colors in the tree T_ϕ, then we can see that we will never visit more than $c(n+1)$ nodes. Since there are only a polynomial number of colors in the tree, the search will only take polynomial time and thus we have $\text{SAT} \in \mathsf{P}$. ∎

One of the more subtle points in this argument is that we don't care how long it takes to compute S or even if it is computable. We only use the reduction $r(\phi)$ to get an element of S which we used as a color in the tree T_ϕ.

The statement of the above theorem is strengthened in the following corollaries.

Corollary 26.2 *(Fortung 1979) If S is sparse and $\overline{\text{SAT}} \leq_L S$ then $P = NP$.*

Proof: Note that the argument for the above theorem will work as long as the number of unsatisfiable colors is polynomially bounded. Formulas of length less than $|\phi|$ can only map to things in S of length $|\phi|^k$ and there is only a polynomial number of elements of S of length less than this, since S is sparse. ∎

Definition 26.2 *We write $S \leq_{P/\log} T$ to say that there is a reduction from S to T using a Turing Machine that takes polynomial time and receives* O(log)n) *advice.*

Corollary 26.3 *(Mahaney) If S is sparse and $\overline{\text{SAT}} \leq_{P/\log} S$ then $P = NP$.*

Proof: If suffices to note that we have proved $\mathsf{P} = \mathsf{P}/\log$ in homework 3.3b. ∎

At this point this point we are almost ready to prove Mahaney's Theorem. We need one lemma first.

Lemma 26.4 *If S is sparse and $S \in NP$ then $\overline{S} \in NP/log$.*

Proof: To decide if $x \notin S$ we can take as advice the number of elements of S of length $|x|$. Since there are a polynomial number of such elements, the number can be written in O($\log n$) bits. We can then guess all the elements of S of length $|x|$ and proofs for them. We can then easily decide if $x \in A$ or not. ∎

Theorem 26.5 *(Mahaney 1980) If S is sparse and NP-complete, then $P = NP$.*

Proof: If S is NP-complete then we know that SAT $\leq_{\mathsf{L}} S$. This implies that $\overline{\text{SAT}} \leq_{\mathsf{L}} \overline{S}$ By the previous lemma, we know that $\overline{S} \in \mathsf{NP}/\log$ and thus $\overline{\text{SAT}} \in \mathsf{NP}/\log$.

Now consider the meaning of the statement $\overline{\text{SAT}} \in \text{NP/log}$. This can be write as

$$\overline{\text{SAT}} = \{\phi \mid (\phi, A(|\phi|) \in T\}$$

where $A(n)$ is the advice function and $T \in \text{NP}$. But since S is NP-complete, know that there exists a function f in L such that $x \in T \Leftrightarrow f(x) \in S$. Thus we can rewrite the above equation as

$$\overline{\text{SAT}} = \{\phi \mid f(\phi, A(|\phi|) \in S\}$$

Thus we can conclude that $\overline{\text{SAT}} \leq_{\text{P/log}} A$, and by Mahaney's Corollary, $\text{P} = \text{NP}$. ■

27 Isomorphism Conjecture

If all NP-complete sets could be shown to isomorphic to one another, the study of them would be greatly simplified. Berman and Hartmanis proved a weaker result in 1977. They demonstrated a polynomial time version of the Schroder-Berstein Theorem and used it to show that all known NP-complete sets are polynomial time isomorphic. More formally, for any two sets S and T, there exists a polynomial time bijection from S to T whose inverse is also computable in polynomial time. From this they conjectured the following

Conjecture 27.1 *All NP-complete sets are polynomial time isomorphic to one another.*

The following intuition against the Isomorphism Conjecture was given in class.

> **Intuition**: Let f be a polynomial time bijection which scrambles its input in a pseudo-random way. Then the set $S = f(\text{SAT})$ is NP-complete, but f has no polynomial time inverse.

We can find examples of such an f in cryptography, where such functions are used to encrypt messages.

Unfortunately, this intuition does not hold up and was negated by a recent paper published by Allendar, Agrawal, and Rudich. This paper showed that all sets that are

NP-complete under AC_0 reduction are AC_0-isomorphic and that all natural encodings of NP-complete sets are complete under AC_0 reductions. We have not studied the complexity class AC_0, but it is similar to NC but allows unbounded fan-in AND and OR gates.

28 Ladner's Theorem

So far, all the problems in NP that we have studied have been shown to be NP-complete. This begs the question of whether or not there are problems in NP that are not in P and not NP-complete. Ladner's Theorem answers this question.

Theorem 28.1 *(Lander 1975) If P $\neq$ NP, there is a language in NP which is neither in P nor is it NP-complete*

Proof: The idea used here is delayed diagonialization. We will diagonalize against all NP-complete sets and all P sets, but it will not be explicitly clear where the set we will create will be different from any particular P set or NP-complete set.

Let S be a set in P and let Z be an NP-complete set. By hypothesis, we know that $S \neq Z$. From these sets, we will construct a set L which has different parts of S and Z interlaced. More specifically, we will construct a set $CUTANDPASTE$ such that $L = (S \cap CUTANDPASTE) \cup (Z \cap CUTANDPASTE)$. We will do this is such a way that any NP-complete set will differ from L in at least one section that comes from S, and and P set will differ from L in at least one section that comes from Z.

We will first need to enumerate all P machines and all NP-complete machines. We may enumerate all P machines by enumerating all possible Turing machines and then adding "clocks" to each one so that machine M_i will reject if it runs more than $|x|^i$ steps on input x. We will denote this new machine by P_i. Note that we can enumerate machines so that every machine gets enumerated infinitely often. Thus we enumerate every machine with arbitrarily large polynomial clocks.

Now we may enumerate NP-complete machines as follows. We first enumerate all possible logspace reductions, R_i. We can do this in a manner similar to P_i, using a "ruler" to measure space instead of a clock for time. Now we can enumerate the machines NP_n as follows. Consider n as a pair so that $n = \langle i, j \rangle$. Then we compute $NP_n(x)$ by checking that for all y such that $|y| \leq |x|$, $y \in \text{SAT} \Leftrightarrow M_i(R_i(x))$ accepts. If we find this to be the case, then $NP_i(x)$ accepts when $M_i(x)$ accepts. Otherwise $NP_i(x)$ will accept when x is in SAT.

Now we shall define a few functions. Let $r_1(n)$ be the first d such that for all $i \leq n$ there exists $d' < d$ such that $|d'| \geq n$ and S and NP_i differ on element d'. Define $r_2(n)$ similarly except that Z and P_i differ on d'. Then we choose $r(n)$ such that it is a proper complexity function and $r(n) \geq \max(r_1(n), r_2(n))$.

Now we can see that the elements between n and $r(n)$ will differentiate P_j from Z and NP_j from S for $j \leq n$. Let us define (for reason we will see later) $r^1(0) = r(0)$ and $r^i(0) = r^{i-1}(r(0))$. Again, the elements between $r^i(0)$ and $r^{i+1}(0)$ will differentiate P_j from Z and NP_j from S for $j \leq i$.

We are ready to construct $CUTANDPASTE$. It will be the set $\{x \mid r^{2n}(0) \leq |x| < r^{2n+1}\}$.

> **Claim:** $CUTANDPASTE \in \mathsf{P}$
>
> Given an x, we can start computing $r^1(0), r^2(0), \ldots$ until we find k such that $r^k(0) \leq |x| < r^{k+1}(0)$. We will accept x if k is even. The one problem is that $r^{k+1}(0)$ may be extremely large. Note that $r(n)$ is proper complexity function and therefore can be computed in $O(r(n))$ time. We can use a clock to time the computation of $r^i(n)$ and abort the computation if it takes too long. We then know that $r^i(0)$ is much larger than $|x|$ and thus we know k is $i - 1$.

Then $L = (S \cap CUTANDPASTE) \cup (Z \cap CUTANDPASTE)$. We can see that $L \in \mathsf{NP}$. To check if a particular x is in L, we first check if $x \in CUTANDPASTE$. If it is, then $x \in L$ if and only if $x \in S$. Otherwise, $x \in L$ if and only if $x \in Z$. We can also see that $L \neq NP_i$ for any i. Choose an even k such that $i \leq r^k(0)$. Then we know that there is an x such that $r^k(0) \leq |x| < r^{k+1}$ and NP_i and L will differ on x. Similarly we can see that $L \neq P_i$ for any i. ∎

CS 15-855 Computational Complexity Theory **October 2, 2000**

Lecture 9: Enter Randomness

Lecturer: Steven Rudich *Scribe: Konstantin Andreev / Editor:*

Synopsis: Comunication Complexity, Checking Matrix Arithmetic, Verifying Arithmetic, Classes RP, ZPP, BPP. Adelman's Theorem, Sipser's Theorem, Schwartz-Zippel's Theorem. Lovasz's randomized algorithm for perfect matchings.

First we give several beautiful examples of the strength of randomized algorithms. In all of them we assume we can flip an unbiased independent coins.

29 Examples of randomized algorithms

29.1 Communication complexity

Two parties Alice and Bob try to determine whether their two n-bit numbers x and y are equal. If they exchange the information bit by bit they will end up in the worst case complexity n and on average with $n - \log n$. There is clever way to verify the equality of the two numbers with certain probability. Bob chooses a random prime number between $2 \leq r \leq 4n^2$. He computes $x \mod r$ and sends it along with r. Alice on the other end verifies if $x = y \mod r$ and sends back the answer. For one iteration the complexity is $2 \log 4n^2 = 4 \log 2n = \mathrm{O}(\log n)$. As shown in Homework 1.2b

$$\Pr[x \neq y \mod r | x \neq y] \geq \frac{1}{2}$$

In other words if $x \neq y \mod r$ then we are certain that $x \neq y$, otherwise we have certainty more than $1/2$ that $x = y$. We can amplify certainty by iterating the protocol k times. If there is an r such that $x \neq y \mod r$ then we are certain that x is different than y. Otherwise we have $x \equiv y(\mod r)$ for all k random choices of r and we have certainty more than $1 - 1/2^k$ that $x = y$.

29.2 Checking matrix arithmetic.

Let M_1, M_2 and N are $n \times n$ matrices over a field F. We want to verify if $M_1 \times M_2 = N$. The best known deterministic algorithm (using Strassen's Algorithm) has time complexity $O(n^{2.37})$. With probabilistic algorithm we can do better. Let r be a random 0-1 bit vector. We multiply the matrix equation from the right with it and we verify the result. The time complexity of this operation is $O(n^2)$.

Fact 2 *If* $M_1 \times M_2 \neq N$ *then* $\Pr_r[M_1 \times M_2 \times r \neq N \times r] \geq 1/2$.

Proof:

Let $M = M_1 \times M_2 - N$ which by assumption is different from the matrix with all zero's. We look at $M \times r$. There is at least one element of M different from zero. If we have exactly one element different from zero in a row the probability that this element is multiplied by a 0 is equal to the probability that it is multiplied by 1. If we have more than one non zero element in a row j, say $M_{j1} \neq 0$, than for every linear combination $\sum_{i=1}^{n} r_i M_{ji} = 0$ we have at least one corresponding linear combination $(1 - r_1)M_{j1} + \sum_{i=2}^{n} r_i M_{ji} \neq 0$. Thus $\Pr[M \times r \neq 0] \geq 1/2$.

■

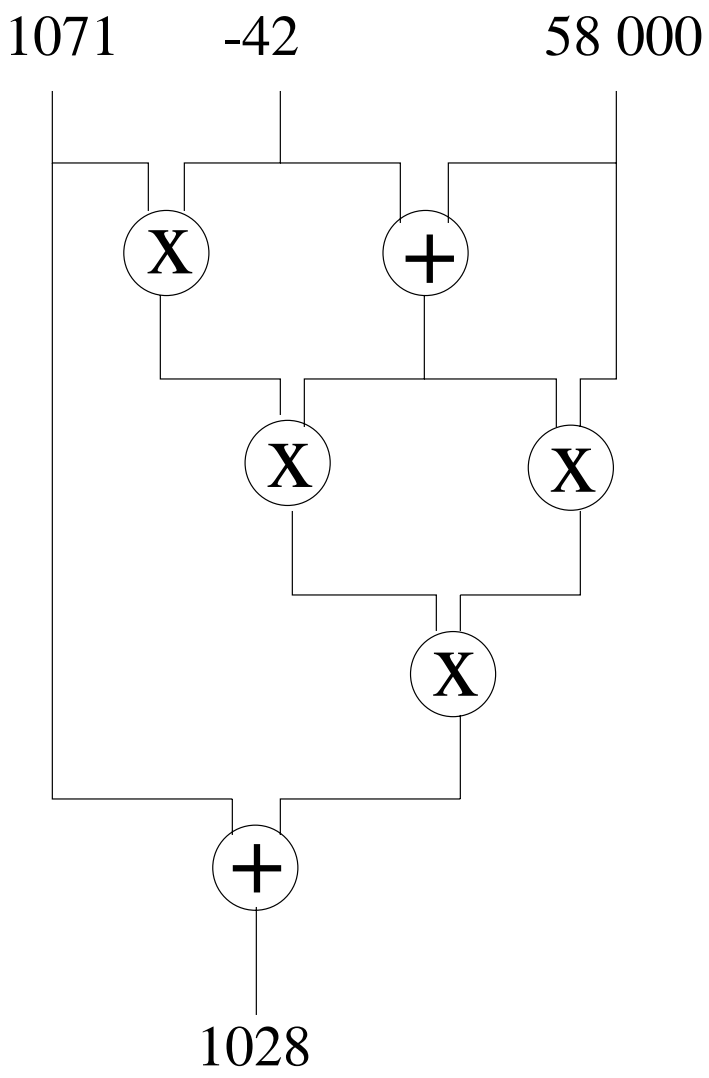
Again if we see an r such that $M_1 \times M_2 \times r \neq N \times r$ we are certain that $M_1 \times M_2 \neq N$. Otherwise we have probability at least $1/2$ that $M_1 \times M_2 = N$. We can amplify the certainty by iterating the protocol.

29.3 Verifying arithmetic

Randomization gives us polynomial time improvement for the previous problem. Now we will give an example in which randomization gives us an exponential time and possibly exponential space improvement.

Suppose we are given a circuits with multiplication and addition gates instead of AND and OR gates. Such a circuits is called arithmetic circuits. We want to verify the output on an given input for this arithmetic circuits.

There is no known deterministic polynomial algorithm to verify the output, since the intermediate results can be exponential in size with respect to the input and still have a polynomial size output. See figure 1



If we apply the same trick as in the case of communication complexity we can get around the intermediate value problem. Choose a large enough prime p. Evaluate the circuit mod p. If the result we are given our evaluation will disagree with that value with probability at least 1/2. If the value that is given is correct than the algorithm will always accept.

30 The RP complexity class

Definition 30.1 *A language A is in RP (randomized polynomial time) if there exists a nondeterministic TM M running in polynomial time such that*

1) M accepts A

2) If $x \in A$ then at least 1/2 of all computational paths accept.

Equivalent way of stating 1) and 2) is

1) If $x \notin$ A then no paths accept.

2) If $x \in$ A then at least half of the paths accept.

We can imagine this as a machine which made all this nondeterministic choices by flipping a fair coin.

Examples of languages in RP are COMPOSITENESS, CHECKING ARITHMETIC CIRCUITS.

30.1 Amplification of certainty

Let $A \in$ RP we can amplify our probability of acceptance by running the corresponding TM, M several times. We accept if and only if the machine accepts one time. Then we have if the machine runs k times

1) $x \notin A$ then $M(x)$ accepts with probability 0

2) $x \in A$ then $M(x)$ accepts with probability greater than $1 - 1/2^k$

If we run the same procedure polynomialy many times instead of constant number of times we get the following equivalent definition of RP.

Definition 30.2 *A language $A \in$ RP if and only if $\forall k$ there exists a probabilistic polynomial time TM M such that*

1) If $x \notin A$ then M(x) rejects always,

2) If $x \in A$ then M(x) accepts with probability at least $1 - 2^{-|x|^k}$.

Definition 30.3 *We define a new class ZPP = RP∩co-RP.*

Remark:

It is true by definition that RP $\subset$ NP. Like NP it is not known that RP $=co-$RP. One example of the significance of this question is PRIMES. It is known that PRIMES $\in$ NP∩co-NP. We also know that PRIMES $\in$co-RP. In 1987 Adelman and Huang proved that PRIMES is in RP. This implies that PRIMES $\in$ ZPP which was a major advancement in computational complexity.

31 The BPP complexity class

We define the class BPP standing for Bounded probability polynomial time.

Definition 31.1 *We say that a language A is in BPP if there exists a probabilistic polynomial time TM M, such that*

1) If $x \in A$ *then* $\Pr[M(x)\ accepts] \geq \frac{3}{4}$,

2) If $x \notin A$ *then* $\Pr[M(x)\ accepts] \leq \frac{1}{4}$.

It is clear that BPP $\subset \mathsf{P}^{\sharp P}$. It is not known whether BPP $\subset$ NP. We will prove in this lecture that BPP $\subset \Sigma_2 \cap \Pi_2$.

31.1 Certainty amplification for BPP

We can amplify the certainty of the answer by running BPP algorithm k-times and then taking the majority of the answers. More precicely as we saw in Homework 1.5 *A useful calculation* if we take $k = 6|x|^c$ then

Corollary 31.1 *1) If* $x \in A$ *then*

$$\Pr[M(x)\ accepts] \geq 1 - \frac{1}{2^{|x|^c}},$$

2) if $x \notin A$ *then*

$$\Pr[M(x)\ accepts] \leq \frac{1}{2^{|x|^c}}.$$

Lets overview the material in this lecture up to now. If a language $A \in \mathsf{BPP}$ then we know it has an efficient randomized algorithm. If $A \in \mathsf{RP}$ we have that A has an efficient Monte Carlo algorithm, i.e. an expected polytime algorithm that makes limited number of misclasifications on words in the language and makes no mistake on words outside of the language. If $A \in \mathsf{ZPP}$ then we know that it has a Las Vegas algorithm, i.e. an expected polynomial time algorithm that answers with certainty. Examples of problems in ZPP are PRIMALITY and FIND SQUARE ROOTS MODULO A PRIME. Still open problem is if $\mathsf{P} = \mathsf{RP}$ or even if $\mathsf{P} = \mathsf{BPP}$. However we are almost sure that PRIMALITY is in P!

Theorem 31.2 Miller'76

If the Extended Riemman Hypothesis holds then PRIMALITY$\in P$.

There are deeper reasons as well.

32 Where does BPP fit in our world picture?

In this section we will present some results how does BPP fit in the world picture.

Theorem 32.1 Adelman

$$BPP \subset P/poly.$$

We will give four "different" proofs of this theorem. Lets first state a slightly modified definition of BPP which uses TM with auxiliary input. We can think of probabilistic polynomial time TM as a polynomial time TM with enough auxiliary input of random bits. Combining this observation with the certainty amplification we give the following equivalent definition of BPP.

Definition 32.1 *A language $A \in BPP$ if there exists a polynomial time TM M, such that*

1) If $x \in A$ then

$$\Pr_r[M(x,r) \text{ accepts}] \geq 1 - \frac{1}{2^{|x|^2}},$$

2) If $x \notin A$ then

$$\Pr_r[M(x,r) \text{ accepts}] \leq \frac{1}{2^{|x|^2}},$$

If r causes $M(x,r)$ to misclassify x we say that r is bad for x. If not,we say that r is good for x.

Lemma 32.2 *There is an r which is good for all inputs of length n.*

If we prove the above Lemma then we will have proven the Theorem. We have that for any language $A \in \mathsf{BPP}$ is true that $A \in \mathsf{P}/poly$ because we can give r as the advice string for inputs of length n and $M(x,r)$ will never misclassify.

Lemma 32.3 *Suppose that for inputs of length n, r is a string of length n^2. Let $R = \{0,1\}^{n^2}$ be the set of all binary strings of length n^2. Then $\exists r \in R$ such that r is good for every input x of length n.*

Proof: *Counting Argument*

For each of the 2^n inputs there are at most a $1/2^{n^2}$ fraction of r's which are bad for it. We throw them away. We tossed out a $2^n/2^{n^2}$ fraction of r's. The remaining are good for all inputs.

■

Proof: *Probabilistic Method*

$\Pr_r[(r \text{ is bad for } x_1) \vee (r \text{ is bad for } x_2) \vee (r \text{ is bad for } x_3) \ldots \vee (r \text{ is bad for } x_n)]$
$\leq \sum_{i=1}^{n} \Pr[r \text{ is bad for } x_i] = \frac{2^n}{2^{n^2}}$.

Hence

$$\Pr_{r \in R}[(r \text{ is good for all } x)] \geq 1 - 2^{-n^2+n}.$$

■

Proof: *Method of Expectations*

Define 0-1 random variables V_x as 1 if r is bad for x and 0 otherwise. Lets

$$V = \sum_x V_x.$$

In this way V is the number of x's for which r is bad.

$$E[V] = \sum_x E[V_x] = \sum_x \frac{1}{2^{n^2}} = \frac{2^n}{2^{n^2}}.$$

Which implies that $E[V] < 1$, so $\exists r$ such that $V < 1$. Since V must be an integer for this r we have $V = 0$, i.e. r is good for all x.

■

Proof:*Kolmogorov-Chaitin randomness*

Suppose $M(x, r)$ misclassifies x, i.e. r is bad for x. By the same argument used in Homework 4.3b we have

$$K(r) \leq |x| + \log(2^{|r|-|x|^2}) + C \leq |r| - |x^2| + |x| + C << |r|.$$

If r is n-random we know that $K(r) \geq |r| - n$ which means that r is good for all x.

■

Combining all this results, we have $\mathsf{BPP} \subset \mathsf{P}/poly$.

Remark:

The four proofs look different, but they are essentially the same.

One more result about BPP.

Theorem 32.4 Sipser

$$\mathit{BPP} \subset \Sigma_2 \cap \Pi_2.$$

Proof:

With out loss of generality pick a TM M with error less than $1/2^{|x|}$ on $x \in L$. Lets define the set

$$A(x) = \{r \mid M(x,r) \text{ accepts}\}.$$

Now we define a translation of $A(x)$ with a vector t as

$$A(x) \oplus t = \{r \oplus t \mid r \in A(x)\}.$$

Lemma 32.5 *If*

$$\frac{|A(x)|}{|R|} > 1 - \frac{1}{2^{|x|}},$$

then there exist $t_1, t_2, \ldots, t_{|r|}$ *such that* $|t_i| = |r|$ *and*

$$\bigcup_{i=1}^{|r|} A(x) \oplus t_i = R.$$

Proof:

Again we will use the probabilistic method. Pick a random sequence $t_1, t_2, \ldots, t_{|r|}$. Let

$$S = \bigcup_{i=1}^{|r|} A(x) \oplus t_i.$$

For all r in R we have

$$\Pr[r \notin S] = \Pr[r \notin A(x) \oplus t_1] \cdot \Pr[r \notin A(x) \oplus t_2] \cdots \Pr[r \notin A(x) \oplus t_{|r|}] = \left(\frac{1}{2^{|x|}}\right)^{|r|} = \frac{1}{2^{|x|\cdot|r|}}.$$

Now using the well known union inequality we get

$$\Pr[(r_1 \notin S) \vee (r_2 \notin S) \vee \ldots \vee (r_{2^{|r|}} \notin S)] \leq \sum_{r_i} \frac{1}{2^{|x|\cdot|r|}} = \frac{2^{|r|}}{2^{|x|\cdot|r|}} << 1.$$

Which means that

$$\Pr[S = R] \geq 1 - \frac{1}{2^{|x|}}.$$

■

Lemma 32.6 *If $x \notin L$ that is $|A(x)|/|R| \leq 1/2^{|x|}$ we have there does not exist $t_1, t_2, \ldots, t_{|r|}$ such that*

$$R = \bigcup_{i=1}^{|r|} A(x) \oplus t_i.$$

Proof sketch:

$$\frac{|r|}{2^{|x|}} << 1.$$

■

This means that $x \in L$ if and only if there $\exists t_1, t_2, \ldots, t_{|r|}$ such that $\forall r \in R$ $M(r \oplus t_1)$ accepts or $M(r \oplus t_2)$ accepts or $\ldots$ $M(r \oplus t_{|r|})$ accepts. The latter is a Σ_2 predicate. Since BPP is closed under compliment, it must also be included in Π_2.

■

It is worth mentioning that M. Sipser's original proof was the first application of hash functions to complexity theory.

33 Schwartz-Zippel's theorem and applications

We will conclude this lecture with two more examples of the power of randomized algorithms.

Lets look at the decision problem: We are given a n-variable polynomial over a finite field presented in the form of a product of brackets and we are asked to verify if this

polynomial is equivalently 0. For example is $(x_1 + 3x_2 - x_3)(3x_1 + x_4 - 1) \ldots (x_7 - x_2)$ equivalent to 0 over the field with 7^{10} elements. Their is no known deterministic polynomial time algorithm to answer this question. However we can answer this question with certain probability using the following theorem.

Theorem 33.1 Schwartz-Zippel

Let $P(x_1, x_2, \ldots, x_n) \not\equiv 0$ be a degree d multivariable polynomial over the field F for any finite $S \subset F$, if we pick at random $r_1, r_2, \ldots, r_n \in S$ we have

$$\Pr[P(r_1, r_, \ldots, r_n) = 0] \leq \frac{d}{|S|}.$$

Proof:

We will proof the Theorem by induction on the number of variables. The base case is $n = 1$. Then we have a univariate polynomial $p(x) \not\equiv 0$ of degree d. It has at most d roots, so

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

Lets assume that the conditions holds for all polynomials on less than n variables. Let $P(x_1, x_2, \ldots, x_n)$ be a polynomial on n variables. We rewrite P with respect to the degree of the first variable

$$P(x_1, x_2, \ldots, x_n) = \sum_{i=0}^{d} x_1^i P_i(x_2, \ldots, x_n).$$

By assumption $P \neq 0$, so there exists an i such that $P_i \not\equiv 0$. We pick the maximum such i. Lets pick $r_2, \ldots, r_n \in S$ at random. By the induction hypothesis

$$\Pr[P_i(r_2, \ldots, r_n) = 0] \leq \frac{d - i}{|S|}.$$

Because of our choice of i, the univariate polynomial $P(x_1, r_2, \ldots, r_n)$ is of degree i. Now from the base case of the induction we have

$$\Pr[P(r_1, r_2, \ldots, r_n) \mid P_i(r_2, \ldots, r_n) \neq 0] \leq \frac{i}{|S|}.$$

Hence, when we add up the two conditional probabilities we get

$$\Pr[P(r_1, r_2, \ldots, r_n) = 0] \leq \frac{d}{|S|}.$$

■

33.1 A randomized algorithm for perfect matchings on bipartite graphs and Lovasz's corollary

The following result is due to Edmonds

Theorem 33.2 Edmonds 1967

Let A be the $n \times n$ matrix obtained from a bipartite graph G as follows:

$$A = \begin{cases} A_{ij} = 1 & \text{If there is an edge between } u_i \text{ and } v_j \\ A_{ij} = 0 & \text{Else} \end{cases}$$

Then the $\det(A) \neq 0 \Leftrightarrow$ *G has a perfect matching.*

Since the determinant can be expanded as a sum over all the permutations we have

$$\det(A) = \sum_{\pi} \operatorname{sgn}(\pi) A_{1,\pi(1)} A_{2,\pi(2)} \dots A_{n,\pi(n)}.$$

There is no possible cancellation of non zero terms in the expansion.

Based on this Theorem Lovasz proved a corollary and gave a randomized algorithm for perfect matchings on bipartite graphs. The $\det(A)$ is a polynomial of degree at most n. Here is the algorithm.

- Pick a finite field F such that $|F| \geq 2n$.
- Pick $r_1, r_2, \dots, r_{|E|} \in F$ at random.
- Create a new matrix $\tilde{A}$ by substituting the value of r_i for x_i in A.
- Compute the $\det(\tilde{A})$.

If G has a perfect matching then by Schwartz-Zippel and Edmond's Theorems, $\det(\tilde{A}) \neq 0$ at least half the time. In 1987 Mulmuley, Vazarani and Vazarani used this to prove a stronger result, namely that PERFECT MATCHING $\in$ RNC.

CS 15-855 Computational Complexity Theory **October 4, 2000**

Lecture 10: Randomized Logspace (RL)

Lecturer: Rudich *Scribe: Luis von Ahn / Editor: Maverick Woo*

Synopsis: Definition of RL. UST-CONN is in RL: Random Walks.

34 RL

We now define the class of languages RL: $A \in \mathsf{RL}$ if there exists a non-deterministic logspace machine M such that for $x \notin A$, $M(x)$ does not accept and for $x \in A$ the computation of M on x has at least half of its paths accept. Alternatively, we can think of putting a probability measure on the paths by flipping a coin at each non-deterministic choice and requiring that:

- $x \in A \implies Pr[M(x) \text{ accepts }] \geq 1/2$.
- $x \notin A \implies Pr[M(x) \text{ accepts }] = 0$.

A technical note is required here: we have defined non-deterministic machines so that all paths terminate. If this were not the case, then we would have that $\mathsf{NL} = \mathsf{RL}$.

35 UST-CONN $\in$ RL

UST-CONN is the analogue of ST-CONN for undirected graphs. Recall that we have shown that ST-CONN is NL-complete. We now show that for undirected graphs, the problem of deciding whether nodes s and t are connected is in RL.

Theorem 35.1 UST-CONN $\in$ *RL*

Proof: Here's the algorithm (which takes an undirected graph $G = (V, E)$ and two elements of V, s and t):

> Start at s and make $2n^3$ random steps. (Taking a random step means the following: when you are about to take a step, you choose unformly from all the neighbors of the node your are in in order to detemrine the next node you will be in.) If you ever see t, accept. Otherwise reject.

We now analyze the algorithm. For this we will need to analyze random walks on graphs.

Let W be an infinite random walk starting from node s on an (undirected, connected) graph $G = (V, E)$. For each $a \in V$ define ϕ_a as the frequency of node a in W. That is,

$$\phi_a := \lim_{n\to\infty} \frac{\#W_n(a)}{n},$$

where $\#W_n(a)$ is the number of times a is seen in the first n steps of W. An important point is to prove that ϕ_a, being a limit, indeed exists. We will not, however, prove this. For $< a, b > \in E$, let $\phi_{<a,b>}$ be the frequency with which W goes accross $< a, b >$ (starting from a).

Claim: Let d_a be the degree of $a \in V$. Then $\phi_{<a,b>} = \frac{\phi_a}{d_a}$

Proof: This should be obvious: the number of times W goes through $< a, b >$ is exactly the number of times W goes through a divided by the degree of a (since, at a, all neighbors of a are equally likely to be chosen by W).

Claim: Let $N(a)$ be the set of neighbors of a in G. Then:

$$\phi_a = \sum_{b\in N(a)} \phi_{<b,a>}.$$

Proof: This should again be clear.

From the previous claims, we get:

$$\phi_{<a,b>} = \frac{\sum_{b'\in N(a)} \phi_{<b',a>}}{d_a}.$$

Lemma 35.2 *$\phi_{<a,b>}$ is maximal $\implies$ $\phi_{<a,b>} = \phi_{<b',a>}$ for all $b' \in N(a)$.*

Proof: Otherwise, we would have that $\phi_{<a,b>}$ is greater than the average over $b' \in N(a)$ of $\phi_{<b',a>}$. ■

Lemma 35.3 *$\phi_{<a,b>}$ is the same for all edges $< a, b > \in E$.*

Proof: Repeated application of the previous lemma and the fact that G is connected yield the result.

■

Lemma 35.4 *$\phi_{<a,b>} = \frac{1}{2|E|}$ for all edges $< a, b > \in E$.*

Proof: $\sum_{<a,b>\in E} \phi_{<a,b>} = 1$, by definition, so the result follows. The 2 is here because we count each edge twice (as the defintion of $\phi_{<a,b>}$ takes the direction of the edge into account, but G is undirected). ■

Corollary 35.5 *$\phi_a = \frac{d_a}{2|E|}$ for all $a \in V$.*

Proof: We have that $\phi_{<a,b>} = \phi_a / d_a$ so:

$$\frac{1}{2|E|} = \frac{\phi_a}{d_a}.$$

■

Now, for each pair of vertices a and b, define $T(a, b)$ to be the expected number of steps W takes to go from a to b.

Lemma 35.6 *$T(a, a) = \frac{1}{\phi_a} = \frac{2|E|}{d_a}$.*

Proof:

$$T(a, a) = \lim_{n\to\infty} \frac{n}{n\phi_a}$$

■

Lemma 35.7 *If a and b are adjacent, then $T(a,b) \leq 2|E|$.*

Proof: Recall that $\phi_{<a,b>} = \frac{1}{2|E|}$. Let $D(a,b)$ be the total distance in W_n of the $[a,b]$ intervals (that is, the total number of nodes traversed by W in its first n steps on paths from a to b). Notice that $D(a,b) \leq n$. Let $N(a,b)$ be the number of $[a,b]$ intervals in W_n. $N(a,b) \geq \frac{n}{2|E|}$. Hence:

$$T(a,b) = \lim_{n\to\infty} \frac{D(a,b)}{N(a,b)} \leq 2|E|.$$

■

Now, let $a_1, a_2, ..., a_{2(n-1)}$ be a walk that visits all of G (notice that such a walk does exist: take a spanning tree of G for instance). Define S to be the expected length of a random walk in order for it to contain the subsequence $a_1, a_2, ..., a_{2(n-1)}$. We have (since the sum of the expectations is the expectation of the sum) that

$$S = \sum_{i=1}^{2(n-1)-1} T(a_i, a_{i+1}) \leq 2(n-1)2|E|.$$

It is, then, clear that the expected number of steps W takes to traverse G is smaller than $2|E|(n-1) \leq n^3$. And, we have:

Corollary 35.8 *The probability that a random walk from a vertex a of G fails to visit all of G in $2n^3$ steps is less than or equal to 1/2.*

Proof: This is a simple consequence of Markov's inequality. ■

Notice that this proves the correctness of our algorithm. ■

As a note, we can add that UST-CONN is actually in $\mathsf{RL} \cap co-\mathsf{RL}$. This was proved by Borodin, Cook, Dymond, Ruzzo and Tompa in 1989.

Remark: (directed)ST-CONN$\in$ RL would imply that RL = NL. This, however, is unlikely.

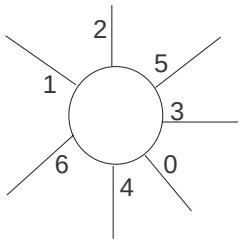
36 Universal Sequences

> The theory of Universal Sequences is of utmost importance to frequent museum goers.
> -Mike Sipser.

Definition 36.1 *A d-regular graph is one in which all nodes have degree d.*

Let G be a d-regular undirected graph. Assume that each edge is doubly labelled, once at each end:

At each node, the d edges going out (or in) are labelled $0, 1, 2, ..., d-1$ in some order:



Note that you can interpret a sequence of numbers (all from 1 to d) as a particular way to walk the graph.

Definition 36.2 *Call a sequence $S \in \{0, 1, ..., d-1\}^*$ universal for d-regular graphs with n nodes if: For any labelling of an n-node d-regular graph G, and for any start node, following the sequence of moves given by S will cause you to visit all nodes in G.*

Proposition: For every n, a universal sequence S_n exists.

Proof: Let us first enumerate all the different labellings of a d-regular n node graph and all start nodes: $G_1, G_2, ..., G_k$. Let S_n be the concatenation of the following sequences:

- Δ_1: a sequence to solve G_1.
- Δ_2: a sequence to solve G_2 starting from where Δ_1 left off (assuming it was performed in G_2).
- Δ_3: a sequence to solve G_3 starting from where $\Delta_1\Delta_2$ left off (assuming it was performed in G_3).

  .
  .
  .

■

Theorem 36.1 *A universal S_n exists where $|S_n| = O(n^3 \log n)$.*

Proof: Let R_n be a random sequence of length $cn^3 \log n$.

Claim: Pr[R_n is universal]> 0.

Proof: Consider a fixed d-regular labelled n-node graph G. Then $Pr[$ a walk of length $2|E|(n-1) = O(n^2)$ fails to visit $G] \leq 1/2$. So, $Pr[R_n$ of length $O(n^2)O(n \log n)$ fails to visit G$] \leq 1/2^{O(n \log n)}$. Now, how many n-node d-regular labelled graphs are there? We can sepcify one using $O(n \log n)$ bits: For each node we write a $d \log n$ bit list of neghbors and a $\log d!$ bit list of out labels. This proves the assertion.

■

By the probabilistic method, this proves our assertion.

■

We now know that there are universal sequences of length $O(n^3 \log n)$. The following are still open questions, though:

1. Is there a polynomial size universal sequence where producing the i-th symbol is in logspace?

2. Is there a way to construct a polynomial sequence in time polynomial in n? (We know that there is an $n^{\log n}$ time way to construct an $n^{\log n}$ length sequence. We also know that a polynomial time way to construct a universal sequence exists for the particular case when $d = 2$.)

It has been conjectured that the length of the shortest universal sequence S_n satisfies:

$$|S_n| \leq \frac{n(n+1)}{2}$$

(The conjecture has been verified for the cases when $n \leq 8$ using computer experiments.)

CS 15-855 Computational Complexity Theory **October 6, 2000**

Lecture 11: The Formal Foundations of Pseudorandomness

Lecturer: Steven Rudich *Scribe: Martin Zinkevich / Editor:*

Synopsis: Indistinguishable ensembles, statistical tests that sample once as opposed to many times. Definition of pseudorandom generator. Hidden bits. A generator obtained by concatenating a hidden bit to the end of the permutation that hides it. One-way functions and permutations.

37 What is Randomness?

When one calls the function rand() in the standard C library, for most implementations the value returned is not really random, unless one has a plug-n-play quantum bit card. Usually the value returned is generated by the rule:

$$\begin{aligned} x_0 &= \text{(random seed)} \\ x_i &= ax_{i-1} + b \bmod m \end{aligned}$$

Where the seed is typically generated by some bits of the clock at the time the program is run. Clearly the bits returned by this function are not "actually" random (although they are pairwise independent when m is prime). However, these bits "look random". We can perform several standard statistical tests, which it will pass, for example the χ^2 test. In fact, this generator is "random enough" for many practical applications. One could consider its output a "patternless pattern". How can these concepts be formalized? That is the topic of this lecture.

We will say that two things "look the same" if you cannot tell them apart. Thus when they are different, we will expect that they have some characteristic that would be different. If we wish to claim that something does not "look random," we need to show that it does not share some detectable characteristic with a random sequence. A random sequence has several such characteristics. For instance, consider the bit sequence:

10	11	01	01	11	00	10	10	11	01

Let us count the number of times each type of 2-digit subsequence occurs:

Subsequence	Number of Occurrences
00	1
01	3
10	3
11	3

If the sequence was random, with a high probability the number of occurrences of each subsequence would be equal. Since this is not the case, one can be fairly certain that this sequence of bits was not generated by a random process, because it does not share this characteristic. This is an example of a *statistical test.*

Definition 37.1 *A* sequence generator *is an efficiently computable rule by which we take any short initial sequence and associate it with a long output sequence.*

Example: $g_n(x) = b_1\ b_2\ b_3\ \ldots\ b_l$ where the initial string x is some integer between 0 and n.

$x^2 \bmod n \to b_1$ (0 or 1 depending on whether $x^2 \bmod n$ is odd or even)

$x^4 \bmod n \to b_2$

$x^8 \bmod n \to b_3$

$\vdots$

$x^{2^l} \bmod n \to b_l$

This is called a *repeated squaring generator.*

Note that, from a theoretical perspective, it is insufficient to have a sequence which looks random to all tests in current commercially available statistical analysis software. To truly appear random, from the theoretical perspective, a sequence should look random for any test that one could devise and execute efficiently.

We will say that a generator is *pseudorandom* if for any efficiently computable test for randomness(including those not yet invented) the output of the generator "looks random" for all but a tiny fraction of initial sequences x.

A major achievement in complexity theory and the foundations of cryptography was the discovery that factoring is hard if and only if the repeated squaring generator

is pseudorandom: if there exists any efficient test discerning something non-random about the output of the repeated squaring generator, this test can be converted into an efficient factoring algorithm. The proof of this equivalence can be found in [1]. The more general result, developed in part by Yao [4], Blum and Micali [2], and Goldreich and Levin [3], states that a one-way function exists if and only if a pseudorandom generator exists.

38 Computational Indistinguishability

Definition 38.1 *An ensemble is a family* $\{X_n\}$ *of random variables, when* X_n *ranges over strings of length* n.

Example: $\{U_n\}$ where U_n is the uniform distribution on Σ^n.

In statistics, there is a straightforward concept of closeness:

Definition 38.2 *The ensembles* $X = \{X_n\}$ *and* $Y = \{Y_n\}$ *are* $\Delta(n)$ statistically close *if*

$$\Delta(n) < \sum_{\alpha} |Pr[X_n = \alpha] - Pr[Y_n = \alpha]|$$

Example: Let X and Y be ensembles, for which the distributions of X_2 and Y_2 are given in the table below:

α	$Pr[X_2 = \alpha]$	$Pr[Y_2 = \alpha]$	$\lvert Pr[X_2 = \alpha] - Pr[Y_2 = \alpha]\rvert$
00	0.3	0.2	0.1
01	0.1	0.1	0.0
10	0.3	0.4	0.1
11	0.3	0.3	0.0

In this example, $\sum_\alpha |Pr[X_2 = \alpha] - Pr[Y_2 = \alpha]| = 0.2$.

Definition 38.3 *Given two vectors* $\vec{x}$ *and* $\vec{y}$, *the* L_1 norm *of the difference is*

$$L_1(\vec{x}, \vec{y}) \equiv \sum_i |x_i - y_i|$$

Definition 38.4 *$\{X_n\}$ and $\{Y_n\}$ are $S(n)$* secure computationally indistinguishable *or $\frac{1}{S(n)}$* computationally close, *denoted $\{X_n\} \cong \frac{1}{S(n)}\{Y(n)\}$, if for every circuit C_n of size $T(n)$:*

$$\frac{|Pr[C_n(X_n) = 1] - Pr[C_n(Y_n) = 1]|}{T(n)} \leq \frac{1}{S(n)}$$

If we are discussing uniform pseudorandomness, then we replace "Every circuit C_n of size $T(n)$" with "Every TM running in time $T(n)$". The circuit (or TM) C_n is called a *statistical test.*

These two definitions are similar, but they are not identical.

Lemma 38.1 *If $\{X_n\}$ and $\{Y_n\}$ are $\varepsilon(n)$ statistically close then they are $\varepsilon(n)$ computationally close.*

The proof is left as an exercise.

One can easily prove that they are different. Consider the ensemble $\{X_n\}$ where the nth bit is the parity of the first $n-1$ bits and $\{U_n\}$. Then $\Delta(n) = 1/2$. Observe that any $n-1$ of the bits of X_n are independent and uniformly distributed, so the only way to detect the difference is to use all n bits, which will take at least $\log n$ time and $n-1$ gates. Thus the ensemble $\{X_n\}$ is $(n-1)$ computationally indistinguishable from uniform. One can think of statistical closeness as a computational closeness if one has infinite computational power.

Definition 38.5 *If $\{X_n\} \cong \frac{1}{P(n)}\{Y_n\}$ for every polynomial $P(n)$, we will simply write $\{X_n\} \cong \{Y_n\}$, and say that they are* polynomially indistinguishable.

Lemma 38.2 *If $f \in FP$ and $\{X_n\} \cong \{Y_n\}$, then $\{f(X_n)\} \cong \{f(Y_n)\}$.*

Proof: Let $\{F_n\}$ be a polynomial-sized, uniform circuit family computing f on inputs of length n (we know such a family exists since $f \in FP$.), and let k be such that $|f(x)| \leq |x|^k$. Let $\{C_{n^k}\}$ be a circuit family distinguishing $f(X_n)$ from $f(Y_n)$ with gap $\varepsilon(n^k) > 1/p(n^k)$ for some polynomial $p(\cdot)$, that is:

$$\frac{|Pr[C_{n^k}(f(X_n)) = 1] - Pr[C_{n^k}(f(Y_n)) = 1]|}{|C_{n^k}|} > \frac{1}{p(n^k)}.$$

Then the circuit $D_n = C_{n^k} \circ F_n$ has size $|D_n| = |F_n| + |C_{n^k}|$; and gap

$$|Pr[D_n(X_n) = 1] - Pr[D_n(Y_n)) = 1]| > |C_{n^k}| \frac{1}{p(n)}.$$

But since $|C_{n^k}|/|D_n|$ is polynomial in n, this means there exists a polynomial $q(\cdot)$ contradicting the $1/poly$ indistinguishability of $\{X_n\}$ and $\{Y_n\}$.

Lemma 38.3 *There exists a $\{X_n\}$ such that $\{X_n\}$ ranges over less than $2^{n/2}$ strings and $\{X_n\} \cong \{U_n\}$.*

Note that these two ensembles are statistically quite different!

Proof: Consider all $\{X_n\}$ which are uniform over some set S_n, $|S_n| \leq 2^{n/2}$. S_n can be shown to exist by a probabilistic argument: Uniformly choose the strings $s_1, s_2, \ldots s_{2^{n/2}}$ from the set $\{0,1\}^n$. For some fixed circuit C_n, let p_n denote $Pr[C_n(U_n) = 1]$. Let $q_i = C(s_i)$ Noting that $E[q_i] = p_n$, we apply the Chernoff bound to get:

$$Pr\left[|p_n - 2^{-n/2} \sum_{i=1}^{2^{n/2}} q_i| > 2^{-n/8}\right] \leq 2e^{-2 \cdot 2^{n/2} \cdot 2^{-n/4}} < 2^{-2^{n/4}}$$

Which means that with probability at least $1 - 2^{-2^{n/4}}$, we choose a sequence which works for C_n (i.e. has computational gap less than $2^{-n/8}$ for C_n). Since there are at most $2^{2^{n/4}}$ circuits of size $2^{n/8}$, this means that there is a non-zero probability of randomly picking a S_n which is $2^{n/8}$ secure computationally indistinguishable from U_n (actually $Pr[S_n \cong_{2^{n/8}} U_n] \geq (1 - 2^{-2^{n/4}})^{2^{2^{n/4}}} \approx 1/e$). Thus such an ensemble must exist.

Definition 38.6 *A function $G\colon \Sigma^* \to \Sigma^*$ is a pseudorandom generator if:*

1. *G is polynomial time computable*
2. *$G\colon \{0,1\}^k \to \{0,1\}^l$*
3. *$\{U_k\} \cong \{G(U_k)\}$*

Definition 38.7 *A function $g : \Sigma^* \to \Sigma^*$ is an $\varepsilon(n)$* pseudorandom generator with stretch l(n) *if:*

1. $g \in \cup_{c \in \mathbb{N}} FTIME(l^c(n))$
2. $g\colon \{0,1\}^n \to \{0,1\}^{l(n)}$
3. $\{g(U_n)\} \cong \frac{1}{\varepsilon(l(n))}\{U_{l(n)}\}$
4. $l(n) > n$

Example: If $X_m \equiv g(U_n)$ where $m = l(n)$, $\{X_m\}$ is $\varepsilon(n)$ pseudorandom.

Theorem 38.4 *If $\{X_n\} \cong \{Y_n\}$, then they are also indistinguishable to polynomial-size circuits that can draw a polynomial number of samples:*

$$|Pr[C_n(X_n^1, X_n^2, X_n^3, \ldots, X_n^p(n)) = 1] - Pr[C_n(Y_n^1, Y_n^2, Y_n^3, \ldots, Y_n^p(n)) = 1]| \leq \frac{1}{n^c}.$$

The proof technique uses hybrid distributions, distributions which are a mixture of two other distributions.

Proof(by contrapositive):

Suppose that $|Pr[C_n(X_n^1, \ldots, X_n^p(n)) = 1] - Pr[C_n(Y_n^1, \ldots, Y_n^p(n)) = 1]| > \frac{1}{n^c}$

First, we will define a set of hybrid distributions:

$$\begin{aligned}
H_n^0 &= \langle X_n^1, X_n^2, X_n^3, \ldots, X_n^{p(n)-1}, X_n^{p(n)} \rangle = X_n \\
H_n^1 &= \langle X_n^1, X_n^2, X_n^3, \ldots, X_n^{p(n)-1}, Y_n^{p(n)} \rangle \\
H_n^2 &= \langle X_n^1, X_n^2, \ldots, Y_n^{P(n)-2}, Y_n^{p(n)-1}, Y_n^{p(n)} \rangle \\
&\vdots \\
H_n^{p(n)-1} &= \langle X_n^1, Y_n^2, Y_n^3, \ldots, Y_n^{p(n)-1}, Y_n^{p(n)} \rangle \\
H_n^{p(n)} &= \langle Y_n^1, Y_n^2, Y_n^3, \ldots, Y_n^{p(n)-1}, Y_n^{p(n)} \rangle = Y_n
\end{aligned}$$

We know that C_n can "distinguish" between H_n^0 and $H_n^{p(n)}$; we'll show that C_n can also distinguish X_n and Y_n.

Definition 38.8 *C_n has a* gap *of $f(n)$ on $(\{X_n\}, \{Y_n\})$ if*

$$|Pr[C_n(X_n) = 1] - Pr[C_n(Y_n) = 1]| \geq f(n).$$

Lemma 38.5 Gap Lemma *If C_n has gap greater than $f(n)$ on $(\{X_n\},\{Y_n\})$ and a gap greater than $g(n)$ on $(\{Y_n\},\{Z_n\})$ then it has a gap less than $f(n)+g(n)$ on $(\{X_n\},\{Y_n\})$.*

Proof: We are given the bounds:

$$|Pr[C_n(X_n)=1]-Pr[C_n(Y_n)=1]|\leq f(n)$$

and

$$|Pr[C_n(Y_n)=1]-Pr[C_n(Z_n)=1]|\leq g(n)$$

and we are interested in bounding the gap

$$|Pr[C_n(X_n)=1]-Pr[C_n(Z_n)=1]|.$$

Let $p_V=Pr[C_n(V_n)=1]$. Rewriting the last gap and applying the triangle inequality, we get the desired bound:

$$\begin{aligned}|p_X-p_Z| &= |p_X-p_Z+(p_Y-p_Y)|\\ &= |(p_X-p_Y)+(p_Y-p_Z)|\\ &\leq |p_X-p_Y|+|p_Y-p_Z|\\ &\leq f(n)+g(n)\end{aligned}$$

Corollary 38.6 *There exists a polynomial $S(n)$ such that for all n there exists an i such that C_n has a gap of $\frac{1}{S(n)}$ on (H_n^i,H_n^{i+1}).*

Since these two distributions differ only in the $p(n)-i$th input, there exists some setting of the other inputs such that the gap is retained. We can prove that this can be accomplished recursively. Suppose that U and V are random vectors for which

$$Pr[C(U)=1]-Pr[C(V)=1]>r$$

, and define

$$b=Pr[U_0=1]=Pr[V_0=1]$$

where U_0 and V_0 denote the first bits of U and V, respectively. Let P_σ denote the event $U_0=\sigma$, let Q_σ denote the event $V_0=\sigma$, and define q_0, p_0, q_1, and p_1 such that

$$\begin{aligned}Pr[C_n(U)=1|U_0=0]&=p_0\\ Pr[C_n(U)=1|U_0=1]&=p_1\\ Pr[C_n(V)=1|V_0=0]&=q_0\\ Pr[C_n(V)=1|V_0=1]&=q_1\end{aligned}$$

Then

$$Pr[C_n(U) = 1] = bp_1 + (1 - b)p_0$$

and

$$Pr[C_n(V) = 1] = bq_1 + (1 - b)q_0$$

and

$$|Pr[C_n(U) = 1] - Pr[C_n(V) = 1]| = |b(p_1 - q_1) + (1 - b)(p_0 - q_0)|$$

Thus the gap on (U, V) is the weighted average of the gap on (P_0, Q_0) and the gap on (P_1, Q_1). Therefore, one of these two gaps is at least as large as (U, V). Hence, we can set the first bit without tightening the gap.

Thus if one fixes all the inputs for which the two distributions H_n^i and H_n^{i+1} do not differ, then the resulting circuit can discriminate between X_n and Y_n with the same or larger gap as on (H_n^i, H_n^{i+1}).

As we mentioned at the outset of the proof, this technique is called a hybrid argument. You should be sure to review it if you don't understand it; we will be getting some serious mileage out of this technique in the remainder of the course.

References

[1] L. Blum, M. Blum, and M. Shub. A simple unpredictable random number generator. *SIAM Journal on Computing*, 15, pp. 364–383 (1986).

[2] M. Blum and S. Micali. How to Generate Cryptographically Strong Sequences of Random Bits. *SIAM Journal on Computing*, 13, pp 850-864 (1984). Preliminary version in *23rd IEEE Symposium on Foundations of Computer Science*, 1982.

[3] O. Goldreich and L.A. Levin Hard-Core Predicates for any One-Way Function. In *21st ACM Symposium on Theory of Computing*, pp 25–32, 1989.

[4] A.C. Yao Theory and Application of Trapdoor Functions. In *23rd IEEE Symposium on Foundations of Computer Science*, pp. 80–91, 1982.

CS 15-855 Computational Complexity Theory **11 October, 2000**

Lecture 13: Hardness Versus Randomness

Lecturer: Rudich *Scribe: Nikhil Bansal / Editor: Martin Zinkevich*

Synopsis: How does randomness help. Relating power of randomness to one-way functions and hardness of functions. Amplifying hardness.

39 What is the power of Randomness

So far we have several complexity classes like RP, BPP, ZPP and others which use randomness. We also saw some problems, like primality testing, which can be solved in polynomial time using randomness, but for which no solution is known in deterministic polynomial time. So in this lecture we will ask the question: What is the power of Randomness? Is $P = RP$ or $P = BPP$? At present we only know that $ZPP \subset \cup_k 2^{n^k}$.

In this lecture we shall show a beautiful duality that exists between hardness and randomness. In particular, if some problems are hard, then other problems are easy. For example we will see that if *factoring* is as hard as it seems to be then, $BPP \in \mathsf{TIME}(2^{\log^c n})$.

39.1 Randomness and One-Way functions

In lecture 12, we observed that, $\exists c > 0$ such that, if $\exists \epsilon(n)$ one-way function, this implies that $\exists\ \epsilon^c(n)$ pseudorandom generator[4] with stretch $\frac{1}{\epsilon(n)}$.

This observation was used by Yao, to relate the results for one-way functions to the time taken by randomized complexity classes.

Theorem 39.1 (Yao) *If an $\epsilon(n)^{\frac{1}{2}}$ pseudorandom generator with stretch $\frac{1}{\epsilon(n)}$ exists, then*

$$BPP - \mathit{TIME}(t) \subset \mathit{TIME}(2^{O(\epsilon^{-1}(\frac{1}{t}))})$$

where ϵ^{-1} is the inverse function of $\epsilon(n)$, i.e. $\epsilon^{-1}(\epsilon(n)) = n$.

[4]The generator runs in time polynomial in the length of its output.

Proof sketch:

- Let M be a BPP algorithm using t random bits.
- Let $g : \epsilon^{-1}(\frac{1}{t})$ bits $\rightarrow t$ bits be an $\epsilon^{\frac{1}{2}}(n)$ pseudorandom generator with stretch $\frac{1}{\epsilon(n)}$.
- M will behave approximately the same way if it is given bits from $g(U_{\epsilon^{-1}(\frac{1}{t})})$ instead of random bits from U_t. Otherwise, M would be a test for g and thus contradict the fact that g is pseudorandom.
- Simulate M on all $2^{\epsilon^{-1}(\frac{1}{t})}$ strings output by g.
- Take the majority outcome.

■

This immediately implies the following:

Corollary 39.2

1. *If a one-way function exists then* $\forall \epsilon > 0, BPP \subset \mathsf{TIME}(2^{n^\epsilon})$
2. *If there is* $\frac{1}{2^{n^\epsilon}}$ *one-way function for some* $\epsilon > 0$ *then* $BPP \subset \mathsf{TIME}(2^{\log^c n})$, *for some* $c \in \mathbb{N}$
3. *If there is a* $\frac{1}{2^{\epsilon n}}$ *one-way function for some* $\epsilon > 0$ *then* $BPP = P$

The above results relate the results about the existence of one-way functions to time taken by randomized algorithms. However, these conditions about one-way functions are rather strong. Noam and Wigderson showed that the above results hold under much weaker conditions. In fact, their results are considered to suggest that probably $P = BPP$.

39.2 Randomness and Hard functions

In this section we will prove Noam and Wigderson's result, which proves results similar to above under much weaker assumptions.

We first begin with an observation.

Observation: For Yao's theorem to work the pseudorandom generator does not need to run in time polynomial is its output[5]. It might as well use time $O(2^n)$ on inputs of length n, since we take 2^n time to enumerate all seeds. In other words, the generator might as well relax.

Definition 39.1 *An* $\epsilon(n)$ pseudorandom generator *g with stretch* $l(n)$ *is said to be* relaxed *if it satisfies the following properties.*

- *Takes* n *bits to* $l(n)$ *bits*
- $g(U_n) \cong_{\epsilon(n)} U_{l(n)}$
- *g runs in time* $O(2^n)$

Definition 39.2 *A function f is* $S(n)$ hard *if,* $\forall$ *circuits* $C(n)$

$$\frac{|Pr_{x\in U_n}[C_n(x) = f(x)] - Pr_x[C_n(x) \neq f(x)]|}{Size\ of\ C(n)} \leq \frac{1}{S(n)}$$

Notation: Let E denote the class $\cup_k \mathsf{TIME}(2^{kn})$ and EXP denote the class $\cup_k \mathsf{TIME}(2^{n^k})$

Theorem 39.3 (Nisan-Wigderson 1988) *If* $\exists\ h \in E$ *such that* h *is* $S^4(n)$ *hard, then there is a* $\frac{1}{S(n)}$ *relaxed pseudorandom generator with stretch* $S(n)$*.*

This implies the following:

Corollary 39.4

- *If* $h \in E$ *is super-polynomial hard, then*

$$BPP \subset \mathsf{TIME}(2^{n^\epsilon}), \forall \epsilon > 0$$

- *If* $h \in E$ 2^{n^ϵ} *hard for some* $\epsilon > 0$ *then*

$$BPP \subset \mathsf{TIME}(2^{\log^c n}), for\ some\ c \in \mathbb{N}$$

[5]This assumption is usually needed in cryptographic applications, however they are not necessary in our scenario, where the simulation of randomized algorithms is considered.

- *If $h \in E$ is $2^{\epsilon n}$ hard then*

$$BPP = P$$

Proof: [Theorem 39.3] We know from Problem set 5, #3, that $\forall$ prime power n, $\exists$ a Logspace computable $S(n) \times n$ matrix A_n that is a $(\log S(n), \sqrt{n})$ design. That is, it satisfies the following properties.

- All rows have exactly $\sqrt{n}$ 1's.
- Any two rows have at most $\log S(n)$ 1's in common.
- There is a logspace machine that will output A_n on input 1^n.

Let $h \in E$ be $S^4(n)$ hard. We will construct a generator g based on h. Let $x = (x_1, x_2, \ldots, x_n)$ be an n bit string. $\forall 1 \leq i \leq S(n)$, the i^{th} bit of output of $g(x_1, x_2, \ldots, x_n)$ is obtained by applying h to the $\sqrt{n}$ bits of x, indicated by the i^{th} row of A_n.

Clearly $g(X)$ is computable in time $O(2^n)$, and it has stretch $S(n)$. We will show that g is $\frac{1}{S(n)}$ pseudorandom generator.

Observe that each bit when view individually is $S(\sqrt{n})$ hard (random), since the value of a bit at position i is just obtained by applying h to a uniform distribution over $\sqrt{n}$ bits on which bit i depends. However, the intuition is that since the bits are based on nearly disjoint sets, they remain hard (random) even when looked at together.

The proof is organized as follows. We will assume that g is not a pseudorandom generator and derive a contradiction to the hardness assumption of h.

If g is not pseudorandom, there is a circuit A such that

$$\frac{|Pr_{x \in U_{S(n)}}[A(x) = 1] - Pr_{x \in U_n}[A(G(x)) = 1]|}{\text{Size of } A} > \frac{1}{S(n)}$$

We will now consider the following lemma, due to Yao.

Lemma 39.5 (Predictor Lemma) *[Yao] Given the circuit A (as described above) exists, there exists a circuit B and $i(n) \in \mathbb{N}$, such that B can predict the next bit of $g(x)$ given $i(|x|)$ bits of $g(x)$ and,*

$$\frac{Pr_{x \in U_n}[B(\text{ First } i(n) \text{ bits of } g(x)) = i + 1^{th} \text{ bit of } g(x)]}{\text{Size of } B} > \frac{1}{S^2(n)}$$

Proof:[**Yao's Lemma**] Let $g(x) = y_1, y_2, \ldots, y_{S(n)}$. Without loss of generality, assume that $Pr_{x \in U_{S(n)}}[A(x) = 1] < Pr_{x \in U_n}[A(G(x)) = 1]$.

Define hybrids H_i, for $1 \leq i \leq S(n)$.

$$H_i = \text{First } i \text{ bits of } g(x) | S(n) - i \text{ random bits}$$

By the Gap Lemma in Lecture 11, $\exists i$ such that A has gap $\frac{1}{S^2(n)}$ on H_i, H_{i+1}.

Fix the random bits so as to maintain the gap of A and define the predictor $B(y_1, y_2, \ldots, y_i)$ as follows:

If $A(y_1, \ldots, y_i, 0, \text{fixed bits}) = A(y_1, \ldots, y_i, 1, \text{fixed bits})$ then flip coin and randomly output a bit.

Else if, $A(y_1, \ldots, y_i, 0, \text{fixed bits}) \neq A(y_1, \ldots, y_i, 1, \text{fixed bits})$, then
if $A(y_1, \ldots, y_i, 0, \text{fixed bits}) = 1$ then output 0, else output 1

Thus we have a circuit B which takes the first i bits of $g(x)$ as input and outputs the $i + 1^{th}$ bit such that

$$\frac{|Pr[B(y_1, y_2, \ldots, y_i) = y_{i+1}] - \frac{1}{2}|}{Sizeof B} > \frac{1}{S^2(n)}$$

■

To achieve a contradiction to the hardness assumption we need a circuit which will compute y_{i+1} from $x = x_1, \ldots, x_n$. Without loss of generality we can assume that $y_{i+1} = h(x_1, , x_2, \ldots, x_{\sqrt{n}})$

Since y_{i+1} does not depend on other bits of x, we can write $Pr[B(y_1, y_2, \ldots, y_i) = y_{i+1}]$ above, as the probability $Pr[B(y_1, y_2, \ldots, y_i) = y_{i+1}]$, where the probability is over x chosen at random, such that only $x_1, \ldots, x_{\sqrt{n}}$ are chosen at random. Thus we can fix $x_{\sqrt{n}+1}, \ldots, x_n$ while maintaining the success over size ratio of predictor B. Now, since we have a $(\log S(n), \sqrt{n})$ design, each one of $y_1, \ldots, y_i$ depends on no more than $\log S(n)$ of the $x's$. So, given the $x's$ a circuit of size $S(n)i$ can be constructed to compute $y_1, \ldots, y_i$ by simulating the pseudorandom generator. Now apply B to $y_1, \ldots, y_i$ computed from x to get a prediction of y_{i+1}. It is clear to see that the success over size ratio of this circuit is at least $\frac{1}{S^4(n)}$. Thus contradicting the hardness of h. Thus the proof follows.

■

In fact, there is better block design. We can construct a $(\log S(n), cn)$ design with $S(n)$ as large as $2^{\epsilon}n$. This construction uses the following *Greedy Method*. For the i^{th} row pick the first set you find that is nearly disjoint from other sets so far. It can be shown using a counting argument that this is possible.

Observe that the greedy method above takes 2^n time. If $n = O(\log S(n))$, then $2^n = S(n)^c$ time.

40 Amplification of Hardness (Yao's XOR-Lemma)

In the previous section, we considered *hard* functions in a very strong sense. The functions we considered hard previously were the ones which could not be approximated almost everywhere. However, we can use Yao's XOR-Lemma to prove similar results about functions which can be approximated almost everywhere. In particular, we can prove similar results about functions f such that

$$\frac{Pr[C_n(x) \neq f(x)]}{Sizeof C_n} > \frac{1}{n^k}$$

for some k.

Theorem 40.1 (Yao's XOR-Lemma) *Suppose $f_1, f_2, \ldots, f_k$ are hard in the following sense:*

If for any C_n, $|C_n| \leq S(n)$ has

$$\forall i, (Pr[C(x) = f_i(x)] - Pr[C(x) \neq f_i(x)]) < \epsilon$$

then $f(x_1, x_2, \ldots, x_k) = \bigoplus_{i=1}^{k} f_i(x_i)$ is harder *in the following sense:*

$\forall \delta > 0, \forall C'_m$ *such that $|C'_m| \leq \delta^2(1-\epsilon)^2 S(n)$, then*

$$(Pr[C'(f(y)) = C(y)] - Pr[C'(f(y)) \neq C(y)]) < \epsilon^k + \delta$$

This gives us following:

Corollary 40.2 *If f is a function computable in $\mathsf{TIME}(2^{O(n)})$, such that for all circuits C_n, $|C_n| \leq S(n)$ and*

$$(Pr[C_n(x) = f(x)] - Pr[C_n(x) \neq f(x)]) \leq (1 - \frac{1}{n^k})$$

for some k, then $\exists f' \in \mathsf{TIME}(2^{O(n)})$ such that f' is $S(n^c)$ hard for some $c > 0$.

In other words, if $f \in \mathsf{TIME}(2^{O(n)})$ cannot be closely approximated by a polynomial sized circuit, then there exists a function $f' \in \mathsf{TIME}(2^{O(n)})$ that cannot be slightly approximated by polynomial sized circuits.

41 Conclusions

Using the results in the previous sections we can conclude the following:

1. If $\mathsf{TIME}(2^{O(n)})$ contains a language that cannot be approximated to with $(1-\frac{1}{n^k})$ by a polynomial-sized circuit family, then $BPP \subseteq \cap_\epsilon(2^{n^\epsilon}TIME)$
2. If $\mathsf{TIME}(2^{O(n)})$ contains a language that cannot be approximated to with $(1-\frac{1}{n^k})$ by circuits of size 2^{n^ϵ} for some $\epsilon > 0$, then $BPP \subseteq \mathsf{TIME}n^{\log^c n}$ (for some $c > 0$).
3. If $\mathsf{TIME}(2^{O(n)})$ contains a language that cannot be approximated to with $(1-\frac{1}{n^k})$ by circuits of size $2^{n\epsilon}$ for some $\epsilon > 0$, then $P = BPP$.

But there is an E-Complete problem that is randomly self-reducible. So, we can change to worst case assumptions:

1. If $\exists h \in E$, and h cannot be computed by polynomial size circuits then $BPP \subset \cap_\epsilon \mathsf{TIME}(2^{n^\epsilon})$
2. If $\exists h \in E$, and h cannot be computed by circuits of size 2^{n^ϵ} for some $\epsilon > 0$, then $BPP \subseteq \mathsf{TIME}n^{\log^c n}$ (for some $c > 0$).
3. (Due to Impagliazzo and Wigderson) If $\exists h \in E$, and h cannot be computed by circuits of size $2^{n\epsilon}$ for some $\epsilon > 0$, then $P = BPP$.

CS 15-855 Computational Complexity Theory **October 16, 2000**

Lecture 15: RANDOMNESS IN REDUCTIONS

Lecturer: Rudich *Scribe: Giacomo Zambelli / Editor: Cory Williams*

Synopsis: Uniquely satisfiable formula and Valiant-Vazirani's Lemma. Definition of DC-uniform circuit families, Toda's Theorem.

42 Valiant-Vazirani's Lemma

Suppose we are given a boolean formula ϕ and we know that, if $\phi \in SAT$ then ϕ has only one satisfying assignment. Can we decide $\phi \in SAT$ efficiently?

Definition 42.1 *An algorithm f is* good *for formulas guaranteed to have at most one solution if*

$$f(\phi) = \begin{cases} \text{accept} & \text{if } \phi \text{ has exactly one} \\ & \text{satisfying assignment} \\ \text{reject} & \text{if } \phi \notin SAT \end{cases}$$

Theorem 42.1 (Valiant, Vazirani [2]) *If there exists an* RP *algorithm f which is good for formulas guaranteed to have at most one solution, then* RP=NP.

Proof: Assuming there exists f as in the statement, we will provide an RP algorithm for SAT. Since SAT is NP-complete, this will imply that RP=NP.

Given positive integers n and k, a $k \times n$ 0,1-matrix M and a vector $\alpha \in \{0,1\}^k$, let $\Gamma_{n,k,M,\alpha}(x_1, ..., x_n)$ be the boolean formula on the variables $(x_1, ..., x_k)$ such that $\Gamma_{n,k,M,\alpha}(x_1, ..., x_n)$ is satisfied if and only if

$$M \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \alpha.$$

Given a set $S \subset \{0,1\}^n$ such that $2^{k-2} \leq |S| \leq 2^{k-1}$, if the coefficients of the matrix M and of the vector α are chosen uniformly at random, then (by Homework 3, problem 4b)

$$Pr[\exists! x \in S \text{ s.t. } Mx = \alpha] \geq \frac{1}{8}.$$

Given any boolean formula ϕ, then $f(\phi)$ rejects with probability 1 if $\phi \notin SAT$, $f(\phi)$ accepts with probability at least $\frac{1}{2}$ if $\phi \in SAT$.
If $S \subset \{0,1\}^n$ is the set of satisfying assignments for ϕ, then

$$Pr[2^{k-2} \leq |S| \leq 2^{k-1}] = \frac{1}{n}.$$

Thus, for any formula $\phi \in SAT$, the probability that $\phi(x_1, ..., x_n) \wedge \Gamma_{n,k,M,\alpha}(x_1, ..., x_n)$ has exactly one solution is, at least $\frac{1}{8n}$.

We define an RP algorithm for SAT as follows.
Given a boolean formula $\phi(x_1, ..., x_n)$:

- Pick k at random in $[1, ..., n]$.
- Pick M a $k \times n$ $0,1$-matrix at random.
- Pick α a k bit $0,1$-vector at random.
- Output $f(\phi(x_1, ..., x_n) \wedge \Gamma_{n,k,M,\alpha}(x_1, ..., x_n))$.

If $\phi \notin SAT$, then $f(\phi(x_1, ..., x_n) \wedge \Gamma_{n,k,M,\alpha}(x_1, ..., x_n))$ rejects with probability 1, if $\phi \in SAT$, then, by the above arguments, $f(\phi(x_1, ..., x_n) \wedge \Gamma_{n,k,M,\alpha}(x_1, ..., x_n))$ accepts with probability at least $\frac{1}{16n}$. Choose a parameter a such that $(1 - \frac{16}{n})^a \leq \frac{1}{2}$. Notice that a only need to be polynomial size in n. Choose uniformly at random a integers k_i in $[1, ..., n]$, a $k_i \times n$ $0,1$-matrices M_i and a k-bit $0,1$-vectors α_i $(1 \leq i \leq a)$. Given $\phi \in SAT$, the probability that, for some i, $f(\phi(x_1, ..., x_n) \wedge \Gamma_{n,k_i,M_i,\alpha_i}(x_1, ..., x_n)) = 1$ is at least $\frac{1}{2}$. ∎

An analogous statement holds for BPP:

Theorem 42.2 *If there exists a* BPP *algorithm f which is good for formulas guaranteed to have at most one solution, then* NP⊆BPP.

43 Toda's Theorem

We have already observed that any language in the Polynomial Hierarchy can be decided using polynomial space. Moreover, every counting problem in #P can be solved using only polynomial space; in fact one can enumerate all the possible solutions reusing the space. How do PH and #P compare in power? Toda's theorem answers to this question.

Theorem 43.1 (Toda [1]) $\mathsf{PH} \subseteq \mathsf{P}^{\#\mathsf{P}}$.

In order to prove Toda's theorem we need to introduce the concept of *Direct Connected Uniform Circuits* and to prove some intermediate results.

43.1 DC Uniform Circuits

Definition 43.1 *Let $\{C_n\}$ be a circuit family of size $S(n)$. Let $TYPE(n,i)$ be an indexing function that outputs the type (AND, OR, NOT, INPUT X_3, OUTPUT, ...) of gate i in circuit C_n. Let $IN(n,i,\delta) = j$ where j is δth in the ordered list of indices to the gates that feed into i (if there is no δth gate, j =NONE). Let $OUT(n,i,\delta) = j$ where j is δth in the ordered list of indices to the gates that i feeds into (if there is no δth gate, j =NONE). $FINDINDEX(n,T) = i$ where i is the index of type T where T can be either of the form INPUT X_k or OUTPUT. If all the above functions are computable in deterministic $O(\log(S(n)))$ time, we say that $\{C_n\}$ is Direct Connect (DC) uniform.*

In Homework 4, problem 5, we proved the following results.

Lemma 43.2 PH *is equivalent to the set of languages accepted by DC-uniform families with constant depth, exponential size, with unrestricted NOT gates, unbounded fan-in AND, and unbounded fan-in OR gates.*

Lemma 43.3 $\oplus\mathsf{P}$ *is equivalent to the set of languages accepted by DC-uniform families with constant-depth, exponential size, and* four *types of gates (XOR, NOT, AND, and OR) such that the XOR gates have unbounded fan-in, the AND and the OR gates have polynomial fan-in, while the NOT gates are allowed to occur anywhere.*

Another technical result we will need is the following:

Lemma 43.4 *Every function computable by an exponential size, DC-uniform family of circuits with unbounded fan-in size $(+)$ gates and polynomially bounded fan-in size $(\times)$ gates and polynomially bounded number of constants can be computed by an $\mathsf{FP}^{\#\mathsf{P}}$ function.*

43.2 Proof of Toda's Theorem

In order to prove Theorem 43.1, we will first show that $\mathsf{PH} \subseteq \mathsf{RP}^{\oplus\mathsf{P}}$ and then $\mathsf{RP}^{\oplus\mathsf{P}} \subseteq \mathsf{P}^{\#\mathsf{P}}$. To accomplish the first step, we state the following.

Theorem 43.5 *An OR gate with fan-in of size 2^d can be replaced by a constant-depth circuit $C_{M,\alpha,k}$ containing AND, OR and XOR gates where the AND and OR gates have fan-in of size d^k for some k and the XOR gates have fan-in of size 2^d, such that*

$$Pr_{M,\alpha,k}[C_{M,\alpha,k}(x) = 1 \mid OR(x) = 1] \geq \frac{1}{8d},$$

where M, α and k are, respectively, a randomly chosen $k \times d$ $0,1$-matrix, a randomly chosen $0,1$-vector of length k and a randomly chosen integer in $[1, ..., d]$.

Proof: Let $(x_1, ..., x_l)$ (where $l = 2^d$) be the input of an OR gate. Construct a circuit $C_{M,\alpha,k}$ has follows. Pick at random an integer k in $[1, ..., d]$, a $k \times d$ $0,1$-matrix M and a $0,1$-vector α of dimension k. Construct a hash filter $H_{M,\alpha,k}$ which takes as input a variable x_i and returns x_i if $M\hat{i} = \alpha$, 0 otherwise, where $\hat{i}$ is the binary vector of length d corresponding to the binary representation of i. Such a hash filter can be built in constant depth, size polynomial in d, using only AND and OR gates with polynomially bounded fan-in.
Replace the original OR gate with an XOR, only instead of feeding in directly the values of the variables $(x_1, ..., x_l)$, we feed in the values of $(H_{M,\alpha,k}(x_1), ..., H_{M,\alpha,k}(x_l))$. Given an assignment $(x_1, ..., x_l)$, let $S = \{i \mid x_i = 1\}$. If $2^{k-2} \leq |S| \leq 2^{k-1}$, then the probability that there is a unique $i \in S$ such that $H_{m,\alpha,k}(x_i) = x_i = 1$ is $\frac{1}{8}$ (as proved in Homework 3, problem 4b). As before the probability that $2^{k-2} \leq |S| \leq 2^{k-1}$ is $\frac{1}{d}$. Therefore, if $\mathrm{OR}(x_1, ..., x_l) = 0$ then $C_{M,\alpha,k}(x_1, ..., x_l) = 0$ with probability 1, if $\mathrm{OR}(x_1, ..., x_l) = 1$ then $C_{M,\alpha,k}(x_1, ..., x_l) = 1$ with probability at least $\frac{1}{8d}$. ∎

Theorem 43.6 $\mathsf{PH} \subseteq \mathsf{RP}^{\oplus\mathsf{P}}$.

Proof: By Lemma 43.2, using De Morgan equality, PH is equivalent to the set of languages accepted by DC-uniform families with constant depth, exponential size, with unrestricted NOT gates and unbounded fan-in OR gates, since we can replace any AND gate by a constant depth circuit using only OR and NOT gates.
Let $\{C_n\}$ be a DC circuit family of this form where n is the input size of C_n. By Theorem 43.5, every OR gate of C_n with large fan-in size (at most 2^{n^h} for some constant h), can be replaced by a constant-depth circuit $C_{M,\alpha,k}$ containing AND, OR and XOR gates where the AND and OR gates have fan-in of size n^k for some k and the XOR gates have fan-in of size 2^{n^h}, such that

$$Pr_{M,\alpha,k}[C_{M,\alpha,k}(x) = 1 \mid OR(x) = 1] \geq \frac{1}{8n^h}.$$

Given a constant a, construct a circuit C composed by an OR gate with fan-in size a, in which the inputs are the outputs of $C_{M_i,\alpha_i,k_i}(x_1, ..., x_l)$, where $1 \leq i \leq a$ and M_i, α_i, k_i are picked uniformly at random.
If $\mathrm{OR}(x_1, ..., x_l) = 0$ then $C(x_1, ..., x_l, M_1, \alpha_1, k_1, ..., M_a, \alpha_a, k_a) = 0$ with probability 1, if $\mathrm{OR}(x_1, ..., x_l) = 1$ then

$$Pr_{\substack{M_1,\alpha_1,k_1 \\ \vdots \\ M_a,\alpha_a,k_a}} [C(x, M_1, \alpha_1, k_1, ..., M_a, \alpha_a, k_a) = 1 \mid OR(x) = 1] \geq 1 - (1 - \frac{1}{8n^h})^a.$$

We want to fix the amplification constant a so that

$$(1 - \frac{1}{8n^h})^a < \frac{1}{2}.$$

Notice that a only need to be polynomial size in n, therefore the AND and the OR gates in C have polynomial fan-in size.
Therefore we can construct a circuit $\tilde{C}_n$ with constant-depth, exponential size, and *four* types of gates (XOR, NOT, AND, and OR) such that the XOR gates have unbounded fan-in while the AND and the OR gates have polynomial fan-in, by replacing each OR gate in C_n by an XOR gadget as above, using the same random bits R in each gadget. At least half of the choices of R will force each XOR gadget to perform properly (i.e. to accept every input $(x_1, .., x_l)$ accepted by the correspondent OR gadget in C_n), therefore any input rejected by C_n will be rejected by $\tilde{C}_n$ with probability 1 and every input accepted by C_n will be accepted by $\tilde{C}_n$ with probability at least $\frac{1}{2}$.

By Lemma 43.3, for a fixed choice of the random bits R, the set of inputs accepted by $\{\tilde{C}_n\}$ is in $\oplus\mathsf{P}$, therefore $\mathsf{PH} \subseteq \mathsf{RP}^{\oplus\mathsf{P}}$. ∎

Theorem 43.7 $\mathsf{RP}^{\oplus\mathsf{P}} \subseteq \mathsf{P}^{\#\mathsf{P}}$.

Proof: Let L be a language in $\mathsf{RP}^{\oplus\mathsf{P}}$, then, by Lemma 43.3, there exists a DC uniform family $\{C_n(R,x)\}$ of circuits with constant-depth, exponential size, and *four* types of gates (XOR, NOT, AND, and OR) such that the XOR gates have unbounded fan-in, the AND and the OR gates have polynomial fan-in, while the NOT gates are allowed to occur anywhere, where $|R|$ is polynomial in n and $|x| = n$, such that if $x \notin L$ then $C_{|x|}(R,x) = 0$ for every choice of R and if $x \in L$ then $C_{|x|}(R,x) = 1$ for at least half of the possible choices of R. Furthermore, using De Morgan inequality we can suppose, w.l.o.g., that each $\{C_n\}$ only uses XOR, AND and NOT gates.
For any n, let $C'_n(R,x)$ be the circuit obtained from $C_n(R,x)$ by replacing each XOR gate with a $(+)$ gate, each AND gate with a $(\times)$ gate and each NOT gate with a gate which increases the input by 1. Notice that, for any fixed choice of R and for every x, $C_n(R,x)$ outputs a 1 if and only if $C'_n(R,x)$ outputs an odd number. Therefore, if $x \notin L$ then $C'_{|x|}(R,x)$ is even for every choice of R and if $x \in L$ then $C'_{|x|}(R,x)$ is odd for at least half of the possible choices of R.

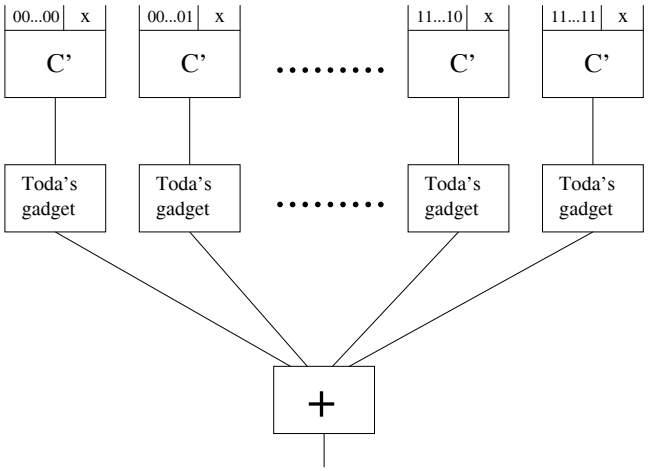


Figure 18: Construction of the type $(+)$ circuit deciding L.

In order to derandomize, we introduce Toda's gadget $T(x)$, which is a circuit with $(+)$ and $(\times)$ gates both with constant fan-in, with the property that $T(x) = 0 \pmod m$

if x is even, $T(x) = -1 \pmod m$ if x is odd, where $2^{|R|} \leq m \leq 2^{n^k}$ for some k. Assuming the existence (which we are going to prove later) of such a gadget, consider the circuit $C''_n(x)$ obtained as follows: for every choice of R, feed $T(C'_n(R, x))$ into a $(+)$ gate with fan-in $2^{|R|}$. The construction is better explained in Figure 18. Thus, if $x \notin L$ then $C''_n(x) = 0 \pmod m$, otherwise $C''_n(x) \neq 0 \pmod m$, which can be checked in polynomial time. By Lemma 43.4, $C''_n(x)$ can be computed by an $\mathsf{FP}^{\#\mathsf{P}}$ function, therefore L can be decided by a $\mathsf{P}^{\#\mathsf{P}}$ algorithm.
We now only need to show how to construct a Toda's gadget. Observe that, for every $x \in \mathbb{Z}$, for every $i \in \mathbb{N}$, $x = -1 \pmod{2^{2^i}}$ if and only $4x^3 + 3x^4 = -1 \pmod{2^{2^{i+1}}}$. This polynomial can be computed by a constant depth circuits S_0 with $(+)$ and $(\times)$ gates with constant fan-in. Toda's gadget is accomplished in $\lceil \log(|R|) \rceil$ stages, where the output of stage zero is $S_0(x)$, while the output of the $(i+1)$th stage is $S_{i+1}(x) = S_0(S_i(x))$ for $0 \leq i \leq \lfloor \log(|R|) \rfloor$. Given $m = 2^{2^{\lceil \log(|R|) \rceil}}$, the final result is given by $T(x) = S_{\lceil \log(|R|) \rceil}(x)$, and one can see that this gadget behaves as desired. ∎

References

[1] S. Toda, *On the computational power of PP and $\oplus$P*, Proc. 30th IEEE Symp. on the Fundations of computer Science, pp. 514-519, 1989.

[2] L. G. Valiant, V.V. Vazirani, *NP is as easy as detecting unique solutions*, Theor. Comp. Science, 47, pp. 85-93,1986.

CS 15-855 Computational Complexity Theory **October 18, 2000**

Lecture 16: Interactive Proofs

Lecturer: Rudich *Scribe: Kedar Dhamdhere / Editor: Kedar Dhamdhere*

Synopsis: Definition of interactive proofs. Arthur Merlin proofs (public coin interactive proofs). IP= PSPACE.

44 What are proofs?

Traditional proofs are static sequences of symbols to be perused by a verifier. NP consists of short traditional proofs.

45 Adding randomness to proofs

An interactive proof is a conversation between a prover and a verifier. The prover is computationally all powerful, whereas verifier has bounded resources (polynomial time). Prover tries to convince verifier of certain assertions. We say that a language is in IPif there is an interactive proof that "$x \in A$" $\forall x \in A$, where the length of the proof is polynomial in $|x|$.

Clearly, NP $\subseteq$ IP, since a traditional proof is just one round of conversation.

Do we add more power by adding interaction ?

Is IP more powerful than NP?

Example: Let ISO $= \{(G, H) \mid G \equiv H\}$. $G \equiv H$ if $\exists$ 1–1 onto map A: $V(G) \rightarrow V(H)$, such that $(u, v) \in E(G) \implies (A(u), A(v)) \in E(H)$. Let NONISO $= \{(G, H) \mid G \not\equiv H\}$.

We know that ISO $\in$ NP. Because we just have to guess an isomorphism between the graphs. However it is unlikely that ISO $\in$ NP-complete. Because it would imply that $PH = \Sigma_2$. On the other hand, we can ask whether NONISO $\in$ NP. This problem is still open. And it seems quite hard to tackle.

However, we can show that NONISO $\in$ IP.

Proof: Verifier picks randomly either G or H. She, then, generates a random permutation of that graphs and presents it to the prover. The prover tries to guess the graph from which it came from. If the two graphs G and H are isomorphic then random permutation of one will be indistinguishable from random permutation of another. But if the prover guesses correctly, then the verifier can be certain with probability 1/2. The prover can repeat the procedure several times to amplify the confidence. ■

45.1 Formal Definitions

An **Interactive Proof System** has two players: A polynomial time **verifier** given by the function, V: $\Sigma^* \times \Sigma^* \times \Sigma^* \rightarrow \Sigma^* \cup \{q_{accept}, q_{reject}\}$, where $V(x, r, S_1\#S_2\#S_3\#\ldots\#S_{i-1}) = S_i$. Here x is input and r denote the random bits known only to the verifier. And a **prover** with unlimited computational power, represented by the function: P: $\Sigma^* \times \Sigma^* \rightarrow \Sigma^*$, where $P(x, S_1\#S_2\#S_3\#\ldots\#S_{i-1}) = S_i$. Here P can be any function, not necessarily efficiently computable.

Notation: We will use $(V \leftrightarrow P)$ for the interactive proof system. We say that $(V \leftrightarrow P)(x, r)$ produces the *conversation* $S_1\#S_2\#\ldots\#S_l$, if

1. $S_l = q_{accept}$ or q_{reject}

2. $\forall$ odd i,
   $S_i = V(x, r, S_1\#S_2\#\ldots\#S_{i-1})$
   $\forall$ even i
   $S_i = P(x, S_1\#S_2\#\ldots\#S_{i-1}$
   $S_1 = V(x, r, \epsilon)$

The interactive proof system $(V \leftrightarrow P)(x, r)$ *accepts*, if the conversation it produces, ends in accept.

$Pr[(V \leftrightarrow P)(x)\ accepts] = Pr_r[(V \leftrightarrow P)(x, r)\ accepts]$

$Pr[V\ accepts\ x] = max_P\{Pr[(V \leftrightarrow P)(x)\ accepts]\}$

We say that, $A \in$ IP $\iff$ $\exists$ a probabilistic polynomial time verifier V such that $x \in A \implies Pr[V\ acceptsx] \geq \frac{2}{3}$

$x \notin A \implies Pr[V\ accepts x] \leq \frac{1}{3}$
And the conversation should be short, i.e. $|S_1 \# S_2 \# \ldots \# S_l| = \mathrm{O}(|x|^k)$

Example: NONISO $\in$ IP

45.2 IP $\subseteq$ PSPACE

A verifier accepting x is equivalent to a variant of alternation. And IP–machine is like polytime alternating machine except instead of AND, OR nodes, it has coin flip (AVG) and maximum (MAX) nodes.

Now $A \in$ IP $\iff \exists$ an IPmachine M such that,
$x \in A \implies Pr[M(x)\ accepts] \geq \frac{2}{3}$
$x \notin A \implies Pr[M(x)\ accepts] \leq \frac{1}{3}$

($\Longleftarrow$) The probabilistic polytime verifier V runs the machine M until she gets to MAX node. At MAX node verifier asks prover which path to choose. Prover, trying to maximize the probability of V accepting the proof, chooses a direction which maximizes the probability of reaching *accept*. Then verifier again runs machine M till she gets to another MAX node. At MAX node, again prover tells which way to go and so on.

($\Longrightarrow$) We want to construct the machine M. So we change the verifier V so that at points when she takes input from prover, the machine M uses a MAX node.

So we have proved that $A \in$ IP $\implies A$ has an IP–machine M. M can be simulated in PSPACE by depth-first search keeping track of the probability that the currently scanned node leads to an accept.

So we have NP $\subseteq$ IP $\subseteq$ PSPACE.

46 Arthur Merlin Proofs

In interactive proofs, the random bits r of verifier are not known to the prover. However a slightly different notion of proof arises if we assume that all random bits are publicly known. Thus, in Arthur-Merlin proofs, Arthur is polynomial time verifier and Merlin is computationally all powerful prover. We can even assume that Arthur's comments are all uniformly random. Formally

Definition 46.1 *Let $(V \leftrightarrow P)$ denote Arthur Merlin (AM) proof system. The conversation generated by $(V \leftrightarrow P)$ is a function of x (input) and $r_1, r_2, \ldots, r_l$ (random bits).*
$(V \leftrightarrow P)(x)(r_1, r_2, \ldots, r_l) =$

1. S_l = *accept or reject*
2. $S_1 = r_1$
3. *For odd i,* $S_i = r_{\lceil i/2 \rceil}$
   For even i, $S_i = P(x, S_1 \# S_2 \# \ldots \# S_{i-1})$

$A \in$ $\mathsf{AM}[l] \iff \exists$ *probabilistic polynomial time verifier V such that*
$x \in A \implies Pr[V \text{ accepts } x] \geq 2/3$
$x \notin A \implies Pr[V \text{ accepts } x] \leq 1/3$
such that $|S_1 \# S_2 \# \ldots \# S_l| = \mathrm{O}(|x|^k)$.

Finally, we define AM *as* $\cup_{l \geq 0}$ $\mathsf{AM}[l]$.

In previous section we saw, $\mathsf{NONISO} \in \mathsf{IP}$. So it is natural to ask: $\mathsf{NONISO} \in \mathsf{AM}$? The proof of $\mathsf{NONISO} \in \mathsf{IP}$ seems to rely on the fact that the random bits generated by the verifier are not known to the prover. Goldwasser and Sipser proved in 1985 that, in fact $\mathsf{NONISO} \in \mathsf{AM}$. The proof uses hashing technique to prove that there is an IP machine which accepts the input.

Theorem 46.1 *(Goldwasser–Sipser)* $\mathsf{NONISO} \in \mathsf{AM}$

Proof sketch: Let us consider the proof that $\mathsf{NONISO} \in \mathsf{IP}$. In that proof, verifier V generates random permutation of either graph G or graph H and presents it to prover P. If the graphs G and H are isomorphic then all random permutations generated by the verifier V are same. On the other hand, if $G \not\equiv H$, then the graphs generated by the verifier won't all be same. Thus number of possible conversations in both cases will be different. For example, if $G \equiv H$, then there will be $n!$ conversations (module some factors) and if $G \not\equiv H$, then there will be $2n!$ conversations. So to prove that $G \not\equiv H$, Merlin can prove to Arthur that number of conversations is strictly greater than $n!$, thus proving that $G \not\equiv H$. ∎

Proof: *With* loss of generality, assume that the graphs G and H are *rigid*, i.e. they have no non-trivial automorphisms. Let $ISO(G) = \{G' \mid G \equiv G'\}$ and $ISO(H) =$

$\{G' \mid H \equiv G'\}$. Since G and H are rigid, then $|ISO(G)| = |ISO(H)| = n!$. More generally,

$$|ISO(G)| = \frac{n!}{|AUTO(G)|}$$

where $AUTO(G) = \{\pi \mid \pi \colon V(G) \to V(G)\ takes\ G\ to\ G\}$. Now, since G and H are rigid,

$$\begin{aligned} |ISO(G) \cup ISO(H)| &= n! \quad if\ \ G \equiv H \\ &= 2n! \quad if\ \ G \not\equiv H \end{aligned}$$

Let $H \colon U \to T$ be an independent family of hash functions such that $|T| = 4n!$. Let $W = \{G' \mid G' \equiv H\ or\ G' \equiv G\}$. So $|W| = n!$ or $|W| = 2n!$. Arthur picks $h \in H, \alpha \in T$ at random. Let $\delta = Pr_{h,\alpha}[\exists G' \in W\ h(G') = \alpha]$. Then by Problem Set 3, # 4(a), we know that,

$$\frac{|W|}{|T|} - \frac{1}{2}\frac{|W|^2}{|T|^2} \leq \delta \leq \frac{|W|}{|T|}$$

So if $|W| = n!$, then $\delta \leq \frac{1}{4}$. And if $|W| = 2n!$, then $\delta \geq \frac{3}{8}$. Hence we can use amplification technique by using more than one hash functions h and hash values α picked independently.

Now if G and H are not rigid, then we construct graphs G_1 from G and H_1 from H by adding a vertex to each and connecting the new vertex to all of the old vertices. Let $M = G_1 \cup H_1$, i.e. M is made up of one copy of G_1 and one copy of H_1. And let $N = G_1 \cup G_1$, disjoint union. Now $|ISO(N)| = 2 \cdot |ISO(G)|^2$. If $G \not\equiv H$, then $|ISO(M)| = |ISO(G)| \times |ISO(H)|$. Further, we can show that $ISO(M) < ISO(N)$ by showing bounds on $AUTO(M)$ (left to the reader as an exercise). Thus we can use the technique described above to get the result $\mathsf{NONISO} \in AM[2]$ with exponentially small error for Arthur.

■

46.1 Graph isomorhpism revisited

We can use any of the four different methods of showing $\mathsf{BPP} \subseteq \mathsf{P}/poly$, to conclude $\mathsf{NONISO} \in \mathsf{NP}/poly$.

Coming back to graph isomorphism problem, if $\mathsf{ISO} \in \mathsf{NP}$-complete, then $\mathsf{NONISO} \in \mathsf{coNP}$ -complete. It would imply that $\mathsf{coNP} \subseteq \mathsf{NP}/poly$ and hence $\mathsf{PH} = \Pi_2$. Thus

if ISO is NP-complete then polynomial hierarchy collapses to level 2. This string evidence suggests that graph isomorphism is not NP-complete.

Theorem 46.2 *(Babai)* $\forall\ k > 2, \quad AM[k] = AM[2]$

Theorem 46.3 *(Goldwasser–Sipser)* $\forall\ k > 2, \quad IP[k] = AM[2]$

Thus, for a fixed k, we have $\mathsf{AM}[k] \subseteq \mathsf{NP}/poly$. Using similar ideas as in theorem 46.1, Goldwasser and Sipser proved that for polynomial interaction, $\mathsf{IP} = \mathsf{AM}[poly] = \mathsf{PSPACE}$.

CS 15-855 Computational Complexity Theory **October 25, 2000**

Lecture 17: IP = PSPACE

Lecturer: Rudich *Scribe: Bryan Clark / Editor: Joshua Dunfield*

Synopsis: In this lecture we will prove that IP = PSPACE. This will be done directly as opposed to through a PSPACE complete problem.

47 Introduction

The proof and major steps leading up to the proof of IP = PSPACE all happened in a fateful week in 1989. First, Fortnow, Karloff, Lund, and Nisan showed that Permanent $\in$ IP which leads to the direct corollary that $P^{\sharp P} \subset$ IP. Then Babai gave an alternative proof that $P^{\sharp P} \subset IP$ by utilizing $\sharp$3SAT. Finally, Shamir showed that IP = PSPACE using TQBF. In this lecture we do not utilize this method to prove that IP = PSPACE. Instead do it directly by examining space bounded Turing Machines. In so doing, we will revisit the ideas in Savitch's theorem.

48 Proof of IP = PSPACE

First we will demonstrate that IP $\subset$ PSPACE. To begin with, we know that IP = AM. Now, PSPACE can simulate Merlin. Whenever "Merlin" has to respond, PSPACE has the ability to examine each possible response and calculate (recursively) the probability that Arthur will accept. Then PSPACE assumes that Merlin will say the thing that will make Arthur most likely to accept. Therefore, we have concluded that IP $\subset$ PSPACE.

For the rest of the lecture we will prove that PSPACE $\subset$ IP. Intuitively, what this is demonstrating is that it is possible for someone to convince you that some particular move "x" is the optimal move in the game of Go. This will be done by transforming into a space where the optimal strategy against random moves is the same as the optimal strategy against optimal moves. Then this space will be transformed into a space where complete random moves are optimal.

We begin by creating a number of notational abbreviations and definitions that will allow us to encode the running of a Turing machine with a low degree polynomial PATH. First, let M be a Turing machine which runs in Space $O(n^k)$, where n is the length of the input. All configurations of the machine $M(x)$ can be described using a string of $l = O(n^r)$ symbols taken from a finite alphabet $\Gamma = \{1, 2, 3, \ldots, k\}$. Let Δ be the set of transition rules of M. Let

$$\alpha = a_1 a_2 \ldots a_l, \qquad \beta = b_1 b_2 \ldots b_l$$

be two configurations of M (where the a_i's and b_i's are from Γ).

We take $\alpha \underline{m}\!\!\rightarrow \beta$ to mean that configuration β can be reached from configuration α in no more than m steps.

We now describe the coding of the PATH polynomial. PATH_0 is defined thus:

$$\mathrm{PATH}_0(\alpha_1\alpha_2, \ldots, \alpha_l, \beta_1, \beta_2, \ldots, \beta_l) = \sum_{1 \le i \le l-3, \delta \in \Delta} \prod_{j \notin \{i, i+1, i+2\}} \mathrm{EQ}(a_j, b_j)\, \mathrm{LEGAL}_\delta \begin{bmatrix} a_i & a_{i+1} & a_{i+2} \\ b_i & b_{i+1} & b_{i+2} \end{bmatrix}$$

where $\mathrm{EQ}(a, b) = 1$ if $a = b$ and 0 if $a \neq b$, and

$$\mathrm{LEGAL}_\delta \begin{bmatrix} a & a' & a'' \\ b & b' & b'' \end{bmatrix} = 1 \text{ if } \delta \text{ changes } [a\ a'\ a''] \text{ to } [b\ b'\ b''], \text{ and 0 otherwise.}$$

Notice that EQ has degree $\le |\Gamma|$ and LEGAL has degree $\le |\Gamma|^6$. We will define

$$\mathrm{PATH}_i(\alpha, \beta) = \sum_{\gamma \in \Gamma^l} \mathrm{PATH}_{i-1}(\alpha, \gamma)\, \mathrm{PATH}_{i-1}(\gamma, \beta)$$

Now, notice the link between the Turing Machine and the polynomial. For all legal configurations α, β, $\mathrm{PATH}_0(\alpha, \beta) = 1$ if $\alpha \underline{1}\!\!\rightarrow \beta$ and 0 otherwise. For all legal configurations α, β $\mathrm{PATH}_i(\alpha, \beta) = 1$ if $\alpha \underline{2^i}\!\!\rightarrow \beta$ and 0 otherwise.

We can now examine how difficult each of these are to evaluate. Each variable in PATH_0 has degree $\le d$. Therefore given any $\alpha, \beta \in Z_p^l$ we can evaluate $\mathrm{PATH}_0(\alpha, \beta)$ in time polynomial in n. (A general $\alpha, \beta \in Z_p^l$ has no intuitive interpretation in terms of M. Nonetheless it will become useful to be able to evaluate PATH_0 for general $\alpha, \beta \in Z_p^l$.) Each variable within $\mathrm{PATH}_i(\alpha, \beta)$ has degree bounded by d. Each term has degree $\le 2ld = \mathrm{Poly}(n)$. For $i = \omega(\log n)$ it is no longer clear that we can evaluate $\mathrm{PATH}_i(\alpha, \beta)$ in time polynomial in n. Nonetheless, if the prover could convince you that $\mathrm{PATH}_{n^k}(C_{\mathrm{start}}, C_{\mathrm{accept}}) = 1 \pmod p$ then you would accept. (This is because

the prover would have convinced you that the Turing Machine would have accepted if the polynomial was 1 (mod p).)

Now, we are going to take a short hiatus from our proof to demonstrate the idea between many known interactive proofs. Essentially what happens is that we break an amazing proof into a bunch of little proofs. Consider Interactive Proofs which are from assertion to assertion.

If Q is true then R is True. If Q is false then R is falsehood (i.e. If the prover is lying at the beginning then the prover has no choice but to be lying at the end)

A slightly weaker requirement is to have the following:
If Q is true then R has to be true,
If Q is false then R is *probably* false.
We call this an ϵ link between Q and R.

Example: Working over Z_p, take 2 univariate polynomials A, B of degree $< \epsilon \cdot P$.

The prover claims these are the same. The verifier picks a random element R of the field Z_p and gives it to the prover. The prover asserts $A(R) = B(R)$. If the polynomials are equal then $A(R) = B(R)$. If the polynomials are not equal, then $A(R) = B(R)$ only in very few cases.

Theorem 48.1 *If $\aleph_1$ is an ϵ_1 link and $\aleph_2$ is an ϵ_2 link, then $\aleph_1 + \aleph_2$ is an $\epsilon_1 + \epsilon_2$ link.*

Returning to our main proof, the strategy is as follows. Develop a $\frac{poly(n)}{2^{poly(n)}} = \epsilon$ -link called the "halving protocol": H. The input assertions will have the form $\text{PATH}_i(\alpha, \beta) = \delta$. The output assertion will have the form $\text{PATH}_{i-1}(\alpha', \beta') = \gamma'$ The halving protocol will be broken into two parts. To begin with, in part 1 we will construct a bridge of Hybrid Polynomials that will create a polynomial that the verifier can check. In the second part, we will utilize Savitch's space saving trick encoded by low degree polynomials.

We will now proceed to constructing Hybrid Polynomials necessary for the halving protocol to function. The first of these will be $\text{PATH}_{n^t-1}(\alpha_1, \beta_1) = \delta_1$. The second of these will be $\text{PATH}_{n^t-2}(\alpha_2, \beta_2) = \delta_2$. This will continue down to $\text{PATH}_0(\alpha_{n^t}, \beta_{n^t}) = \delta_{n^t}$. Effectively the k'th hybrid takes in k variables functioning as constants and sums over all the ways of getting from α to γ and from γ to β. Essentially, these hybrids look partially like initial machines and partially like something else. Formally, for all

$\alpha, \beta \in Z_p^l$ and $i \in \mathbb{N}$ define $H_0, H_1, H_2, \ldots, H_l$ as follows:

$$H_k(C_1, C_2, \ldots, C_k) = \sum_{\gamma = C_1, C_2, \ldots, C_k, \gamma', \gamma'' \in \Gamma^{l-k}} \mathrm{PATH}_{i-1}(\alpha, \gamma)\, \mathrm{PATH}_{i-1}(\gamma, \beta)$$

There are a number of virtues associated with creating the hybrids in this manner. First, PATH_0 is something that can be checked in polynomial time. Moreover, each time a hybrid is created we only get ϵ additional error, so the probability of error is going to be at most $n^t \cdot \epsilon$. Also notice that:

1. $H_0 = \mathrm{PATH}_i(\alpha, \beta)$
2. $H_l(\gamma) = \mathrm{PATH}_{i-1}(\alpha, \gamma)\, \mathrm{PATH}_{i-1}(\gamma, \beta)$
3. $\sum_{j \in \gamma} H_k(\gamma_1, \ldots, \gamma_k - 1, j) \;=\; H_{k-1}(\gamma_1, \gamma_2, \ldots, \gamma_{k-1})$
4. For any fixed $\gamma_1, \gamma_2, \ldots, \gamma_{k-1}$, $H_k(\gamma_1, \gamma_2, \ldots, \gamma_{k-1}, x)$ is a degree $2d$ univariate polynomial that can be written down in space polynomial in n. (This is written down for you by the prover).

Now, we need a protocol to go from the k'th hybrid to the $(k+1)$'th hybrid. Let

$$\begin{aligned} S(x) &= \mathrm{PATH}_{i-1}((\gamma - \alpha)x + \alpha, (\beta - \gamma)x + \gamma) \\ S(0) &= \mathrm{PATH}_{i-1}(\alpha, \gamma) \\ S(1) &= \mathrm{PATH}_{i-1}(\gamma, \beta) \end{aligned}$$

Hence $S(x)$ is a univariate degree $\mathrm{poly}(n)$ polynomial, because for each variable in PATH_i we substitute an expression that is linear in x.

Now let us say that the prover wishes to prove

$$\mathrm{PATH}_{i-1}((\gamma - \alpha)x + \alpha, (\beta - \gamma)x + \gamma) = \delta$$

To accomplish this the prover sends $S(x)$. Then the verifier does a reality check on this polynomial by verifying that $S(0) \cdot S(1) = \delta$. (This is where we utilize Savitch's space saving theorem.) If the polynomial checks out, the verifier sends a random $c \in Z_p$. Then the verifier checks R: $\mathrm{PATH}_{i-1}(\alpha', \beta') = S(c)$ where $\alpha' = (\gamma - \alpha)c + \alpha$ and $\beta' = (\beta - \gamma)c + \gamma$. If the initial statement the prover was trying to establish was true, then R is true. If the initial statement was false and the polynomial was correct (which we have already done a reality check on earlier) then we know that

$$\Pr[\mathrm{PATH}_{i-1}(\alpha', \beta') < S(c)] \;<\; \frac{Poly(n)}{2^n}$$

Therefore, PSPACE $\subset$ AM $\subset$ IP.

49 Concluding remarks

For millennia man has sought a proof of the existence of God. We have shown a partial result: if a demi-god exists then he/she can prove this fact to you.

CS 15-855 Computational Complexity Theory **October 30, 2000**

Lecture 18: Approximation Algorithms

Lecturer: Steven Rudich *Scribe: Sanjit Seshia / Editor: Bartosz Przydatek*

Synopsis: Introduction to Approximation Algorithms. A few representative examples: Max-Cut, Maximum Satisfiability and the Knapsack problem. PTAS and FPTAS. Hardness of Approximation.

"We show why it's hard for Santa Claus to effectively steal elephants from a museum".

Steven Rudich

50 Introduction

As Papadimitriou mentions in his book [P94], although we group NP-complete problems together and have talked about them in the same breath for most of this course, the grouping of these problems into one class based on worst case complexity is about all that they share in common. The notion of *approximation* is one viewpoint from which they display what he calls a "healthy, confusing" diversity.

Approximation algorithms are defined (Williamson [W98]) as follows:

Definition 50.1 *An algorithm A is an α-approximation algorithm for an optimization problem Π if*
1. A runs in polynomial time
2. A always produces a solution which is within a factor of α of the value of the optimal solution.

The factor α is called the *performance ratio*. Often, for minimization problems, α is defined to be > 1 and for maximization problems $\alpha < 1$. We will define $\alpha > 1$ for both types of problems but just vary how we compute the ratio of the cost of the approximation over the cost of the optimal solution.

In this lecture we will limit ourselves to a discussion of "worst-case approximation" algorithms; i.e., algorithms that guarantee an approximation factor of α in the worst case.

50.1 Max Cut

Consider the Max-Cut problem. Informally, the problem is, given a graph, to find the cut with the maximum number of edges. An instance of the decision version of Max-Cut can be formally defined as follows:

Definition 50.2

$$\textbf{Max-Cut} \equiv \{< G, k > | G \textit{ has a cut which has at least } k \textit{ edges crossing it}\}$$

The set of all feasible solutions is just the set of all cuts. In the above definition, we measure the cost of a cut by the number of edges crossing it. But clearly, we can define other versions of Max-Cut if we change the cost function. Two such variants are finding *Minimum Weight* and *Maximum Weight* cuts in a graph G, given that each edge e in G is assigned a weight $w(e)$.

Although these two variants seemingly arise from the same problem, their worst-case complexities are very different. The minimum-weight cut problem is known to be in P(solved using flow techniques), whereas the maximum-weight problem is known to be NP-complete. This latter problem is the subject of this section, and we will hereafter refer to it as "Max-Cut".

Max-Cut is NP-complete. Given a graph $G = (V, E)$, we can non-deterministically choose a cut, and check if it has weight $\geq k$ in $O(|E|)$ time (where E is the edge set).

We can get a 2-approximation algorithm for Max-Cut. Consider the following simple randomized algorithm. Pick a subset $S \subseteq V$ of vertices and consider these to be on one side of the cut, with the remaining vertices on the other side. If we pick the vertices in a pairwise independent manner, we can show that the expected number of edges crossing the cut is $|E|/2$. To get a polynomial time algorithm from this, we can derandomize this algorithm by the bit-amplification technique we've seen earlier in this course.

Given one approximation algorithm for a problem like Max-Cut, a natural question to ask is: What is the smallest ϵ such that an approximation algorithm with ratio $1 + \epsilon$ can be found?

This question has been somewhat well studied in the case of Max-Cut. Goemans and Williamson [GW95] found a 1.138 algorithm that uses Semi-definite programming. Recently, it has also been shown that finding a 1.0624 algorithm is not possible unless NP=ZPP. These upper and lower bound results give us some feeling for how hard the approximation problem is.

These results also bring up the point that approximation algorithms are indeed very diverse with a mind-boggling range of approximation ratios. How does one go about categorizing such a diverse set of results? This lecture describes attempts to address this problem.

51 Maximum Satisfiability

As in the case of almost all complexity classes, we can define an appropriate version of SAT for approximation algorithms. The discussion of this problem also enables us to demonstrate the use of a common randomized approximation technique called the "Conditional-Expectation" method.

We define two problems related to satisfiability.

Definition 51.1 k-MaxSat: *Given a formula ϕ that is a conjunction of disjunctive k-clauses, how many of them can we satisfy by an assignment?*

Definition 51.2 k-MaxGSat: *Given a sequence $(\phi_1, \phi_2, \ldots, \phi_m)$ of boolean formulas, each of which depends on $\leq k$ variables, how many of them can be satisfied by a single assignment to the variables?*

As you might have guessed, the "G" in the name of the second problem stands for "global". It might also stand for "general", since this problem is more general than k-MaxSat. More precisely, an instance of k-MaxSat can be mapped to an instance of k-MaxGSat by merely considering each clause of k-MaxSat as a separate boolean formula. Therefore we will focus our approximation efforts on k-MaxGSat.

Here is a deterministic approximation algorithm for k-MaxGSat that apparently arises from a starting point that seems to indicate randomization. Let $\phi_1, \phi_2, \ldots, \phi_m$ be the m boolean formulas each of which depends on $\leq k$ of the variables $x_1, x_2, \ldots, x_n$. Define a random variable Y_i as follows

$$Y_i = \begin{cases} 1 & \text{, if } \phi_i \text{ is True,} \\ 0 & \text{, otherwise.} \end{cases}$$

Now, the expectation $E[Y_i]$ is just the probability that the formula ϕ_i is satisfied by a random assignment to the variables $x_i, \ldots, x_n$. Define $\Delta \equiv \sum_i E[Y_i]$; thus Δ is the expected number of ϕ_i's to be satisfied.

Now, notice that since each ϕ_i depends on $\leq k$ variables, we can compute Δ by the brute force approach, computing each $E[Y_i]$ (in 2^k-time, which is $O(1)$ since k is a constant).

Here is the *conditional expectation* trick. Pick the first variable, say x_1. Set $x_1 = 0$ and compute the new expectation Δ_0 with this constraint. Similarly, calculate Δ_1. Since $\Delta = \frac{1}{2}(\Delta_0 + \Delta_1)$, one of Δ_1 and Δ_0 must be larger than Δ. Let $\Delta_i = max(\Delta_1, \Delta_0)$. Fix $x_1 = i$. Continue this procedure until all x_i's have been assigned.

Since at every step we maximize the "expectation-to-go", we will end up with an assignment that satisfies more than Δ of the formulas $\phi_1, \phi_2, \ldots, \phi_m$.

Throw out any formulas that are not satisfiable. Let m' denote the number of the remaining formulas. Define

$$\delta = min_{1 \leq i \leq m'} E[Y_i]$$

Clearly, $\Delta \geq \delta m'$. Furthermore, the optimal value $opt \leq m'$. Thus, we get

$$\frac{\Delta}{opt} \geq \delta \quad \text{or} \quad \frac{opt}{1/\delta} \leq \Delta \ .$$

We can conclude that our algorithm gives a $1/\delta$ approximation to k-MaxGSat.

We can use this $1/\delta$ approximation to k-MaxGSat to give an approximation for k-MaxSat. Consider any clause in the k-MaxSat instance. If it has exactly k variables, the probability that it is satisfiable is exactly $1 - 2^{-k}$. We can combine this with the preceding discussion to get the following theorem.

Theorem 51.1 *There is a $(1-2^{-k})^{-1}$ approximation for k-MaxSat when each clause has exactly k variables.*

In fact, it is known that this is the best approximation possible for $k \geq 3$, unless P=NP (due to Håstad). In particular, an $\frac{8}{7}$th approximation algorithm is the best we can do for 3-MaxSat.

Another interesting point is that even though 2-SAT is known to be in P, 2-MaxSat is not! Moreover, 2-MaxSat has a 1.0741 approximation, but no 1.0476 approximation.

52 The Knapsack Problem, PTAS and FPTAS

This section is mandatory reading for prospective department store thieves and Santa Clauses with dark sides to their personality :-) .

Recall the well known Knapsack problem:

Definition 52.1 Knapsack: *Consider having a knapsack of weight capacity W. We have a set of objects of weights $w_1, w_2, \ldots, w_n$ with corresponding values $v_1, v_2, \ldots, v_n$. We wish to find a subset S of these n objects such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i$ is maximized.*

A natural approach to this problem is to use Dynamic Programming.

Define $W(i, v)$ to be the minimum weight attainable by selecting among the first i items so that their total value is exactly v. We have the following recurrence relations

$$\begin{aligned} W(0, v) &= \infty \ \forall v \\ W(i+1, v) &= MIN\{W(i, v), [W(i, v - v_{i+1}) + w_{i+1}]\} \end{aligned}$$

By building a table for W we can solve the Knapsack problem in time polynomial in n and in the number of possible values of the v_i's. This is not very good because the value of the v_i's can be potentially exponential in n.

Instead, we work with values $v'_1, v'_2, \ldots, v'_n$ where v'_i is obtained by truncating v_i to $k \log n$ bits. If we do this, each v_i can only have $O(n^k)$ possible values. We will show how this truncation strategy can yield a $1 + \epsilon$ approximation for any $\epsilon > 0$.

Without loss of generality, assume that $w_i \leq W$ for all i (i.e., toss out the useless items that you can't put in anyway). Let $v = \max_i v_i$, and let l be the the number of bits needed to express v precisely. Pick k such that $\frac{2n}{n^k} < \epsilon$. Use this k to truncate each v_i to a value v'_i of precision at most $k \log n$ bits (i.e., set all but the $k \log n$ most significant bits of v_i to 0). We run our dynamic programming algorithm with the v'_is instead of the v_is.

Now, notice that any feasible solution obtained using the v_is that gives a total value of V will have a feasible counterpart solution using the v_is with total value V' such that $V' \geq V - n2^{l - k \log n + 1}$. (We get this by counting the value lost due to missing precision).

The loss in value is $L = n2^{l - k \log n + 1}$, and we can check that $\frac{L}{2^l} = \frac{2n}{n^k} < \epsilon$.

Therefore, our truncation-based algorithm returns a value within $(1 - \epsilon)$ of the optimal. In other words, we have a $1 + \epsilon$ approximation. Moreover the algorithm runs in time $O(n^3/\epsilon)$.

Notice that the approximation algorithm includes the approximation parameter in its running time complexity. This fits in with our intuition as well - the closer the

approximation, the longer we expect the algorithm to take. In fact, we have a formal way of characterizing algorithms that exhibit this tradeoff and this is embodied in the following definitions.

Definition 52.2 PTAS: *A* polynomial-time approximation scheme *(PTAS) is an algorithm A for an optimization problem Π where on input (x, ϵ), A returns a solution which is a $1+\epsilon$ approximation of the optimal in time $P_\epsilon(|x|)$, where $P_\epsilon(\cdot)$ is a polynomial dependent on ϵ.*

Definition 52.3 FPTAS: *A* fully polynomial-time approximation scheme *(FPTAS) is a PTAS that runs in time polynomial in $|x|$ and $1/\epsilon$.*

Thus we see that Knapsack has an FPTAS.

On the other hand, we see that Max-Cut and 3-MaxSat cannot be approximated to within ratios of 1.0624 and 8/7 respectively. The problem of finding the maximum size clique cannot even be approximated to within $n^{1-\epsilon}$ for any $\epsilon > 0$! This clearly indicates there is a hierarchy of hardness in approximation. We will take this up in the next section.

53 Hardness of Approximation

Much of the material in this section is based on an excellent survey by Arora and Lund [AL96].

Just as 3-SAT is the "ancestor" of all NP-complete problems, 3-MaxSat will be the starting point of the theory of hardness of approximations. The following theorem by Arora et al. [ALM+92] gives a way of defining hardness of 3-MaxSat by quantifying the amount of unsatisfiability in it.

Theorem 53.1 *There exists $\epsilon > 0$ and a polynomial time reduction R from SAT to 3-MaxSat such that*

$\forall I \in SAT$, $\quad R(I)$ *is satisfiable.*

$\forall I \notin SAT$, $\quad R(I)$ *will never have more than* $\dfrac{1}{1+\epsilon}$ *fraction of its clauses satisfiable*

This result leads to the corollary that $\exists \epsilon > 0$ such that 3-MaxSat cannot have a $1+\epsilon$ approximation unless P=NP.

In their survey, Arora and Lund give a classification of approximation hardness results. They outline four broad classes as shown in the table below:

Class	Approximation Hardness factor	Representative problems
I (Max-SNP)	$1+\epsilon$	3-MaxSat, Vertex-Cover Max-Cut, Metric TSP
II	$O(\log n)$	Set Cover, Hitting set
III	$2^{\log^{1-\delta} n}$, $\forall \delta > 0$	Longest Path Nearest Lattice vector
IV	n^{ϵ}	Clique, Chromatic #

53.1 Gap Preserving Reductions

Proving the NP-hardness of approximating a problem Π involves giving a special kind of reduction to Π from an NP-complete problem; the reduction must produce a gap in the value of the optimum for Π. Such a reduction is said to be a *gap-producing* reduction [AL96]. For example, suppose we want to prove the NP-hardness of approximating a minimization problem Π to within a factor g. We can do this by constructing a gap-producing reduction from SAT to Π that maps satisfiable formulae to instances whose solution is of value at most c (for some c), and unsatisfiable formulae to instances with solutions of value at least gc (see Figure 19).

To see how this works, assume Π has a polynomial time approximation algorithm A_Π that guaranteed a factor $g^* < g$. Consider an arbitrary satisfiable formula; this will map to an instance whose optimum is $\leq c$. Therefore, on this instance of Π, A_Π would return an answer x, where $x \leq g^* c < gc$. Now, we know that any unsatisfiable formula will map to an instance of Π with optimum greater than gc; furthermore, A_Π can only return an answer greater than gc for this instance. This means that we can correctly conclude in polynomial time that the formula is satisfiable, and this works for an arbitrary satisfiable formula. In other words, we can solve SAT in polynomial time, which in turn implies that P = NP! Therefore, assuming P $\neq$ NP, Π is hard to approximate to within a factor of g.

The above method of proving the hardness of approximation of Π uses a reduction from a known NP-complete problem. It is sometimes desirable to have a mechanism

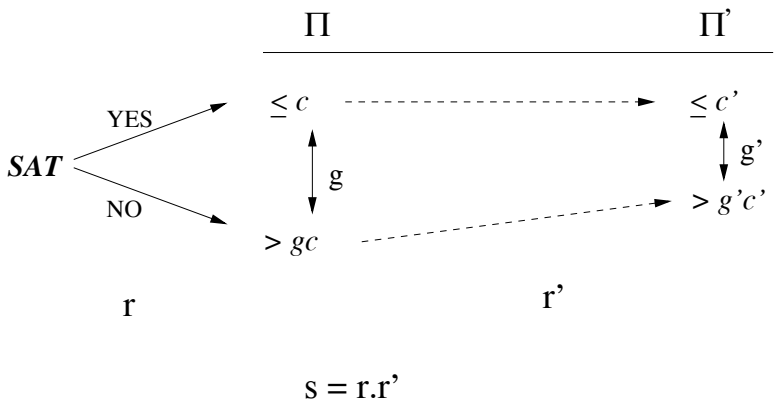


Figure 19: Gap Preserving and Producing Reductions

that allows us to conclude the approximation hardness of a problem Π', given the approximation hardness of a problem Π, instead of having to find a mapping from an NP-complete decision problem to Π'. This is achieved using the notion of a *gap-preserving* reduction.

Definition 53.1 Gap-Preserving Reduction: *[AL96] Let* Π *and* Π' *be two minimization problems. A* gap-preserving reduction *from* Π *to* Π' *with parameters* (c, g) *and* (c', g')*,* $(g, g' \geq 1)$*, is a polynomial time algorithm* f*, which for each instance* I *of* Π *produces an instance* $I' = f(I)$ *of* Π'*. The optima of* I *and* I'*, denoted* O_I *and* $O_{I'}$ *respectively, satisfy the following two properties:*

$$O_I \leq c \;\Rightarrow\; O_{I'} \leq c' \tag{1}$$

$$O_I > gc \;\Rightarrow\; O_{I'} > g'c' \tag{2}$$

Suppose we have a gap-producing reduction r from SAT to Π. The existence of a gap-preserving reduction r' from Π to Π' proves the existence of a gap-producing reduction s from SAT to Π'; s is the composition of r and r'. In other words, the reduction s shows that achieving a performance ratio of g' for Π' is NP-hard. Figure 19 clarifies this concept.

Note however, that the gap-preserving property of a reduction does not suffice for exhibiting the ease of approximating Π given that Π' is easy to approximate.

Also, the term "gap-preserving" is somewhat misleading, because the gap is really not preserved. "Gap-accounting" might be a somewhat more appropriate term.

53.2 Max-SNP problems

Definition 53.2 Max-SNP: *An optimization(maximization) problem* Π *with instance* I *is in Max-SNP if* $\exists k > 0$ *and a polynomial time algorithm* A *such that* $A(I)$ *outputs a list of boolean formulas where each of them depends on at most* k *variables, and the optimum of* I *is the maximum number of simultaneously satisfiable formulas in* $A(I)$.

The above definition just says that we should be able to map Π in polynomial time to k-MaxGSat for some constant k.

A problem is Max-SNP-hard if it has a gap-preserving reduction to any problem in Max-SNP. We know that both 3-MaxSat and Max-Cut are in Max-SNP.

Notice that if we could give a PTAS for any problem in Max-SNP, then we could give one for 3-MaxSat, but we know that this is not possible (assuming $P \neq NP$).

Papadimitriou and Yannakakis have defined a different kind of reduction called L-reduction, and we can show that 3-MaxSat is Max-SNP complete under L-reductions. Here is a sample of problems are Max-SNP hard under L-reductions: Max-Cut, 2-MaxSat, Max-Independent-Set, Shortest-Superstring, Chromatic Number.

If any one of these problems has a PTAS, then all of them do.

References

[P94] C. Papadimitriou, *Computational Complexity*, Addison Wesley, 1994.

[GW95] M. Goemans and D. Williamson, *0.878 Approximation Algorithms for MAX-CUT and MAX-2SAT*, In *Proceedings of STOC'95*, 1995.

[W98] D. Williamson, *Lecture Notes on Approximation Algorithms*, *IBM Research Report RC 21409*, Fall 1998.

[ALM+92] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, *Proof verification and intractability of approximation problems.* In *Proc. 33rd FOCS*, pp. 13-22, 1992.

[AL96] S. Arora and C. Lund, *Hardness of Approximations.* In *Approximation Algorithms for NP-hard Problems*, Dorit Hochbaum, *Ed.*, PWS Publishing, 1996.

CS 15-855 Computational Complexity Theory **Nov. 1, 2000**

Lecture 19: PCP — Holographic Proofs

Lecturer: Rudich *Scribe: Bartosz Przydatek / Editor: Sanjit Seshia*

Synopsis: We introduce the notion of *Probabilistically Checkable Proofs* and define the associated complexity classes $\mathsf{PCP}(r(n), q(n))$. We present also an overview of one of the most important results in Theoretical Computer Science in the last decade: a proof of the PCP Theorem, which states that $\mathsf{NP} = \mathsf{PCP}(\log n, 1)$.

"Today's lecture is carefully designed around today's homework."
Steven Rudich

54 Introduction

So far we have worked with the standard NP proofs. These proofs are very brittle: if there is a tiny error somewhere in the proof the entire proof is wrong! In this lecture we will learn a new characteriztion of NP, which is more robust. Also, this characterization is quite surprising and of great importance: it has a major impact on the study of combinatorial optimization. In particular, it provides powerful techniques for proving lower bounds on the performance of approximiation algorithms.

The new characterization of NP is based on the notion of *Probabilistically Checkable Proofs* (PCP), which is defined below.

A *verifier* is a polynomial-time probabilistic Turing Machine, with an input tape, work tape, a source of random bits and a and a random acces to a (read-only) *proof string* Π. We say that a verifier is $(r(n), q(n))$-*restricted*, if on each input of size n it uses $O r(n)$ random bits and queries $O q(n)$ bits of the proof. We assume, that the verifier works *non-adaptively*, i.e. the positions on which the verifier queries the proof are chosen in advance, depending only on the random bits used by the verifier.

Definition 54.1 *A language L is in $PCP(r(n), q(n))$ if there exists a polynomial time $(r(n), q(n))$-restricted verifier V such that*

$$x \in L \quad \Rightarrow \quad \exists \text{ a proof } \Pi_x \text{ s.t. } Pr_r\left[V(x,r) = yes\right] = 1 \qquad \text{(completeness)}$$

$$x \notin L \quad \Rightarrow \quad \forall \text{ proof } \Pi \;\; Pr_r\left[V(x,r) = yes\right] \leq \frac{1}{2}\,. \qquad \text{(soundness)}$$

55 The PCP Theorem

Theorem 55.1 (The PCP Theorem)

$$NP = PCP(\log n, 1).$$

One inclusion, $\mathsf{PCP}(\log n, 1) \subset \mathsf{NP}$ is pretty easy (cf. Problem Set 7). The other one, $\mathsf{NP} \subset \mathsf{PCP}(\log n, 1)$, is more involved and is the topic of this lecture. More explicityly, it says that for any set $A \in \mathsf{NP}$ there exists a polynomial time verifier V with a random access to a "proof string" Π, tht uses $O\log n$ random bits and queries Π at $O1$ places, such that

$$x \in A \quad \Rightarrow \quad \exists \text{ a proof } \Pi_x \text{ s.t. } V^{\Pi}(x) \text{ always accepts}$$

$$x \notin A \quad \Rightarrow \quad \forall \text{ proof } \Pi \;\; \mathsf{Pr}_r\left[V^{\Pi}(x,r) \text{ accepts}\right] \leq \frac{1}{2}\,.$$

Remark: Some stronger versions of the PCP Theorem are known. In particular, Håstad's "Ultimate PCP Theorem" shows that there exists a verifier, which uses $O\log n$ random bits, queries only 3 (!) bits of the proof (instead of $O1$ bits) and still achieves the required soundness, with completeness weakened by only an arbitrarly small $\epsilon > 0$.

56 PCP and Hardness of Approximation

In the previous lecture we have seen a theorem (without a proof), that there exists a gap preserving reduction from SAT to MAX$-3-$SAT [ALM+92]. Now, with a help of the PCP Theorem, we can actually prove it, as the following lemma shows. Since we know also, that MAX$-3-$SAT is MAX-SNP-complete and that (MAX$-3-$SAT has a PTAS $\Leftrightarrow \mathsf{P} = \mathsf{NP}$), we get immediately that

$$\text{all of MAX-SNP has a PTAS} \;\Leftrightarrow \mathsf{P} = \mathsf{NP}\,.$$

Lemma 56.1 *If* $\mathsf{NP} = \mathsf{PCP}(\log n, 1)$*, then there exist* $\epsilon > 0$ *and a polynomial time reduction* R *from* SAT *to* 3CNF *such that*

$$\begin{aligned} \varphi \in \text{SAT} \quad &\Rightarrow \quad R(\varphi) \text{ is satisfiable} \\ \varphi \notin \text{SAT} \quad &\Rightarrow \quad \text{no assignment satisfies more than} \\ &\qquad a \tfrac{1}{\epsilon+1} \text{ fraction of clauses of } R(\varphi) \, . \end{aligned}$$

Proof: Let φ be a Boolean formula and $n = |\varphi|$. Since SAT $\in \mathsf{NP}$ and $\mathsf{NP} = \mathsf{PCP}(\log n, 1)$, there exists a $\mathsf{PCP}(\log n, 1)$ verifier V which takes (φ, Π) as input (where $|\Pi|$ is polynomial in n) and decides if Π is a proof that $\varphi \in$ SAT.

V picks $O\log n$ random bits r and checks the proof at $\leq k$ (constant) of places. Let $Y_{i_1}, \ldots, Y_{i_k}$ denote the k bits read by V, and let $f_r(Y_{i_1}, \ldots, Y_{i_k})$ be the function that describes the output behavior of V for given the random string r. Since $|r| = O\log n$, there are polynomialy many such f's, say $f_1, \ldots f_{n^c}$, for some constant c. Each f_r can be expressed as a k-CNF formula with at most 2^k clauses.

Let Φ be the conjunction of all the clauses over all f's. Formula Φ has at most $2^k n^c$ clauses of size k, so $|\Phi|$ is polynomial in n.

If $\varphi \in$ SAT, then there exists a proof Π that satisfies *all* f's, and hence satifsfies Φ. If $\varphi \notin$ SAT, then for every proof Π only a constant fraction of f's return true, hence only a constant fraction $(1 - \frac{1}{2^{k+1}})$ of clauses in Φ can be satisfied simultaneously.

Finally, we transform Φ into a 3CNF formula by replacing each k-clause $(a_1 \vee a_2 \vee \cdots \vee a_k)$ by $(a_1 \vee a_2 \vee z_1)(a_3 \vee \bar{z}_1 \vee z_2) \ldots (a_{k-1} \vee a_k \vee \bar{z}_{k-3})$. Clearly, this transfomation preserves the constant fraction property (with a different constant). ∎

57 Proof of the PCP Theorem: An Overview

We construct a $\mathsf{PCP}(\log n, 1)$ verifier for 3SAT by *composing* two "less efficient" verifiers:

- a verifier V_A, which shows that 3SAT $\in \mathsf{PCP}(n^3, 1)$ (cf. Problem Set 7)
- a verifier V_B, which shows that 3SAT $\in \mathsf{PCP}(\log n, polylogn)$ (this construction comes in a later lecture).

For the compostion to work out we will require both V_A and V_B to be in *normal form*, which is a bit mysterious and technical, but is designed to have two important properties:

1. Bits that must always be read together will be grouped and viewed as a single symbol in a larger alphabet. The size of the alphabet will be allowed to grow as a function of the input (theorem) size.
2. (theorem, proof)-pairs will be written in such a way, that the verifier can check their consistency by reading O1 symbols, i.e. there is no need to read the theorem to check the proof! The magic used to accomplish this are the error correcting codes.

We can state this more formally:

Definition 57.1 *A* 3SAT *verifier is in* normal form *if*

1. *For an input formula φ with n variables the verifier expects the proof Π to be a string over an alphabet Σ, whose size is a function of n. A query of a verifier involves reading a symbol of Σ, not just a single bit.*
2. *The verifier has a subroutine, which can efficiently check proofs of a very special form. More precisely, for any given positive integer p the subroutine behaves as follows:*
   (a) *It defines an error correcting code $\mathcal{C}_p$ over Σ, with $\delta_{\min} > 0.3$ and a 1-1 encoding function $\sigma : \{0,1\}^{n/p} \to \mathcal{C}_p$.*
   (b) *It expects the proof string S to have the form*

$$S = \sigma(a_1) \circ \sigma(a_2) \circ \ldots \circ \sigma(a_p) \circ \Pi \ ,$$

   where $a_1 \circ \ldots \circ a_p$ is an assignment satisfying φ and Π is a proof that this is indeed the case.

   Formally, we say that the subroutine checks assignments split into p parts *if on the proofs of the form $S = z_1 \circ \ldots \circ z_p \circ \Pi$, $S \in \Sigma^*$, it has the following behaviour:*

   - *If $z_1, \ldots, z_p$ are codewords, such that $\sigma^{-1}(z_1) \circ \ldots \circ \sigma^{-1}(z_p)$ is a satisfying assignment then there exists Π such that*

$$\mathsf{Pr}[\textit{subroutine accepts } z_1 \circ \ldots \circ z_p \circ \Pi] = 1 \ .$$

- *If* $\exists\, i : 1 \le i \le p$ *such that* z_i *is not* $\delta_{\min}/3$*-close to a codeword, then for all* Π

$$\mathsf{Pr}[\textit{subroutine accepts } z_1 \circ \ldots \circ z_p \circ \Pi] < \frac{1}{2}\ .$$

- *If* $\forall\, i : 1 \le i \le p$ z_i *is* $\delta_{\min}/3$*-close to a codeword* $N(z_i)$*, but* $\sigma^{-1}(N(z_1)) \circ \ldots \circ \sigma^{-1}(N(z_p))$ is not *a satisfying assignment, then*

$$\mathsf{Pr}[\textit{subroutine accepts } z_1 \circ \ldots \circ z_p \circ \Pi] < \frac{1}{2}\ .$$

Definition 57.2 *A verifier in normal form is* $(r(n), q(n), t(n))$-restricted *if on inputs of size* n

- *it uses* $\mathrm{O}r(n)$ *random bits,*
- *it reads* $\mathrm{O}q(n)$ *symbols from the proof* Π*,*
- *its* decision time[6] *is* $poly(t(n))$*, and*
- *its special subroutine for checking assignments split into* p *parts reads* $\mathrm{O}p \cdot q(n)$ *symbols.*

Remark: In the above definition verifier queries $\mathrm{O}q(n)$ *symbols*, i.e. $\mathrm{O}q(n)\log|\Sigma|$ bits. There is no need to put an explicit bound on $|\Sigma|$, since it is already implicitly bounded: the decision time includes the time needed to process $\mathrm{O}q(n)\log|\Sigma|$ bits of information, so $\mathrm{O}q(n)\log|\Sigma| \le poly(t(n))$.

Indeed, the definition of a normal form verifier looks pretty esoteric. Let's have a look at an example, i.e. let's construct a *normal form* verifier V_A from the verifier found in the homework (Problem Set 7), which showed that 3SAT $\in \mathsf{PCP}(n^3, 1)$: The "homework"-verifier, say V_H, works as follows: the prover takes a satisfying assignment $a_1 \ldots a_n$ and encodes it holographically using three "boxes":

$$\begin{aligned} B_1(z_1, \ldots, z_n) &= \sum a_i z_i \\ B_2(z_{11}, \ldots, z_{nn}) &= \sum (a_i a_j) z_{ij} \\ B_3(z_{111}, \ldots, z_{nnn}) &= \sum (a_i a_j a_k) z_{ijk}\ . \end{aligned}$$

[6]Decision time is the time the verifier needs after reading all the symbols to make its final decision.

Verifier checks that each B_i is close to a linear function, and that $B_2 = B_1 \otimes B_1$ and $B_3 = B_1 \otimes B_1 \otimes B_1$. Finally, V_H checks whether the assignment encoded in B_1, B_2 and B_3 is satisfying, by evaluating a polynomial derived from the φ.

The normal form verifier V_A works in a similar way. Suppose that $n = p \cdot m$, and consider the assingment $a_1 \dots a_n$ split into p parts: $[a_1 \dots a_m], [a_{m+1} \dots a_{2m}], \dots$, $[a_{(p-1)m+1} \dots a_{pm}]$. Let

$$\begin{aligned} f_1(z_1, \dots, z_m) &= \sum_{i=1}^{m} a_i z_i \\ f_2(z_{m+1}, \dots, z_{2m}) &= \sum_{i=m+1}^{2m} a_i z_i \\ &\vdots \\ f_p(z_{(p-1)m+1}, \dots, z_{pm}) &= \sum_{i=(p-1)m+1}^{pm} a_i z_i \ . \end{aligned}$$

A normal form verifier expects a proof string of the form

$$\underbrace{f_1 \circ f_2 \circ \ldots \circ f_p}_{\substack{\text{encodes an} \\ \text{assignment}}} \circ \underbrace{B_1 \circ B_2 \circ B_3}_{\substack{\text{proves, that} \\ \text{the encoded} \\ \text{assignment} \\ \text{satisfies } \varphi}} .$$

Verifier V_A performs the same checks as V_H (i.e. V_A checks the linearity of B_1, B_2, B_3, whether $B_2 = B_1 \otimes B_1$ and $B_3 = B_1 \otimes B_1 \otimes B_1$, and whether the encoded assignment is indeed satisfying). Additionally, V_A verifies that $\forall i$ f_i is close to a linear function $\hat{f}_i$, and every point x for which $B_1(x)$ is queried, V_A checks whether $B_1(x) = \sum_{j=1}^{p} \hat{f}_j(x)$.

It follows that V_A is a normal form $(n^3, 1, 1)$-restricted verifier for 3SAT, because

- If $f_1 \circ f_2 \circ \ldots \circ f_p \circ B_1 \circ B_2 \circ B_3$ is correct, then verifier V_A always accepts.
- If there is f_i which is not close to a linear $\hat{f}_i$, the linearity test will catch it and V_A will reject.
- If all f_i's are close to linear functions $\hat{f}_i$, whose coefficients do not make a satisfying assignment, then either B_1, B_2, B_3 are not valid or $B_1(i) = \sum_{j=1}^{p} \hat{f}_j(x)$ will not always be true, and V_A will reject.

Remark: Verifier V_A reads only $O1$ bits of $f_1 \circ \ldots \circ f_p$, so it does not even know what it just verified.

Lemma 57.1 (Composition Lemma) *Let V_1 and V_2 be normal form verifiers for* 3SAT *that are $(R(n), Q(n), T(n))$-restricted and $(r(n), q(n), t(n))$-restricted, respectively. Then there exists a normal form verifier for* 3SAT *that is $(R(n) + r'(n), Q(n) \cdot q'(n), t'(n))$-restricted, where $r'(n) = r(poly(T(n)))$, $q'(n) = q(poly(T(n)))$, and $t'(n) = t(poly(T(n)))$.*

Remark: When we use this lemma, both $Q(n)$ and $q(n)$ are $O1$, hece the three verifiers have bounds $(R(n), 1, T(n))$, $(r(n), 1, t(n))$ and $(R(n) + r'(n), 1, t'(n))$, respectively. Furthermore, if $t(n)$ is a slowly growing function, say $\log n$, then the decision time for the new verifier is $O\log(T(n))$, an exponential improvement over $T(n)$!

How do we get the PCP Theorem from the Composition Lemma? As mentioned earlier, we use two verifiers:

- $(n^3, 1, 1)$-restricted verifier V_A (cf. the construction described above),
- $(\log n, 1, \log n)$-restricted verifier V_B (cf. notes of a later lecture).

First we compose V_B with itself (i.e. apply the Composition Lemma with $V_1 = V_2 = V_B$), to get a $(\log n, 1, \log\log n)$-restricted verifier V'. Finally, we compose V' with V_A ($V_1 = V'$, $V_2 = V_A$) to get $(\log n, 1, 1)$-restricted verifier V, which completes the proof of the PCP Theorem.

Proof sketch: (of *Composition Lemma*, cf. [A94, Chapter 3]) Once we fix the random string r of the verifier V_1, its decision depends upon $p = OQ(n)$ symbols of the proof string, and is computed in $poly(T(n))$ time. By the Cook-Levin Theorem there exists a 3CNF formula Φ of size $poly(T(n))$, which is satisfiable if and only if $V_1(r)$ accepts. Instead of having $V_1(r)$ actually query the proof string and decide, we modify $V_1(r)$ to use the verifier V_2 to verify that $V_1(r)$ would accept if it only took the trouble to read p symbols of the proof. In other words, V_2 checks whether the symbols of the proof V_1 would have read form a satisfying assignment for Φ. Doing this involves using V_2's ability to check split assignments and requires that relevant portions of the proof-string be present in an encoded form (using V_2's encoding). ∎

References

[**Note:**] All the references listed below can be found also on the Sanjeev Arora's list of publications at `http://www.cs.princeton.edu/~arora/publist.html`.

[ALM+92] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, *Proof verification and intractability of approximation problems*. In *Proc. 33rd FOCS*, pp. 13-22, 1992.

[A94] S. Arora, *Probabilistic checking of proofs and the hardness of approximation problems*. Ph.D. Thesis, UC Berkeley, 1994.

[AL96] S. Arora and C. Lund, *Hardness of Approximations*. In *Approximation Algorithms for NP-hard Problems*, Dorit Hochbaum, *Ed.*, PWS Publishing, 1996.

CS 15-855 Computational Complexity Theory **November 6, 2000**

Lecture 20: The PCP Theorem: 1

Lecturer: Madhu Sudan *Scribe: Amitabh Sinha / Editor: Abraham Flaxman*

Synopsis: Introduction to the PCP Theorem. History. Constraint Satisfaction Problems. Gap Problems. Preliminaries towards the PCP Theorem.

58 Introduction

The PCP Theorem is a new characterization of the complexity class NP in terms of proof verification systems. In terms we will soon formally define, it can be stated as $\mathsf{NP} = \mathbf{PCP}[\mathrm{O}(\log n), \mathrm{O}(1)]$. This surprising result has numerous consequences in computational complexity theory and hardness of approximations.

In this lecture, we will introduce and explain the concept of probabilistically checkable proofs. We will formally state the PCP Theorem, and we will build up some machinery which will be used in later lectures to actually prove the PCP Theorem. In this lecture we will not actually prove anything.

Informally, the PCP Theorem can be stated as follows:

Theorem 58.1 *Consider a language $L \in \mathsf{NP}$, and a string x which may or may not be in L. There exists a format for writing a proof that $x \in L$ and a probabilistic polynomial time algorithm for verifying this proof with the following properties:*

- *The verification method reads only a constant number of bits of the proof.*
- (**Completeness:**) *Correct proofs are accepted with probability 1.*
- (**Soundness:**) *Proofs of false assertions are accepted with probability no more than $\frac{3}{4}$.*
- *The proof is only polynomially longer than "classical" proofs.*

The soundness parameter need not be exactly $\frac{3}{4}$; it can be made any arbitrary number greater than $\frac{1}{2}$.

This theorem displays a surprising co-existence of probability and logic. It also provides a machinery to prove that for many NP-hard combinatorial optimization problems, finding approximately optimal solutions is also NP-hard. This has spawned a new industry in the study of combinatorial optimization problems and approximation algorithms.

58.1 Formal Definition

Definition 58.1 *A* **verifier** *is a probabilistic polynomial time algorithm V^π, with oracle-access to a "proof" π.*

Definition 58.2 *A verifier is (r, q)-***restricted** *if on inputs of length n, it uses $r(n)$ random bits and queries $q(n)$ bits from the oracle.*

Definition 58.3 *A language L is in the class $\mathbf{PCP}_{c,s}[r, q]$ if there exists an (r, q)-restricted verifier V^π satisfying the following properties:*

- (**Completeness**) *If $x \in L$, then there exists a proof π such that $V^\pi(x)$ accepts with probability at least c.*
- (**Soundness**) *If $x \notin L$, then for all proofs π, $V^\pi(x)$ accepts with probability no more than s.*

A PCP class is therefore parametrized by four quantities. Often, the completeness parameter c is 1, in which case we skip mentioning it. If neither of the subscripts c, s are mentioned, we have $c = 1$ and $s = \frac{1}{2}$. Here n is the length of the test string x.

Using this definition, the PCP Theorem states that $\mathsf{NP} = \mathbf{PCP}[O(\log n), q]$ for some constant q.

The size of the proof is roughly exponential in the amount of randomness $r(n)$ (assuming that r is at least linear). The reason is that our verifier is restricted to be polynomial time, so it can access only polynomially many bits of the proof deterministically, and the number of bits it can address for oracle access is exponential in the number of random bits it has. If the proof was super-exponential, there would be a large number of bits which the verifier is never going to ask for, and we could simply prune them out of the proof.

59 History

59.1 History of Definitions

The origins of checking proofs probabilistically and interactively are in the seminal works of Goldwasser, Micali and Rackoff [GMR, 1985], and Babai [Ba, 1985] on interactive proof systems. A probabilistic verifier with oracle access can be traced to the work of Fortnow, Rompel and Sipser [FRS, 1988].

Babai, Fortnow, Levin and Szegedy [BFLS, 1991] focus on the computation time and size of the verifier and the proof (transparent/holographic proofs). PCP verifiers were implicitly defined in the work of Feige, Goldwasser, Lovasz, Safra and Szegedy [FGLSS, 1991], and explicitly in the work of Arora and Safra [AS, 1992].

59.2 History of Results

The sequence of results culminating in the PCP Theorem is long and interesting. We briefly overview this history and our proof of the PCP Theorem will follow the historical lines.

Phase 0: Triviality. It is automatic that $\mathsf{NP} = \mathbf{PCP}[0, poly(n)]$, because this allows the verifier to deterministically read the entire proof, and NP sets have polynomial length proofs. Note that allowing our verifier to use $\log n$ random bits does not add any power. The verifier can simply enumerate all possible values of these random bits. Hence $\mathsf{NP} = \mathbf{PCP}[\log n, poly(n)]$.

Phase 1: Opening results. The first non-trivial results came about in the works of Babai, Fortnow and Lund [BFL, 1990], [BFLS, 1991] and [FGLSS, 1991]. They showed that $\mathsf{NP} \subseteq \mathbf{PCP}[polylog(n), polylog(n)]$, where $polylog(n)$ means a polynomial in $\log(n)$. This was a remarkable breakthrough because it has an exponential reduction in the number of queries. However, the amount of randomness required is still super-logarithmic, and hence the containment is strict.

Phase 2: Characterizations. Arora and Safra [AS, 1992] first showed that $\mathsf{NP} = \mathbf{PCP}[\mathrm{O}(\log n), \mathrm{o}(\log n)]$. This was subsequently improved by Arora, Lund, Motwani, Sudan and Szegedy [ALMSS, 1992] to the exact characterization $\mathsf{NP} = \mathbf{PCP}[O(\log n), O(1)]$. This is asymptotically optimal in both parameters, but not tight. To be more precise about the notation, what the theorem really means is that there exists a constant c_q, such that for all NP languages L, there exists a constant c_r such that

$L \in \mathbf{PCP}[c_r \log n, c_q]$. The surprising fact about this proof was that the query complexity of the system was completely independent of the proof size!

Phase 3: Near-optimal characterizations. The next major development in this direction was a result of Polishchuk and Spielman [PS, 1994] showing that for every $\epsilon > 0$, $\mathtt{SAT} \in \mathbf{PCP}[(1+\epsilon)\log n, \mathrm{O}(1)]$. It was also known that the number of queries required to check a proof was at least 3. Håstad [Ha, 1997] showed that this in fact could be achieved, but with a slight loss in the perfect completeness we were dealing with earlier. Specifically, he showed that $\mathsf{NP} = \mathbf{PCP}_{1-\epsilon,\frac{1}{2}}[\mathrm{O}(\log n), 3]$, for every $\epsilon > 0$.

60 Proof Outline

We will roughly follow the historical path in proving the PCP Theorem. We will begin by showing the main steps in the Phase 1 result above, $\mathsf{NP} = \mathbf{PCP}[\mathrm{O}(\log n)$, $polylog(n)]$. Two tools we will use in this are *arithmetization* and *low degree testing*.

We will then digress briefly to study Multiprover Interactive Proofs (MIPs) and Optimal Inner Verifiers. These will be useful in later stages. We will then compose PCPs to obtain, first,

$$\mathsf{NP} = \mathbf{PCP}[O(\log n), O(1)],$$

and with one more composition,

$$\mathsf{NP} = \mathbf{PCP}_{1-\epsilon,\frac{1}{2}}[O(\log n), 3].$$

61 Constraint Satisfaction Problems and Gap Problems

Definition 61.1 *An instance of a Max w-CSP(B)* **(Constraint Satisfaction Problem)** *consists of n B-ary variables, and t w-ary constraints $C_1, C_2, \ldots, C_t$ on the variables. The objective is to find an assignment to the variables maximizing the number of satisfied constraints.*

CSPs arise naturally in the study of PCPs. There is a rough correspondence between $\mathbf{PCP}[r, q]$ and w-$\mathbf{CSP}(2)$ as follows. The bits of the proof correspond to variables, so that $B = \{0, 1\}$. The verifier's examination and verdict on each string corresponds to a constraint. Hence we have $t = 2^r$ constraints. The number of queries q is the width w of the CSP. Now it is clear that the probability of acceptance by the verifier is simply the proportion of maximum satisfiable constraints among all the constraints.

61.1 Gap Problems

Gap problems allow us to deal in a decision-problem framework (which is often used in the world of complexity) with approximation problems. For example, we could define the following gap version of a CSP: Gap w-$\mathrm{CSP}_{c,s}(B)$ is a decision problem where:

- (**YES** instances:) $x \in$ Gap w-$\mathrm{CSP}_{c,s}(B)$ if there exists an assignment to the binary variables satisfying at least a c fraction of the constraints.
- (**NO** instances:) $x \notin$ Gap w-$\mathrm{CSP}_{c,s}(B)$ if for all assignments to the binary variables, no more than a c fraction of the constraints are satisfied.

Our task is to distinguish YES instances from NO instances. Of course, some instances are neither YES nor NO instances; in that case we don't care what we output.

In the next lecture we will show that GAP 3-$\mathrm{CSP}_{1-\epsilon,\frac{1}{2}}(2)$ is NP-hard.

61.2 Polynomial Constraint Satisfaction

Coming again from the perspective of CSPs, suppose B is a finite field $\mathbb{F}$. Let $\mathbb{F}^m$ be an m-dimensional vector space such that $n = |\mathbb{F}|^m$, where n is the number of variables. Assignments to the variables are therefore functions $f : \mathbb{F}^m \to \mathbb{F}$. We now define the Polynomial Constraint Satisfaction (PCS) problem.

Definition 61.2 *An instance of a w-PCS$(d, \mathbb{F})$* **(Polynomial Constraint Satisfaction Problem)** *consists of w-ary constraints on functions $f : \mathbb{F}^m \to \mathbb{F}$. The objective is to find an m-variate polynomial of degree at most d which satisfies all constraints.*

This is essentially a constraint satisfaction problem, with an extra low-degree restriction. We can also define the Gap version of the PCS, in terms of a parameter t.

Definition 61.3 *An instance of Gap-PCS$_{1,\epsilon}(m, w, d, q)$ has the usual completeness and soundness parameters (1 and ϵ), along with the restrictions that $m = m(t)$, $d = d(t)$, $w = w(t)$ and $|\mathbb{F}| = q(t)$.*

We will now state three Lemmas which play a very important role towards proving the PCP Theorem. The first deals with the hardness of Gap-PCS.

Lemma 1. *For every $\epsilon > 0$, Gap-PCS$_{1,\epsilon}(m, w, d, q)$ is NP-hard for $w, d, q = polylog(t)$ and $m = \log t / \log\log t$.*

This Lemma immediately gives us a problem where we have a gap. Moreover, since q^m is polynomial in t, we can just ask for the function f to be listed as a table of all its values. The verifier can pick a random constraint (from the t constraints) and verify it in time polylogarithmic in t. These seem to give us a lot of power. On the flip side, however, we have a low-degree restriction on f which a priori may seem to be troublesome.

61.3 Low degree Testing

This brings us to the problem of being efficiently able to test for low degree polynomials. We'd like to be able to do the following, though it turns out that it is impossible.

Definition 61.4 *Given a positive integer d and oracle access to a function $f : \mathbb{F}^m \to \mathbb{F}$, the* **low degree testing problem** *is to test whether f is a polynomial of degree no more than d in time polynomial in m and d. If yes, we should accept (completeness); if no, we should reject with high probability (soundness).*

This is really asking for too much. For example, let f be a function which is exactly a degree d polynomial everywhere except at one point in $\mathbb{F}^m$. We are therefore forced to weaken our soundness requirement. We first define closeness of two functions.

Definition 61.5 *Functions f and g are δ-close if $Pr_x[f(x) \neq g(x)] \leq \delta$, for x chosen uniformly at random.*

We now revise our definition of the low degree test to one which is actually achievable.

Definition 61.6 *Given d, δ and $f : \mathbb{F}^m \to \mathbb{F}$, the* **low degree test** *is to verify if f is close to a polynomial of degree no more than d. If it is, we should accept (completeness); but we should reject with high probability if it is not δ-close to any degree d polynomial (soundness).*

We are now ready to state our second Lemma.

Lemma 2. *There is a low-degree test, which has running time polynomial in $m, d, \frac{1}{\delta}$ and which uses $O(m \log |\mathbb{F}|)$ random bits.*

61.4 Self correction of polynomials

Definition 61.7 *The problem of* **self correcting a polynomial** *can be stated as this. Given d, δ, a function $f : \mathbb{F}^m \to \mathbb{F}$ and an element $x \in \mathbb{F}$ such that f is δ-close to a degree d polynomial p, can we compute p in polynomial time?*

We can answer this question in the affirmative.

Lemma 3. *There is a randomized Self-corrector for polynomials that runs in time polynomial in $m, d, \frac{1}{\delta}$ and uses $O(m \log |\mathbb{F}|)$ random bits, provided δ is sufficiently small.*

None of these Lemmas have been proved as yet. We will prove them in the coming lectures. First let's see what we can do with them.

62 PCP verifier for Phase 1

Suppose we are given a string x to test for membership in some NP language L. We compute a corresponding ϕ, which is in Gap PCS if and only if $x \in L$. The prover provides an oracle for the satisfying assignment f. We first run the low-degree test on f, and reject the proof if the low degree test rejects. We next pick a random constraint C of ϕ, and verify that Self-Correct(f) satisfies C. If not, we reject. If we pass all these tests, we accept the proof that $x \in L$.

This requires only $polylog(|x|)$ bits of the proof, and uses only $O(\log |x|)$ random bits. We have perfect completeness, and soundness $s \ll \frac{1}{2}$. This proves the first PCP theorem.

Theorem 1. $\mathsf{NP} \subseteq \mathbf{PCP}[\log n, polylog(n)]$.

The next lectures will prove the Lemmas, and then work towards formally proving the main PCP Theorem.

References

[ALMSS, 1992] S. Arora, C. Lund, R. Motwani, M. Sudan and M. Szegedy. Proof verification and hardness of approximation problems. *Journal of the ACM*, 45(3):501-555, 1998.

[AS, 1992] S. Arora and S. Safra. Probabilistic checking of proofs: A new characterization of NP. *Journal of the ACM*, 45(1):70-122, 1998.

[Ba, 1985] L. Babai. Trading group theory for randomness. In *Proceedings of the Seventeenth Annual ACM Symposium on the Theory of Computing*, 421-429, 1985.

[BFLS, 1991] L. Babai, L. Fortnow, L. Levin and M. Szegedy. Checking computations in polylogarithmic time. In *Proceedings of the Twenty Third Annual Symposium on the Theory of Computing*, 21-31, 1991.

[BFL, 1990] L. Babai, L. Fortnow and C. Lund. Non-deterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1:3-40, 1991.

[FGLSS, 1991] U. Feige, S. Goldwasser, L. Lovasz, S. Safra and M. Szegedy. Interactive proofs and the hardness of approximating cliques. *Journal of the ACM*, 43(2):268-292, 1996.

[FRS, 1988] L. Fortnow, K. Rompel and M. Sipser. On the power of multiprover interactive protocols. *Theoretical Computer Science*, 134:545-557, 1994.

[GMR, 1985] S. Goldwasser, S. Micali and C. Rackoff. The knowledge complexity of interactive proofs. *SIAM Journal of Computing*, 18:186-208, 1989.

[Ha, 1997] J. Håstad. Some optimal inapproximability results. Technical Report TR97-037, Electronic Colloquium on Computational Complexity, 1997.

[PS, 1994] A. Polishchuk and D. Spielman. Nearly-linear size holographic proofs. In *Proceedings of the Twenty Sixth Annual Symposium on the Theory of Computing*, 194-203, 1994.

CS 15-855 Computational Complexity Theory **13 November, 2000**

Lecture 22: PCP Theorem, Part 3

Lecturer: Madhu Sudan *Scribe: Nick Hopper / Editor: Nikhil Bansal*

Synopsis: Definition of Multi-Prover Interactive Proofs (MIP). Construction showing NP $\subset$ PCP$[poly(n), 1]$. Composition of PCPs to show that NP = PCP$[\log n, 30]$.

63 Multi-Prover Interactive Proofs (MIP)

So far all of the Interactive Proofs we have seen (IP, AM, and to an extent, PCP) have involved a probabilistic, polynomial time verifier V interacting with a single prover or oracle. As we have seen, these proofs can be extremely powerful, as they are capable of proving membership in any language in PSPACE. Still, we might wonder, what can be proved if we allow our verifier to interact with more than one prover? If we allow the verifier to interact with p provers who are not allowed to communicate once the proof begins, and ask each only a single question, we get a model of computation called Multiprover Interactive Proofs. We'll flesh out this informal idea with some definitions.

Definition 63.1 *A probabilistic polynomial time interactive Turing machine V is called a* (p, r, a)-restricted verifier *if it makes one query to each of p provers, receives at most a bits of response, and tosses at most r coins.*

With this definition, we can define parameterized complexity classes for MIP, just as we have for PCP.

Definition 63.2 *A language L is in $MIP_{c,s}[p, r, a]$ if there exists a (p, r, a)-restricted verifier V checking $x \in L$ with completeness c and soundness s.*

As usual, we will leave off the completeness and soundness parameters when it is clear from the context what they should be, or when $c = 1$ and $s = \frac{1}{2}$.

Why should we study MIP? First, they give us another clean model in which to think about interactive proofs. Second, this model is more restrictive than PCP — it gives us more fine-grained accounting of the parameters of interest, since $\mathsf{MIP}[p, r, a] \subseteq \mathsf{PCP}[r, pa]$. That is, we can always construct a PCP oracle for which each bit of a single prover's response to a particular question can be queried individually, and then the MIP verifier can be run on the results. In this section, we'll demonstrate the opposite – that is, we'll take our $(\log n, poly(\log n))$-restricted PCP verifier for NP and turn it into a 3-prover MIP verifier with similar efficiency. Surprisingly, it will turn out that this is a central intermediate step towards our goal of showing $\mathsf{NP} = \mathsf{PCP}[O(\log n), O(1)]$.

63.1 3 Prover MIP for NP

Recall that in the previous lecture, we demonstrated a $\mathsf{PCP}[O(\log n), poly(\log n)]$ verifier for NP via Gap-PCS. The proof consists of a supposed low-degree polynomial allegedly satisfying the constraints; our verifier performs a low-degree test and then checks that the closest low-degree polynomial to the proof satisfies a randomly chosen constraint. The first part of this procedure is already a 2-prover MIP: ask one prover for $P(x)$ and another for the line $P_{x,y}$; do the low-degree test. But we can't just ask a third prover for the values of P at the w points $x_1, x_2, \ldots, x_w$ making up our randomly chosen constraint, because that would violate soundness – the prover could realize which constraint we're checking and reply with values satisfying only that constraint. So we need to find a way to query the third prover at w points without revealing which constraint we're checking.

The solution to this problem is to pick a "random curve" $\mathbb{C}$ passing through the points $x_1, x_2, \ldots, x_w$ and ask the third prover for the value of P restricted to $\mathbb{C}$, $P|_{\mathbb{C}}$. If the prover responds with $P|_{\mathbb{C}}$ where P satisfies the constraints, we're fine; completeness is preserved. On the other hand, if the prover responds with some wrong polynomial h then with high probability $P(\mathbb{C}(t)) \neq h(t)$ on a random point t, so we also preserve soundness.

63.2 Random Curves

Thus we need only to refine our notion of a "random curve" through $\mathbb{F}^m$. We'll define a curve $\mathbb{C}\colon \mathbb{F} \to \mathbb{F}^m$ as a collection of m functions $\mathbb{C}_i\colon \mathbb{F} \to \mathbb{F}$, and define $\deg \mathbb{C} = \max_i\{\deg \mathbb{C}_i\}$. Then it should be obvious that for any $x_1, \ldots, x_w$, there are many such curves:

Proposition: For any $x_0, x_1, \ldots, x_w$ there exists a degree w curve $\mathbb{C}$ with $C(j) = x_j$.

Proof: We can perform polynomial interpolation on each of the m coordinates of the $w+1$ points, yielding m degree-w polynomials. Thus the degree of the curve will be w as well. ∎

We also need to know that in restricting the polynomial P to a curve, we don't cause the degree to increase so much that our low-degree tools won't work. This next proposition gives us what we need, since we've chosen both w and d to be polylogarithmic in n.

Proposition: If the polynomial P has degree at most d and the curve $\mathbb{C}$ has degree at most w then the polynomial $P|_{\mathcal{C}}$ has degree at most wd.

Proof: Each variable in the expression for P can be replaced by a degree-w polynomial by the curve $\mathbb{C}$. The degree of each term in P will then be increased by a factor of w, resulting in a polynomial of degree wd. ∎

Finally, we need to know that we really can get "random points" from the curve $\mathbb{C}$ to give to provers 1 and 2 in the low-degree test. The following proposition satisfies our needs, guaranteeing that while the points on a "random curve" are not independent, they are distributed uniformly.

Proposition: For every $x_1, x_2, \ldots, x_w$, call the curve resulting from picking x_0 uniformly at random and fitting $\mathbb{C}$ to the points $x_0, x_1, \ldots, x_w$ as above a random curve. Then for $t \notin \{1, 2, \ldots, w\}$, $\mathbb{C}(t)$ will be distributed uniformly at random over $\mathbb{F}^m$.

Proof: We need to show that for arbitrary $x \in \mathbb{F}^m$, $Pr_{x_0}[\mathbb{C}(t) = x] = 1/|\mathbb{F}|^m$. Suppose we are given $x_1, \ldots x_w$, t, and x, and we want to find the curve $\mathcal{C}'$ defined by $\mathcal{C}'(j) = x_j$ for $j \in \{1, \ldots, w\}$, and $\mathcal{C}'(t) = x$. Since there is a unique degree w curve satisfying these points, there will be a unique value for x_0 such that $\mathcal{C}'(0) = x_0$. The probability that we pick this x_0 when we define the curve $\mathbb{C}$ is $1/|\mathbb{F}|^m$, as required. ∎

63.3 Summarizing the MIP verifier

So in summary, our 3-prover MIP for Gap-PCS has the following steps, where Π_1 will be our "random point" prover, Π_2 will be our "random line" prover, and Π_3 will be the "random curve" prover:

- Random Choices:
  1. Pick a constraint C_j at random.
  2. Pick a random curve $\mathbb{C}$ passing through the w points of P that C_j constrains.
  3. Pick a random point x on $\mathbb{C}$.
  4. Pick a random line $l_{x,y}$ passing through x.
- Queries:
  1. Let a be the result of asking Π_1 for $P(x)$.
  2. Let g be the answer from Π_2 for $P_{x,y}$.
  3. Ask Π_3 for $P_{\mathbb{C}}$, and call the result h.
- Tests:
  1. Reject unless $g(x) = h(x) = a$.
  2. Reject if C_j is not satisfied by $h(1), h(2), \ldots, h(w)$.
  3. Accept otherwise.

Theorem 63.1 $NP \subseteq MIP[3, O(\log n), poly(\log n)]$

Proof sketch: First, it is fairly easy to see that for a YES instance of Gap PCS, there are three provers which cause the verifier to accept with probability 1: these provers are equipped with the low-degree polynomial satisfying our constraints.

Now, consider a NO instance. Let $\tilde{P}$ be the function used by Π_1. If $\tilde{P}$ is not δ-close to some low-degree polynomial P then the test $g(x) = a$ will fail with probability $\delta/2$, by our lemma on low-degree testing from the previous lecture.

If $\tilde{P}$ is δ-close to some low-degree polynomial P, then with probability at least $(1-\epsilon)$ the verifier has chosen a clause C_j such that $\tilde{P}$ does not satisfy C_j. In this case,

there are two possibilities. The first is that Π_3 responds with some h such that $h \neq P|_{\mathbb{C}}$; but then with non-zero probability $h(x) \neq P(x)$ but (since $\tilde{P}$ is δ-close to P) $\tilde{P}(x) = P(x)$, and we reject in test 1. Otherwise, if Π_3 responds with the curve $h = P|_{\mathbb{C}}$ then h will not satisfy C_j, and so we will reject in test 2.

■

64 Reducing PCP Query Complexity for NP

On exercise 1 of Problem Set 7, we demonstrated that $SAT \in \mathsf{PCP}[n^3, O(1)]$. In this section we'll use the same techniques to show that $CSAT \in \mathsf{PCP}[poly, 1]$.

Since $\oplus, \wedge, \neg$ is a complete set of gates (which happens to arithmetize well), we'll focus on that version of $CSAT$. An instance is a circuit K composed on k inputs, composed of $n - k$ AND, XOR, and NOT gates; the question is whether K has a satisfying assignment. It is not too hard to see that $CSAT$ is NP-complete. One simple witness for $CSAT$ is to provide a satisfying assignment to the inputs $a_1, \ldots, a_k$. Since we know that $CVAL \in P$ this assignment should suffice. But an easier witness to verify presents not only a satisfying assignment to the inputs, but also the output of all gates: $a_1, \ldots, a_k, \ldots, a_n$. This is easy to verify: if gate j has inputs a_{i_1}, a_{i_2} and output a_{i_k}, then the output should be consistent with the inputs and the behavior of the correct gate type; also, the output gate should output 1.

The result is that we can express $CSAT$ in terms of another low-degree polynomial problem: $QPSAT$. An instance of $QPSAT$ is a set of t degree 2 (quadratic) polynomials $p_1, \ldots, p_t$ on n elements of $\mathbb{F}_2$; the YES instances are those in which the polynomials have a common zero. The reduction from $CSAT$ expresses the above constraints, that is, an XOR gate with inputs a_{i_1}, a_{i_2} and output a_{i_3} results in the polynomial $a_{i_1} + a_{i_2} + a_{i_3}$, an AND gate results in the polynomial $a_{i_1} a_{i_2} + a_{i_3}$, and a NOT gate results in the polynomial $a_{i_1} + a_{i_2} + 1$. So to prove $CSAT$ it will be sufficient to provide an appropriate encoding of $a_1, \ldots, a_n$.

Fortunately, from our homework we already know an appropriate encoding of an assignment $a_1, \ldots, a_n$ on a degree 2 polynomial. We'll encode the assignment $a_1, \ldots, a_n$ as two functions, a linear function $\sum_i a_i x_i$ (This is called the Hadamard encoding of a) and a quadratic function $\sum_{ij} a_i a_j x_{ij}$ (which we'll call the Quadratic Functions or QF encoding of a). In the homework we exhibited tests which allow us to determine if a Hadamard encoding and a QF encoding are consistent, and we demonstrated that we can self-correct in case the encodings are only δ-close to linear and quadratic

functions. By a slightly modified argument, we can form the random polynomial $P_r = \sum_j r_j p_j$ and test to see whether the given encoding causes P_r to evaluate to 0; if K is satisfiable then a satisfying assignment will always cause P_r to evaluate to 0, whereas if K is not satisfiable then P_r will evaluate to 0 with probability at most $\frac{1}{2}$. Summing the number of probes into the proof required, we see that 3 probes each are required for linearity testing of the Hadamard and QF encodings (6 total), 4 probes (actually, one self-corrected probe of each) are required to test the consistency of the Hadamard and QF encodings, and a single self-corrected probing of the QF encoding is required to test whether $P_r(a) = 0$. Therefore we have a construction which demonstrates $\mathsf{NP} \subseteq \mathsf{PCP}[poly(n), 12]$.

65 Composition of PCPs

Recall that in section 1, we gave a 3-prover MIP for NP. This verifier had the advantage of reaching our goal of $O(\log n)$ random coins, but the answers (or query complexity) were $poly(\log n)$ bits. One thing to notice about this verifier is that its verdict is an easily computable function of the answers it gets: that is, it is computable by a circuit of size $poly(\log n)$. On the other hand, our previous construction yields a query-efficient verifier with soundness bounded away from 1, but which needs exponential sized proofs, and verifies circuit satisfiability. What we would really like is if there was some way to get the randomness complexity of the MIP verifier, but with the query complexity of the PCP verifier.

But reading over our summary should lead to an idea: compose the verifiers. A naive attempt might compose them in the following way: start with the 3-prover MIP verifier. Prepare queries q_1, q_2, and q_3 as before, and construct the small circuit ($poly(\log n)$ gates) C that determines our verdict. Then send our queries to the provers, but instead of asking for the responses a_1, a_2, a_3, ask for a proof that a_1, a_2, a_3 would satisfy C, using the 12-query PCP verifier. This is different from either of the previous systems, because it queries only 12 bits, and uses only $O(\log n)$ randomness for the MIP verifier and $poly(poly(\log n)) = poly(\log n)$ bits of randomness for the PCP verifier, that is, it uses only $poly(\log n)$ bits of randomness overall. Unfortunately this naive approach is not quite good enough, because the verifier just described violates the soundness condition: there's nothing guaranteeing that the satisfying solution to C wasn't chosen to satisfy just C and not any of the non-randomly chosen clauses. This is sort of like asking a SAT prover to give assignments to a few clauses separately, without checking to see whether the assignments are consistent between clauses.

Fortunately, we can overcome this difficulty with only a little extra work. The solution

is to force the prover to commit to the response to the query q_i (via a Hadamard encoding A_i of the resulting answer a_i) as well as a QF + Hadamard encoding B of a satisfying assignment to the circuit resulting from our choice of random coins. Then we can still use the PCP 12-query verifier on the QF + Hadamard encoding of the satisfying assignment for completeness. To get soundness, we need to check that each encoded answer A_1, A_2, and A_3 is consistent with the satisfying assignment to our circuit. But since each A_i is just a linear function encoding and we have a linear function encoding of all of the variables in A_i plus some extras, we can just test that, for a random x with $|a_i|$ bits, $A_i(x) = self - correct(B(x))$, where x is padded appropriately in the linear function encoding B. Thus these extra consistency checks take 3 queries each, and the entire construction makes 21 queries.

So in other words, a correct proof that $x \in Gap - PCS$ for our verifier will have the expected oracle form Π, where $\Pi(O, R, \cdot)$ will serve as the oracle giving the Hadamard + QF encoding of the satisfying assignment to the inner PCP verifier when tossing coins R; and $\Pi(i, q_i, \cdot)$ will serve as the Hadamard encoding of the response of prover i to question q_i in the outer MIP verifier, for $1 \leq i \leq 3$. Thus the complete verifier program will be:

- Toss random coins R as for the outer verifier
- Prepare the queries q_1, q_2, and q_3 as for the outer verifier
- Construct the circuit C which computes the response of the outer verifier as a function of the responses to the queries q_i constructed in the previous step.
- Run the inner verifier on the oracle $\Pi(0, R, \cdot)$ with input C. if the inner verifier rejects, then reject.
- Reject if $\Pi(i, q_i, \cdot)$ is not consistent with $\Pi(0, R, \cdot)$ for some $1 \leq i \leq 3$.
- Accept otherwise.

This verifier has randomness complexity $poly(\log n)$ and query complexity 21, and soundness bounded away from 1, so we get the result $\mathsf{NP} \subseteq \mathsf{PCP}[poly(\log n), 1]$.

66 The PCP Theorem

So far we've come very close to our goal of showing $\mathsf{NP} = \mathsf{PCP}[\log n, 1]$, but we're not quite there yet: we need to get our verifier down to logarithmic randomness

rather than polylogarithmic. The randomness requirement of the construction in the previous section is polylogarithmic because the provers for the outer verifier return answers of size $poly(\log n)$ and the inner verifier requires randomness polynomial in the size of its input. So we need some way to make the circuit even smaller.

Note that we have introduced a general composition paradigm in which, given an outer verifier and an inner verifier of the correct types, we can compose the two to get the randomness efficiency of the outer verifier combined with the query efficiency of the inner verifier, so long as we can force the proof to commit to an answer consistent with the queries of the outer verifier. For the outer verifier, we require an MIP verifier with a small number of provers and easy decision criterion (small accept/reject circuit), which has low soundness error. For the inner verifier, we require a PCP with small query efficiency, which can verify the commitment to a proof, and which admits a self-corrector, for robustness.

Notice that our $\mathsf{MIP}[3, \log, poly(\log)]$ verifier can also be made into an inner-verifier for the NP predicate which our PCP verifier evaluates, yielding an $\mathsf{MIP}[6, \log, poly(\log\log)]$ verifier for NP. This is done by asking the second set of three provers for a proof that the answers from the first three would have caused the verifier's circuit to accept, along with verifying the consistency of the second set of provers' answers with the answers of the first set. But now notice that the size of the acceptance circuit of the inner verifier is $poly(\log(poly(\log n))) = poly(\log\log n)$. So we can compose the $\mathsf{MIP}[6, \log, poly(\log\log)]$ verifier with our $\mathsf{PCP}[poly, 21]$ verifier, which will now require $poly(poly(\log\log n)) = poly(\log\log n) = O(\log n)$ bits of randomness, add three more consistency checks, and get a $\mathsf{PCP}[\log, 30]$ verifier for NP. (Where we skip deducing the soundness constant) Thus, combined with our homework exercises showing $\mathsf{PCP}[\log, 1] \subseteq \mathsf{NP}$, we've shown that $\mathsf{NP} = \mathsf{PCP}[\log, 1]$.

CS 15-855 Computational Complexity Theory **November 15, 2000**

Lecture 23: Conditional/Unconditional Complexity

Lecturer: Adam Kalai *Scribe: Jason Crawford / Editor: Shuchi Chawla*

Synopsis: Conditional vs. unconditional complexity theory. Diagonalization and simulation, and their limitations. Oracle results and circuit complexity.

67 Conditional/Unconditional Complexity Theory

With this lecture we enter Phase III of the course. In Phase I, we studied basic definitions, building up an elaborate (hypothetical) world picture of complexity classes. In Phase II, we studied randomness and its applications to complexity. Both of these areas are *conditional*—their primary interest relies on unproven assumptions. (For instance, showing a new problem NP-complete is interesting only if $\mathsf{P} \neq \mathsf{NP}$.)

In Phase III we will study *unconditional* complexity theory—that which we can prove without any assumptions. Unfortunately, this will turn out to be: not much. For instance, we *can* prove:

- $\mathsf{P} \neq \mathsf{PSPACE} \vee \mathsf{P} \neq \mathsf{NC}$
- $\mathsf{P} \neq \mathsf{SPACE}(n)$
- $\mathsf{P} \subsetneq \mathsf{TIME}(n^{\log n})$
- Uniform $\mathsf{NC}_1 \supsetneq \mathsf{SPACE}(n)$

We *can't* even prove:

- $\mathsf{P} \neq \mathsf{PSPACE}$
- $\mathsf{PH} \neq \mathsf{L}$
- $\mathsf{RP} \neq \mathsf{EXP}$
- $\mathsf{BPP} \neq \mathsf{NEXP}$

67.1 A Constrast of Methods

In unconditional complexity research, we are usually interested in discovering something true of every algorithm or circuit which computes a particular problem. In conditional complexity, by contrast, we are primarily interested in showing *relationships* among problems (e.g., reductions). Without knowing anything about any algorithm to solve SAT, we can say that *if* SAT is easy, then many other problems (all of NP) are easy.

To look at this another way: An algorithms researcher, for instance, may discover (e.g., through a reduction) that some problem A being easy implies that another problem B is easy. He publishes "A easy $\Rightarrow$ B easy." Even better for his line of work, if A is known to be easy, he can conclude that B is easy.

A complexity theory researcher, on the other hand, proceeds as follows. He discovers "A easy $\Rightarrow$ B easy," and publishes: "B hard $\Rightarrow$ A hard." If B is a problem suspected to be hard (or if B is, say, all of NP), then this is strong formal evidence that A is hard.

As the field stands today, conditional complexity is "a thriving, deep and important area despite the fact that it sidesteps the issue of the existence of hard problems." Unconditional complexity, on the other hand, has not come nearly as far. Progress has been slow and difficult.

In this part of the course, we will study results proved in unconditional complexity, as well as meta-theoretic concerns about which lines of attack can and cannot be fruitful.

68 On Diagonalization and Simulation

Diagonalization and simulation have had tremendous success in math and logic. Cantor first used the technique to prove the uncountability of the reals. Gödel used it later to prove his Incompleteness Theorem. Thus it is not unreasonable to think that diagonalization and simulation can help us resolve the P = NP? question. However, a 1975 paper by Baker, Gill and Solovay [1] provides formal evidence that this technique may not in fact be useful.

68.1 Oracle Results

While there are many aspects of our world picture which we cannot prove, there are many different (and conflicting) hypotheses which can be shown to be true relative to a certain oracle. For instance, it can be shown that:

$$\begin{aligned} \exists A \quad & \mathsf{P}^A = \mathsf{NP}^A \\ \exists B \quad & \mathsf{P}^B \neq \mathsf{NP}^B \\ \exists C \quad & \mathsf{P}^C \neq \mathsf{NP}^C \wedge \mathsf{NP}^C = \mathsf{coNP}^C \\ \exists D \quad & \mathsf{NP}^D \neq \mathsf{coNP}^D \wedge \mathsf{P}^D = \mathsf{NP}^D \cap \mathsf{coNP}^D \\ \exists E \quad & \mathsf{NP}^E \neq \mathsf{coNP}^E \wedge \mathsf{P}^E \neq \mathsf{NP}^E \cap \mathsf{coNP}^E \end{aligned}$$

as well as many other similar results.

For concretization of the first two claims above, consider the following:

$$\mathsf{P}^{TQBF} = \mathsf{NP}^{TQBF}$$

(as seen in Lecture 5, both are equal to PSPACE), but for a *random* oracle A,

$$\mathsf{P}^A \neq \mathsf{NP}^A$$

(as seen on Homework 7.4).[7]

68.2 Relativizing Proofs

It may not be obvious, but oracle results such as these are actually evidence that diagonalization and simulation will not work to solve $\mathsf{P} = \mathsf{NP}$?. To see why, consider (for instance), the Time Hierarchy theorem. It begins with a simulator theorem: Any TM M running in time $f(|x|)$ can be simulated by another TM running in time $f^3(|\langle M, x\rangle|)$. This is followed by a diagonalization theorem: We can create a TM D_f which simulates any machine running in time f, and then does the opposite. Thus deciding the set accepted by D_f is in $\mathsf{TIME}(f^4)$, but not in $\mathsf{TIME}(f)$.

Now, notice that this argument would be *unchanged* if every machine and complexity class in the argument were augmented with a given oracle O. Such an argument is said to *relativize*. BGS's paper pointed out that, because of the conflicting oracle results mentioned above, *no relativizing argument can resolve P vs. NP.*[8]

[7] Baker, Gill and Solovay *constructed* such an oracle—by diagonalization!

[8] This does not necessarily mean that diagonalization and simulation will not eventually be used to show $\mathsf{P} \neq \mathsf{NP}$, as we will see in a future lecture.

Arguments which work by diagonalization and simulation typically treat the TM as a black box, and hence these arguments relativize. These results are discouraging, because most arguments, such as the Space Hierarchy Theorem as well as the Speed-Up and Gap Theorems (and many more) all relativize.

69 Oracle Results and Circuit Complexity

In the late 1970s, complexity theory researchs had yet to prove two particular oracle results in the spirit of those above:

$$\exists A \quad \mathsf{PH}^A \subsetneq \mathsf{PSPACE}^A$$
$$\exists A \quad \forall i \quad \Sigma_i^A \neq \Sigma_{i+1}^A$$

Recall, however, as we have seen in previous lectures and on homework 4.5, that there is a deep connection between classes like PH and PSPACE and certain types of circuits. It is thus that, in trying to solve the first of these by diagonalization, Furst, Saxe and Sipser ran into a question in circuit complexity:

Question: Must any constant-depth circuit to compute parity have $\Omega(2^{\log^i n})$ gates, for all i?

Furst, Saxe and Sipser [2] showed in a 1981 paper that the parity function is not in AC^0, but this was not enough to prove the existence of the desired oracle. Yao[3] and Håstad[4] proved successively better lower bounds which did establish the result. Håstad finally showed that parity requires $\Omega(2^{(\frac{1}{10})\frac{d}{d-1}\sqrt[d-1]{n}})$ gates to compute with a depth d circuit family.

Håstad also proved that there exist functions $f_1, f_2, \ldots$ such that for all k, f_k is computable by a poly-size depth-k circuit family, but requires superpolynomial-size depth-$(k-1)$ circuits. This was sufficient to prove the second desired oracle result.

Remark: All that was required for the desired oracle results was a lower bound for uniform AC^0. All the proofs, however, show the stronger non-uniform bound. Complexity theory researchers show non-uniform bounds because they can't find any use for uniformity outside of diagonalization.

70 Hope For the Future?

In showing these circuit bounds, something more important than the oracle results had been accomplished. The proofs of the circuit bounds were *non-relativizing*. Rather than relying on black-box type arguments, these proofs actually dug around in the guts of the circuit and made assertions about what the computation would have to look like. This suggested that perhaps the $\mathsf{P} = \mathsf{NP}$? question can be resolved by circuit complexity.

The rest of the course will concentrate on circuit lower bounds, concluding with a lecture on "Natural Proofs," an idea by Razborov and Rudich which shows the inherent limitations of this approach.

References

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

[2] M. Furst, J. B. Saxe, and M. Sipser. Parity, circuits, and the polynomial-time hierarchy. *22nd Annual Symposium on Foundations of Computer Science*, 1981, pp 260-270.

[3] A. Yao. Separating the polynomial-time hierarchy by oracles (preliminary version). *26th Annual Symposium on Foundations of Computer Science*, 1985, pp 1-10.

[4] J. Håstad. Almost optimal lower bounds for small depth circuits. *18th Annual ACM Symposium on Theory of Computing*, 1986, pp 6-20.

CS 15-855 Computational Complexity Theory **November 20, 2000**

Lecture 24: Lower Bounds for Constant-Depth Circuits

Lecturer: Adam Kalai *Scribe: Venk Natarajan / Editor: Chris Wallace*

Synopsis: Discussion of lower bounds for constant-depth circuits. Hastad's Switching lemma. Lower bound on circuit size for constant depth parity circuit.

The main thrust of this lecture is to prove that there is no way to create a reasonably-sized constant depth circuit to evaluate the parity function. First we will discuss two concepts that will be very useful in proving this fact – Restrictions and Decision Trees. Next, we'll talk about the Hastad Switching lemma, which is the "hardest" thing we'll need to prove in order to accomplish our task.

71 Preliminaries

71.1 The Parity Function

As noted in the synopsis, we will prove that the parity function takes exponentially many gates to compute in constant depth. For review:

Definition 71.1 (Parity Function) *The parity function, denoted $\oplus$ or XOR, returns 1 iff an odd number of the inputs are set to 1, and returns 0 otherwise.*

Despite the very simple statement of parity, the parity function takes very many gates to compute in a constant depth regardless. Why? Why is parity so hard? The main reason is as follows – flipping a single bit of the input to the pairty function will *always* change the output. We formalize this as follows:

71.2 Restrictions

Definition 71.2 (Restriction) *A restriction $\rho : [1..n] \to \{0, 1, *\}$ of $f : 0, 1^n \to \{0, 1\}$, denoted $f \restriction \rho$, is the function induced by $x_i \to P(i)$ if $P(i) = 0$ or $P(i) = 1$,*

and $x_i \rightarrow x_i$ if $P(i) = *$. We call a restriction interesting if $\rho(i) = *$ for some i.

In other words, think of the numbers in $[1..n]$ that are being set to "*" as being "unset" bit positions. Then think of $f \restriction \rho$ as being the function that we get when we only consider those inputs that are "consistent" with the values given to those bit positions that are set. For instance, if $\rho(5) = 1$, then we only consider inputs to f whose fifth bit is set to 1. But if $\rho(5) = *$, then we consider inputs regardless of what the fifth bit is set to.

Now, any "reasonably sized" constant-depth AND/OR/NOT circuit has a possible restriction to the inputs which induces a constant output. For example, we can restrict the AND by setting the 1st bit to be 0. If the first bit of the input to any AND function is forced to be zero, then the AND will never output anything other than zero.

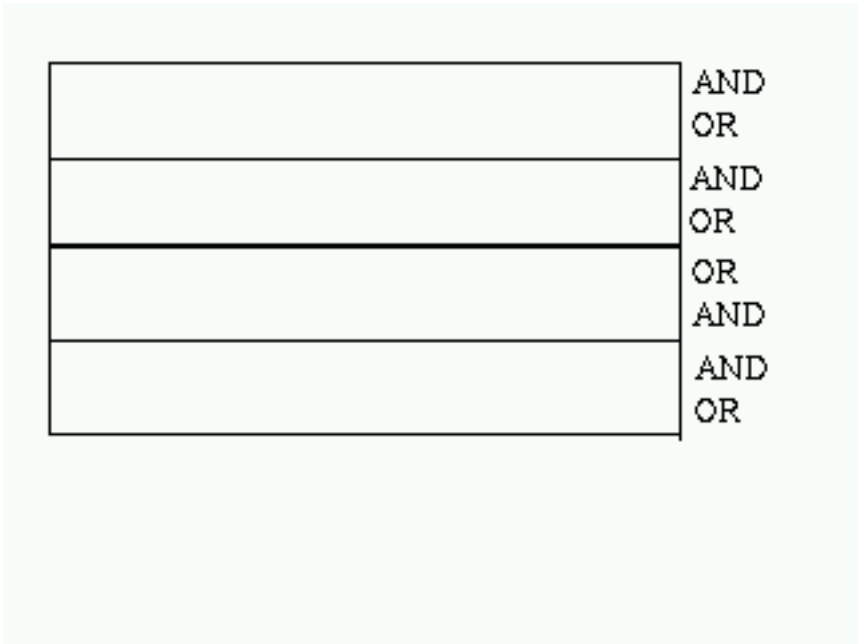
So, we are now ready to frame the "difficulty" of the parity functon in terms of restrictions. That is, the only way to restrict the input to the parity function and get a constant function, is if we restrict *all* of the bits of input. In other words, no interesting restriction of the parity function produces a constant function.

So it is enough to see that if we van prove that, for any small constant-depth circuit, that there is an interesting restrition that produces a constant function, then we will be done.

How are we going to prove the existence of such a circuit? Randomness!

71.2.1 Why we Care about Restrictions

The idea of the Hastad Switching Lemma (we'll formally state the lemma later) is that, with high probability, a random restriction on a circuit will turn an OR of a small number of ANDs into an ANDs of a small number of ORS. So even if we have multiple such ORs of ANDs, we'll be able to find at least one restriction that switches *all* of these ORs of ANDs into ANDs of ORs. Why is this ability useful?



The first two boxes in the above diagram respresent a circuit that has a layer of ORs, followed by a layer of ANDS, then one of ORs, then one of ANDs. The second two boxes represent what the circuit would look like if we were able to switch an OR of ANDS (represented by the first box above) into an AND of ORs, without making drastic changes to the number of gates. Then we would have two consecutive rows of ANDs, as shown in the second pair of boxes, which we may then merge into one row without any significant increase in circuit size.

We end this section by definining a class of restrictions of a certain type.

Definition 71.3 $\underline{R_n^\ell}$ *is the set of all restrictions on* n *varaibles that leave* ℓ *variables unset.*

This class of restrictions will come in handy when we talk about the Hastad Switching Lemma.

71.3 Decision Trees

Definition 71.4 (Decision Tree) *A* <u>*decision tree*</u> *is a binary tree whose nodes are labelled with a bit positon of the input, and whose edges are labelled with 0 or 1. We*

traverse from the root of the tree to the leaf, following the 0 or 1 edge, depending on what the value of the bit referred to by our current node is.

Now, let's look at a decision tree, and consider what it does to a set of inputs. Suppose that the tree has depth S. We output a 1 iff we go down some path to a leaf that tells us to output 1. Each path represents an AND of x_i's (where some of these x_i's may be negated), and there can only be S terms that we have to follow. So it's easy to see that a decision tree can be expressed as an OR of ANDS, where each AND has at most S variables.

Using DeMorgan's laws to create a complement tree, one can check that a decision tree can also be exressed as an AND of ORs, where each OR has at most S variables.

Now, let us consider a formula F that is given in DNF, so that the terms are preseneted in some canonical order. (The particular choice of order isn't important, but this choice is made beforehand and fixed throughout all this).

That is,

$$F = C_1 \vee C_2 \vee C_3 \cdots \vee C_n$$

We can, as seen, also restrict this formula in the same way that we can restrict a circuit or any boolean function. The remaining function is also in DNF, and will usually be simpler. (An easy exercise to see this).

So

$$F \restriction \rho = C_1' C_2' \ldots C_n'$$

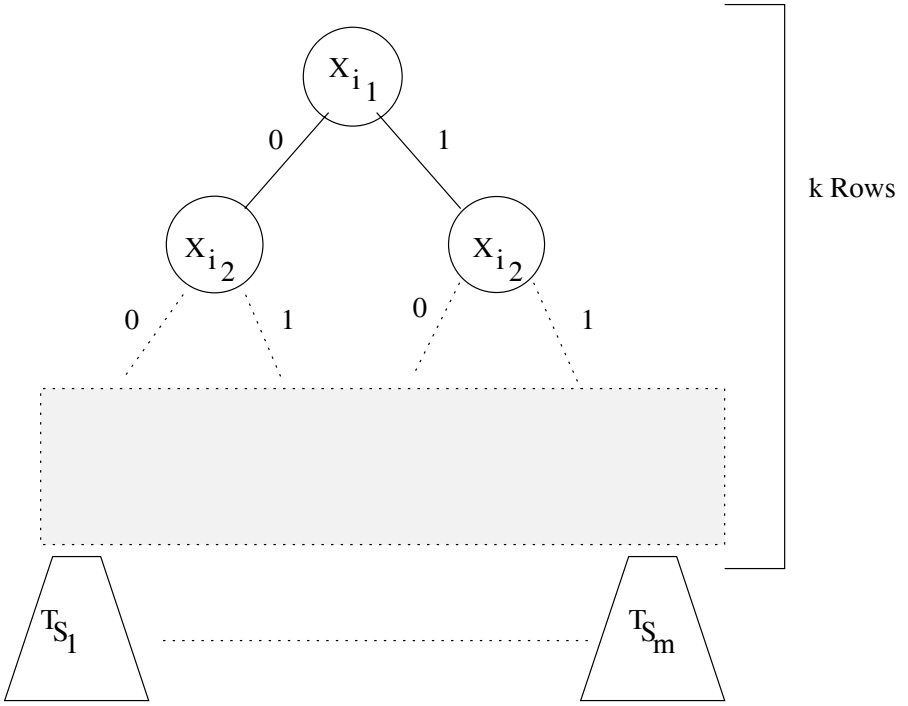
Where each $C_i' = C_i \restriction \rho$. Each C_i' will look like the original term C_i, except that some variables may be gone. But those variables that do remain are presented in the same order as they are in the C_i.

Example: Let $F = x_1\overline{x_2}x_e \vee x_1x_2x_3$. Let $\rho(i) = 1$, $\rho(2) = \rho(3) = *$. Then $F \restriction \rho = \overline{x_2}x_3 \vee x_2x_3$.

So let us come back to decision trees. We can, given a DNF formula F, inductively

define the canonical decision tree for F as follows (this is where we need to use the fact that we have a canonical ordering to work with):

Definition 71.5 *We inductively define the Caoninical Decision Tree for F: $T(F)$ is the tree that always returns 0 if F has no terms. If $F = C_1 \vee F'$, where C_1 is the empty term, then $T(F)$ is the tree that returns 1. (Think of this as an empty and always returning true...). Now, suppose that C_1 uses variables $x_{i_1}, x_{i_2} \ldots x_{i_k}$. Then $T(F)$ is the tree represented below.*



Ok, so what is each T_{S_j}?

The tree that we get should output 1 on any input that makes C_1 true. So for all paths of S_1 true, we do not have a T_{S_j} after the first k rows, and instead output 1. On the other hand, if we take a path for which C_1 is 0, then we have to evaluate F. So think of each S_j as being a truth assignment on $x_{i_1}, \ldots, x_{i_k}$, and each T_{S_j} is located by following the path down T as determined by the given truth assignment. If we are given a truth assignment, then we don't need to know *all* of F'. Having gone down the given path, we already know what each of the $x_{i_1}, \ldots, x_{i_k}$ are. So we really only need to know what $F' \restriction \rho_j$ is, where ρ_j is the restriction resulting from setting $x_{i_1}, \ldots, x_{i_k}$ to be the values in the truth assignment S_j.

71.3.1 Why we care about Decision Trees

The nice thing about decision trees is that they can be computed as a small OR of ANDs, or a small AND of ORs. So they give us a way of formalizing the "switching" effect of restrictions that we talked about before.

72 Hastad Switching Lemma

The Hastad Switching Lemma tells us that with high probability, a restriction of a large proportion of the variables will create a shallow decision tree:

Lemma 72.1 (Hastad Switching Lemma) *Let F be an OR of ANDS of size $\leq r$. For $k \geq 0$, $\ell = pn$, $p \leq \frac{1}{7}$, we have:*

$$\frac{\left|\left\{\rho \in R_n^\ell \mid DEPTH\left(T(F \restriction \rho)\right) \geq k\right\}\right|}{|R_n^\ell|} \leq (7pr)^k$$

So if we restrict most of the variables (here, $\frac{6}{7}$ of them), then the tree will very likely have a small depth.

Example: Let us consider an OR of v variables from among n inputs. Let $r = 1$, $k = \log v$, $p = \frac{1}{14}$, and $\ell = \frac{n}{14}$. Then the probability that we can't do the restricted OR by a decision tree of depth less than $\log v$ is at most $\frac{1}{2}^k = \frac{1}{2^{\log v}} = \frac{1}{v}$.

For now, let us assume that the Hastad Switching Lemma is true, and see how that will apply to the main theorem (i.e., there is no small constant-depth circuit family computing the parity function). We will later return to proving the Hastad Switching Lemma.

72.1 Application of the Hastad Switching Lemma

Theorem 72.2 *There is no constant-depth circuit family that computes the parity function.*

Proof:

Here's a basic fact from probability theory:

Fact 3 (Boole's Inequality) *If $X_1, \ldots, X_n$ are events, then:*

$$\Pr(X_1 \vee \cdots \vee X_n) \leq \sum_{i=1}^{n}$$

Example 72 showed us that if we restrict $\frac{13}{14}$ of the inputs, then there is only probability at most $\frac{1}{v}$ that a restriction will fail to convert an OR into a decision tree of $\log S$ depth. If we have $S-1$ of these ORs, then by Boole's Inequality, the probability that a restriction will fail to convert all ORs into a depth $\log S$ decision tree is still less than 1. Therefore there *is* a restriction in $R_n^{n/14}$ that will convert *all* of $S-1$ ORs into some tree of depth $\log S$.

So, let's take a circuit of n inputs, size S and depth d, and restrict $\frac{13}{14}$ths of the inputs. Then we have a circuit that is size S, depth d, but with only $\frac{n}{14}$ inputs. Each input wire of depth 1 can be computed by a depth $\log S$ decision tree, and thus can be computed by *both* an OR of $\log S$ ANDs, and as an AND of $\log S$ ORs.

Inductively, each wire at depth i can be computed by a decision tree of depth $\log S$.

So essentially, each time we are taking away 13/14 of the variables, and each time we are creating a tree of depth $\log S$. In this manner we can repeatedly switch ANDs and ORs, and then merge as we showed above, to turn the circuit into a constant size.

We can repeat the process here until we have a depth 2 circuit on $\frac{n}{14(14 \log S)^{d-2}}$ variables.

Now, remember from the homework that $\oplus$ requires 2^{n-1} gates to compute in a depth two circuit, and this comes in the form of OR or ANDs of full size. We don't have all n variables left, but we do have $\frac{n}{14(14 \log S)^{d-2}}$ variables left.

So, if S is the number of gates in any constant-depth circuit family computing parity, then $\log S \geq \frac{n}{14(14 \log S)^{d-2}}$. Solving for S yields:

$$S \geq 2^{\frac{1}{14} n^{\frac{1}{d-1}}}$$

■

72.2 Proof of Hastad Switching Lemma

We repeat the statment of the Hastad Switching Lemma given at the beginning of this section:

Lemma 72.3 (Hastad Switching Lemma) *Let F be an OR of ANDS of size $\leq r$. For $k \geq 0$, $\ell = pn$, $p \leq \frac{1}{7}$, we have:*

$$\frac{\left|\left\{\rho \in R_n^\ell \mid DEPTH\left(T(F \restriction \rho)\right) \geq k\right\}\right|}{|R_n^\ell|} \leq (7pr)^k$$

Proof:: Let $S = \{\rho R_n^\ell \mid DEPTH(T(F \restriction \rho)) \geq s\}$. The statement of the lemma says that S should be small, so clearly this is our aim. We will do this by giving a clever, altermative method of describing the members of S.

In particular, what we are going to do is exhibit a 1-1 map from S to a small set.

Let $STARS(r, s)$ be the collection of all sequences $(\beta_1, \beta_2, \ldots, \beta_\triangle)$, where $\beta_i \in \{*, -\}^r - \{-\}^r$, and the total number of stars in $(\beta_1, \beta_2, \ldots \beta_\triangle)$ equals s.

We will exhibit a $1-1$ map from S into $R_n^{\ell-s} \times STARS(r, s) \times \{0, 1\}^s$. But this map is only of use if we can establish that $R_n^{\ell-s} \times STARS(r, s) \times \{0, 1\}^s$ is actually a small set. So let's do that first:

1. $R_n^{\ell-s}$: We can choose $\ell - s$ elements to be unset, and the remaining variables ($n - (\ell - s)$ of them) can be set in $2^{n-(\ell-s)}$ ways. So there are $\binom{n}{\ell-s} 2^{n-(\ell-s)}$ elements in $R_n^{\ell-s}$.
2. $STARS(r, s)$: This is a trickier counting argument. In fact, to attempt to count this *exactly* would be a difficult, if not hopeless task. Instead, we can use upper bounds for the factorials and binomial coefficients to show that $STARS(r, s) < \left(\frac{r}{\ln 2}\right)^s$.
3. $\{0, 1\}^s$: This clearly has size 2^s.

So if we can find a map from S to $R_n^{\ell-s} \times STARS(r, s) \times \{0, 1\}^s$ then $|S|$ is at most the product of the three numbers we computed above, which is at most

$$2^{n-(\ell-s)} \left(\frac{r}{2}\right)^s 2^s$$

The same argument that we gave in the first item above shows that $\left|R_n^\ell\right| = \binom{n}{\ell} 2^{n-\ell}$.

Now arithmetic will give us, when $p < \frac{1}{7}$:

$$\frac{|S|}{|R_n^\ell|} < \left(\frac{4pr}{s(1-p)\ln 2}\right)^s < (7pr)^s$$

Construction of 1-1 map: Let $\rho \in S$. Then by definition of S, $DEPTH(T(F \restriction \rho)) > s$. Let π be the lexicographically first path in $T(F \restriction \rho)$ with length at least s. If π larger than s, then trim π to length S.

So we are going to use F, $T(F \restriction \rho)$, and π to calculate the map.

As we saw earler, the tree is basically build by piling trees for circuits on top of one another. So we can think of the path as being a sequence of pathsneeded for each circuit. Since the path has length at least s, we know that we must conver enough circuits of that length. In other words, we have the following

So we have the following:

$$\begin{aligned} C_1 \restriction \rho\pi_1 &= 0 \\ C_{i_2} 2 \restriction \rho\pi_2 &= 0 \\ &\vdots \\ C_{i_6} \restriction \rho\pi_1 \cdots \pi_6 = 0 & \end{aligned}$$

But, we know that $C_{i_7} \restriction \rho\pi_1 \cdots \pi_7$ is not equal to 0, since the path doesn't end (length is at least s). (7 above is just for expository terms – really what we're interested is the fact that the path has length s.)

Where each C above represents the "next" surviving conjunction after we apply the restriction ρ.

Consider the π_i partitions that we had above. We can let σ_k be what happens in the case where we had to go the other way instead of what we did after π_i ended. Then we have that $C_1 \restriction \rho\sigma_1 = 1, \ldots, C_{i_5} \restriction \rho\pi_1 \ldots \pi_4\sigma_5 = 1$, but if we then apply the pattern to C_{i_6}, we just get something nonzero, but perhaps not yet terminating either.

Let $STARS_i$ be so that the jth variable of C_i is set by σ_i if the j^th entry is a "*".

Then our map is $\rho \to (\rho\sigma_1 \dots \sigma_k, \{STARS_1, \dots, STARS_k\}, \delta)$, where δ is the bitwise XOR of the bits of π with those of $\sigma_1 \dots \sigma_k$.

Since our path is of length π, the scond coordinate will really have s stars in it.

We can then recover ρ from its image. C_1 is the first clause set to 1 by $\rho\sigma_1 \dots \sigma_k$ ("first" in the canonical ordering we referred to a long time ago). The STARS in each sequence tells us which variables are in the σ_i, and the δ acts as a way of getting at how the variables are set by σ_i by comparing these values in π.

We can then inductively retrieve the rest of the variables.

■

CS 15-855 Computational Complexity Theory **December 4, 2000**

Lecture 26: Approximation Method, II

Lecturer: Rudich *Scribe: Joshua Dunfield / Editor: EDITOR NAME*

Synopsis: Razborov-Smolensky: a circuit lower bound for mod$q$. Voting polynomials; weak and strong degree of functions. Aspnes et al.: a circuit lower bound for parity.

73 Recap

We are in the middle of establishing a pair of circuit lower bounds by means of the Approximation Method.

Theorem 73.1 (Razborov-Smolensky.) *For all distinct primes p and q: Using AND, OR, NOT and* mod$p$ *gates, we cannot approximate* mod$q$ *with constant error in fewer than $2^{\Omega(n^{1/(2k)})}$ gates, where k is the depth of the circuit.*

In the previous lecture, it was shown that every circuit of this kind can be approximated with constant error by a small ($o(\sqrt{n})$-degree) polynomial.

Theorem 73.2 (Aspnes, Beigel, Furst, Rudich.) AC^0 *circuits with a single "majority" gate at the root cannot approximate the parity function $\oplus$.*

In the previous lecture, we proved this theorem:

Theorem 73.3 *Every AC^0 circuit with a single majority gate at the root can be approximated to within ϵ by the sign of a polynomial of degree $O((\log(S^2/\epsilon)\log S)^d)$.*

74 Razborov-Smolensky

Theorem 74.1 *(Smolensky's theorem.) For all $p > 2$, no degree-$o(\sqrt{n})$ polynomial over $\mathbb{Z}_p$ can approximate parity with constant error.*

The proof of this theorem is rather involved, so we will not prove it in full generality. Instead, we will demonstrate that it holds when $p = 3$.

Encode *true* as -1 and $false$ as 1. Then $\oplus$ on n inputs $x_1, x_2, \ldots, x_n$ is just the product $x_1 x_2 \ldots x_n$. (This choice of encoding is fully general — we know from Hwk. 6 that the degree of the resulting polynomial is invariant over encodings of *true* and $false$.)

Let W be a set of true/false assignments to the n variables, and let $w = |W|$. Now let $P(x_1, \ldots, x_n)$ be a degree-d polynomial which agrees with $x_1 \ldots x_n$ on all true/false assignments *except* those in W. Thus w is the number of assignments for which P disagrees with $\oplus$. We will show that d and w cannot both be "small"—if P is of low degree, then P must disagree with many assignments, making it a poor approximation.

Lemma 74.2 *Every multi-linear polynomial $q = q_1 + \cdots + q_m$ can be represented in the following form:*

$$x_1 x_2 \ldots x_n \ell_1 + \ell_2$$

where the degrees of ℓ_1 and ℓ_2 are at most $n/2$.

Proof: Break q into $\ell_1' + \ell_2$ according to the degree of each term q_i:

$$\ell_1' = \{q_i \mid q_i \text{ has degree} \geq n/2\}$$
$$\ell_2 = \{q_i \mid q_i \text{ has degree} < n/2\}$$

Now let

$$\ell_1 = x_1 x_2 \ldots x_n \ell_1'$$

Since q is multi-linear, each variable x_i can appear in a term at most once. If x_i appears in ℓ_1, it appears squared in ℓ_1—but $true^2 = (-1)^2 = 1$ and $false^2 = 1^2 = 1$, so x_i does not appear in ℓ_1. On the other hand, if x_i does not appear in ℓ_1', it will

appear once in ℓ_1. Thus, if each term in ℓ_1' has degree d', $d' \geq n/2$, the degree of the corresponding term in ℓ_1 is $n - d'$. So the degree of ℓ_1 is at most $n/2$.

Finally,

$$\begin{aligned} x_1 x_2 \ldots x_n \ell_1 + \ell_2 &= x_1 x_2 \ldots x_n x_1 x_2 \ldots x_n \ell_1' + \ell_2 \\ &= x_1 x_1 x_2 x_2 \ldots x_n x_n \ell_1' + \ell_2 \\ &= \ell_1' + \ell_2 = q \ , \end{aligned}$$

so $x_1 \ldots x_n \ell_1 + \ell_2$ does in fact equal q. ∎

Corollary 74.3 *If we ignore the assignments in W, any multi-linear polynomial q can be represented as $p\ell_1 + \ell_2$, where ℓ_1 and ℓ_2 have degrees of no more than $n/2$.*

Proof: Represent q as $x_1 x_2 \ldots x_n \ell_1 + \ell_2$. Since p agrees with $x_1 \ldots x_n$ on every assignment not in W, we obtain

$$q = x_1 x_2 \ldots x_n \ell_1 + \ell_2 = p\ell_1 + \ell_2$$

for every assignment not in W. ∎

We are working in $\mathbb{Z}_3$, so a function can produce any of three values for a particular input. There are 2^n possible true/false assignments to n variables. Hence there are 3^{2^n} multi-linear functions. Excluding the assignments in W, there are $3^{2^n - w}$ distinct functions.

We represent each such function by a multi-linear polynomial q. By Corollary 74.3, q can be represented as $p\ell_1 + \ell_2$ (ignoring assignments in W). We chose p to have degree d, and by Lemma 74.2, $\ell_1 + \ell_2$ is of degree $\leq n/2$. Hence $q = p\ell_1 + \ell_2$ has degree $d + n/2$.

How many functions exist that have degree $d + n/2$? If there are less than $3^{2^n - w}$ we have a contradiction.

When $d = o(\sqrt{n})$, the number of monomials is

$$\sum_{i=0}^{n/2+d} \binom{n}{i} = 2^{n-1} + o(2^n)$$

so the number of functions of degree $\leq d + n/2$ is $3^{2^{n-1} + o(2^n)}$. We need the following inequality to hold:

$$3^{2^{n-1} + o(2^n)} \geq 3^{2^n - w}$$

But:

$$\begin{aligned} 3^{2^{n-1}+o(2^n)} &\geq 3^{2^n-w} \\ 2^{n-1}+o(2^n) &\geq 2^n - w \\ -2^{n-1}+o(2^n) &\geq -w \\ 2^{n-1}-o(2^n) &\leq w \\ \Theta(2^n) &\leq w \\ w &= \Omega(2^n) \end{aligned}$$

Thus the low-degree polynomial is wrong on a more-than-$o(1)$ fraction of the inputs, contradicting the assumption that$w$ was small.

75 Lower bound for $\oplus$ in $\mathbf{AC}^0$

75.1 Voting polynomials

We will use "voting polynomials" to approximate $\oplus$, the parity function.

Definition 75.1 *A* **voting polynomial** *is a polynomial over $\mathbb{R}$ that represents a Boolean function in the following way: the polynomial is positive wherever the function is 1 and negative wherever the function is 0.*

Definition 75.2 *The* **strong degree** SD_f *of a Boolean function f is the degree of the minimal-degree voting polynomial which represents it.*

For example, the strong degree of parity on n variables, $SD_\oplus$, is n.

75.2 Low-Degree Approximation of $\oplus$

What degree do we need to reasonably *approximate* a given function, specifically $\oplus$?

Suppose we have a degree k univariate polynomial f. Then $f(\Sigma_{i=1}^n x_i)$ has the correct sign when $\Sigma_{i=1}^n x_i$ is between $\frac{n-k}{2}$ and $\frac{n+k}{2}$ (see Fig. 20).

Consider all the possible input strings of length n. There are few strings such that the number of 1's is close to 0 or close to n. Our function $f(\Sigma_{i=1}^n x_i)$ is correct for the inputs with sums near the center of width, k, of the normal curve on Fig. 21.

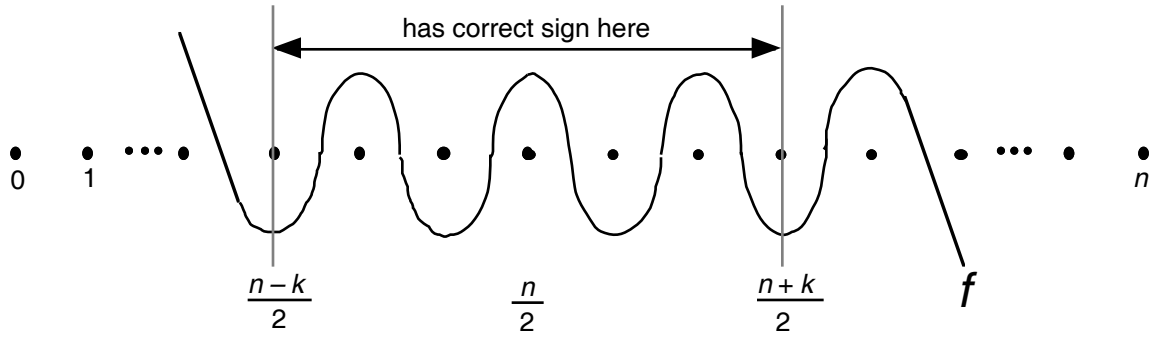


Figure 20: Graph of f, a degree k univariate polynomial.

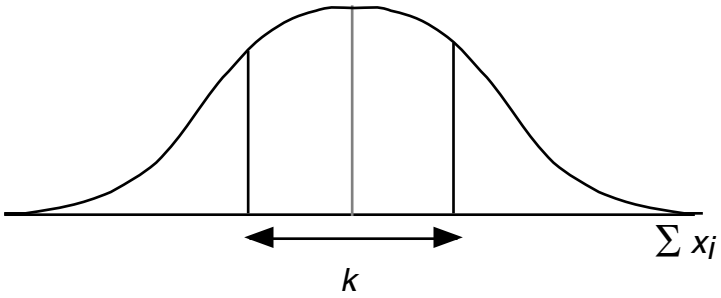


Figure 21: Distribution of Σx_i

So if k grows faster than $O(\sqrt{n})$, the function f approximates $\oplus$ well.

Note that the degree of $f(\Sigma_{i=1}^{n} x_i)$ is at most k.

Definition 75.3 *The* **weak degree** WD_f *of a Boolean function f is the degree of a minimal degree non-zero voting polynomial p which, for every assignment a_i to its variables, either has the correct sign (positive for true, negative for false)* or *equals 0. We say that such a polynomial p* **weakly expresses** *f.*

We exclude the all-zeros polynomial from the definition because such a polynomial is completely useless. (We still allow polynomials that are 0 at all but one place, but such polynomials must have high degree, making them uninteresting here.)

Suppose p weakly expresses $\oplus$. Observe that

1. $p \cdot \oplus > 0$, because each term in $\Sigma(p(x) \oplus x)$ is non-negative and we excluded

the all-zeros polynomial.

2. For any p of degree less than n, $p \cdot \oplus = 0$, because $\oplus$ is orthogonal to every monomial in p.

Therefore $WD_{\oplus} = n$.

75.3 Duality

Although this result is not used in the proof, it is interesting enough to be included here:

Theorem 75.1 Duality. $WD_f + SD_{f\oplus} = n$.

Suppose p weakly expresses f, and p' strongly expresses $f\oplus$. Then $p \times p'$ weakly expresses $\oplus$ (consider the possible signs). So the degree of $p \times p'$ is at least n.

$WD_f + SD_{f\oplus} \leq n$ can be shown using Farkas' lemma (not included).

75.4 Finishing the proof

Lemma 75.2 *There exists a "small" degree f' which is non-trivial but zero on all points in W.*

Proof: A degree k polynomial can have $\Sigma_{i=0}^{k}\binom{n}{i}$ coefficients. Choose k so that $\Sigma_{i=0}^{k}\binom{n}{i} > |W|$. Then choose the coefficients of f' so that f' sends all points in W to 0, i.e. choose coefficients according to a homogeneous system of $|W|$ equations in $v = \Sigma_{i=0}^{k}\binom{n}{i}$ variables. Since we chose k to be less than v, there is a non-trivial solution. ∎

Suppose a "low" degree f approximates $\oplus$ "very well". From this it will follow that $WD_{\oplus}$ is "low". Assume f is $\oplus$ on all but a "small" W. It follows from the above lemma that $(f')^2$ is a non-zero, "low" $(2k)$ degree polynomial that is zero on W (because f' is zero on W) and non-negative elsewhere. Furthermore, since f is $\oplus$ outside W, $f \cdot (f')^2$ weakly expresses $\oplus$ — inside W, $(f')^2 = 0$ so $f \cdot (f')^2 = 0$, and outside W the sign of $f \cdot (f')^2$ is the sign of f.

Theorem 75.3 *A degree k approximation to f will be wrong on at least*

$$\sum_{i=0}^{(WD_f-k-1)/2} \binom{n}{i}$$

points.

A function h is symmetric iff, for all x, y such that x and y are permutations of each other, $h(x) = h(y)$. Thus, a symmetric function depends only on the *number* of 1's in its input. Obviously, $\oplus$ is symmetric.

Corollary 75.4 *The best approximation to $\oplus$ is the symmetric scheme (Note: not true for other symmetric functions.)*

A proof of Corollary 75.4 can be found in Section 3 of [ABFR93].

Now, by Theorem 73.3, there is a voting polynomial of degree $O((\log(4s^2)\log s)^d) = O((\log s)^{2d})$ that approximates $\oplus$ except in at most $2^n/4$ instances. It follows from Theorem 75.3 that the polynomial's degree must be $\Omega(\sqrt{n})$:

$$\begin{aligned}(\log s)^{2d} &= \Omega(\sqrt{n}) \\ (\log s)^{2d} &\geq \sqrt{n} \\ \log s &\geq (n^{1/2})^{1/(2d)} = n^{1/(4d)} \\ s &= 2^{\Omega(n^{1/(4d)})}\ ,\end{aligned}$$

which is the result we originally set out to prove.

76 Concluding remarks

Some other known lower bounds:

1. For distinct primes p, q, $\bmod p^n$ can't be computed by constant-depth, sub-exponential size circuits with $\bmod q^n$ gates.
2. Majority can't be computed with $\bmod p$ gates.

Finally, if you'd like a Ph.D., answer this question:

Is there a constant-depth, sub-exponential size circuit built from AND, OR, NOT and $\bmod 6$ gates that can compute SAT?

References

[ABFR93] J. Aspnes, R. Beigel, M. Furst and S. Rudich. The expressive power of voting polynomials. http://www.cs.cmu.edu/~rudich/papers/voting.ps

CS 15-855 Computational Complexity Theory **DATE, 12/6/2000**

Lecture 28: Approaches to Barrier Problems

Lecturer: Rudich *Scribe: Ioannis Koutis / Editor: Ioannis Koutis*

Synopsis: Barriers in the efforts to show lower bounds. Even in surprisingly simple models of computation the best known lower bounds seem to be very weak; some of the simplest ways to state our ignorance. Valiant's barrier and open problems that might be the next step. A discussion about counter-factual, non-relativizing diagonalization approaches.

77 Simple models and Lower Bounds

What is the current status in lower bounds? Here are some different kinds of circuits

- Depth-3 circuits with majority gates
- Depth-3 circuits with $\wedge, \vee, \neg$ and + MOD6 gates
- Logarithmic depth linear size circuits.
- Depth-3 circuits with gates implementing $\times$ and + over a field F

These models may seem simple but no one knows how to show that a some function in NP is not computable in these models.

An interesting "biological" question. Our brain neurons are very slow, but we are quite fast in many tasks! So, if our brain is reasonably modeled by a circuit, the depth of our "circuit" can not be very big. Is this an indication that small depth circuits are very powerful? Or, is this related to our failure to get powerful lower bounds?

78 Valiant's barrier

Question: Can all of functions in FNP be computed by logarithmic depth linear size circuits? Our intuition is that this is not possible because the information can't be routed through the directed acyclic graph, which underlies the circuit.

The following lemma is in the core of Valiant's approach.

Lemma 78.1 *Let G be a directed, acyclic graph, with m edges, whose longest path has length d. It is possible to remove fewer than $km/\lceil \log d \rceil$ edges so that the resulting graph will have a longest path of length less than $d/2^{k-1}$.*

Before we proceed to the proof some definitions and propositions are needed.

Definition 78.1 *Let G be a directed acyclic graph. A depth function d is a function from the nodes of G to the set of naturals $[0 \ldots q]$ such that if there is an edge $u \to v$, then $d(u) < d(v)$.*

Notice that the definition includes the usual notion of depth, but it is broader. Two easy consequences of the definition capturing these observations are given in the following propositions.

Proposition: If G has a depth function $d \to G \to [0 \ldots q]$ then G has depth at most q.

Proposition: The function $d(v)$="the length of longest path to v", is a depth function onto $[0 \ldots D]$, where D is the usual depth of the graph.

We now give a sketch of the proof.

Proof sketch: Let $d(v)$ be the length of the longest path to v. Now do the following: (i) mark each node v using depth function $d(v)$ written in binary, using $\log d$ bits per node, (ii) mark each edge $u \to v$ by the index of the most significant bit in which $d(u)$ and $d(v)$ differ.

Notice that there are m edges and each edge is labeled with an index between 1 and $\lceil \log d \rceil$. By the pigeonhole principle it easily follows that there exists an index i that occurs in less than $n/\lceil \log d \rceil$ edges.

Now we do the following two steps: (i) **delete** all edges labeled with i from G, (ii) **adjust** the labels of the nodes, by deleting their i^{th} bit to obtain a new labeling $d' : G \to [0 \ldots \lceil \log d \rceil / 2]$.

Claim: The new labeling d' is a depth function. To prove this, we need to show that if $u \to v$ is an edge in G then $d'(u) < d'(v)$. This is in fact easy. Let Δ be the index of the most significant bit on which $d(u)$ and $d(v)$ differ. There are three cases: (i) if $\Delta = i$ then the edge $u \to v$ was deleted, (ii) if $\Delta < i$ then $d'(u)$ and $d'(v)$ are produced from $d(u)$ and $d(v)$ by truncating some identical least significant bits, so the property $d'(u) < d'(v)$ is "inherited" from $d(u) < d(v)$, (iii) similarly, if $\Delta > 0$ we can only have $d'(u) < d'(v)$ because otherwise we would have $d(u) > d(v)$.

The above two steps can be repeated iteratively k times, to remove at most $km/\lceil \log d \rceil$ edges and get a depth function into $[0 \ldots \lceil \log d \rceil / 2^k]$, from which it follows that the resulting graph has depth at most $d/2^{k-1}$. This concludes the proof. ∎

Corollary 78.2 *Let C be a bounded family of logarithmic depth $(c \log n)$ and linear size $(c'n)$ circuits. Let ϵ be any positive real. We can cut $O(n \log \log n)$ wires so that the resulting circuit has depth at most $\epsilon \log n$.*

Proof: We set $k = \log(c/\epsilon)+1$ (a constant), so that we cut at most $kO(n)/\log(c \log n) = O(n/\log \log n)$ wires. By Valiant's lemma depth has been cut by a factor of $2^k = c/\epsilon$. ∎

So, we can cut $O(n \log \log n)$ wires of C so that it has depth at most $\epsilon \log n$. It follows that each output of the circuit C depends on at most n^ϵ values, some inputs and some cut wires. So, for showing a lower bound, i.e. a function that can't be computed by logarithmic depth linear size circuit, it is enough to invent a function that depends on many inputs. Currently **no** such function is known. Are there any good candidates?

79 Interesting Open Problems

Definition 79.1 *The rigidity of a matrix A over a field F denoted by $R_A(r)$, is the number of entries of A that must be changed to reduce the rank below r.*

Valiant showed that almost all $n \times n$ matrices have rigidity $(n-r)^2$ over infinite fields, and $\Omega((n-r)^2/\log n)$, over finite fields. He also proposed the problem of finding an

explicit family of matrices A such that $R_A(\epsilon n) \geq n^{\delta n}$ for some $\delta > 0$. By his lemma and some additional work it can be shown that A_n can't be computed by a linear size logarithmic depth circuits with gates computing liner functions over F. Another relevant problem (which happens to be Avi Wigderson's current favorite).

Definition 79.2 *Define the gaussian complexity of a matrix A to be the number of row operations performed by the shortest sequence of operations that gets A into reduced form.*

Proposition: For most $n \times n$ matrices the gaussian complexity is $\Omega(n^2)$.

Open problem: Find an explicit family of matrices A_n such that the gaussian complexity of A_n is super-linear !

80 A simple way to state our ignorance

Definition 80.1 *A polynomial $f(x_1, \ldots, x_n)$ over $\mathbb{Z}_m$ represents the $OR(x_1, \ldots, x_n)$, if $\forall X \in \{0,1\}^n$, it is $OR(X) = 0 \Leftrightarrow f(X) = 0$. What is the lowest polynomial that represents the OR ?*

It turns out that if m is a prime and f represents the OR over $\mathbb{Z}_m$ then the degree of f is $\lceil n/(m-1) \rceil$. But this is not the case for m which is not a prime. If $m = 6$, OR can be represented by a $O(\sqrt{n})$ degree polynomial, as showed by Barrington, Beigel and Rudich in 1993. In 1994, Barrington and G. Tardos showed that the degree grows faster than $o(\log n)$. So, an open problem is, what is the degree of the OR over Z_6? It must be something in between $[\log n \ldots \sqrt{n}]$.

81 Discussion

As Baker, Gill, and Solovay observed in an 1975 paper, that all diagonalization proofs to date, relativize, but the major open problems of the field do not. For example there exist oracles A, B, such that $\mathsf{P}^A = \mathsf{NP}^A$ and $\mathsf{P}^B \neq \mathsf{NP}^B$. Does this mean that the diagonalization is the wrong approach?

Some classes turn out to be the same for non-relativizing reasons. For example we know that $\mathsf{IP} = \mathsf{PSPACE}$, but there exists an oracle A, such that $\mathsf{IP}^A \neq \mathsf{PSPACE}^A$.

But there have been counter-factual non-relativizing results that use diagonalization. We begin assuming that $classA = classB$, we obtain other non-relativizing corollaries for collapses, and then we point out that we have collapsed to classes known to be different by diagonalization. And, Fortnow and van Melkebeek showed the following theorem.

Theorem 81.1 *SAT can't be solved by a machine using $n^{o(1)}$ space and n^{α} time where $\alpha < \phi \approx 1.618$, where ϕ is exactly the golden ration.*

So, it is an open problem to give an updated version of the [BGS] paper that will allow us to understand why we can't prove $\mathsf{P} \neq \mathsf{NP}$ this way. But it still may possible to prove $\mathsf{NP} \neq \mathsf{L}$ by using counter-factual non-relativizing diagonalization.

Remark: This lecture gives at least two or three PhD topics!

CS 15-855 Computational Complexity Theory **November 6, 2000**

Lecture 29: Natural Proofs

Lecturer: Steven Rudich *Scribe: Abraham Flaxman / Editor: Pat Riley*

Synopsis: "Proof" that circuit complexity is hard: a meta-theoretic result in circuit complexity, which suggests why current approaches may not be able to prove important conjectures, such as $\not= \mathbf{NP}$.

82 Introduction

The previous lectures in this course have introduced and discussed many famous, difficult, open problems: $\not=$versus **NP** , $\not=$versus **PSPACE** , and $\not=$versus **NC** to name a few.

In this lecture we will argue that these problems are "difficult".

How can we argue that a mathematical question is difficult? One tempting way is to say "smart people have tried and failed." This is a very common and not very convincing argument. Another approach is to reduce the question to a famous claim, for example to prove that $\not= \mathbf{NP}$ implies the Riemann Hypothesis. Then you can argue "*for generations*, smart people have tried and failed." This lecture will take a different approach, by proving that the current techniques in circuit complexity are incapable of proving the desired theorems. A totally new technique could do the trick, but we have to invent it, and smart people have tried and failed.

This argument is analogous to the relativizing oracle argument of Baker, Gill, Solovay [BGS75], which showed that no relativizing proof can resolve $\not=$versus **NP** . Prior to [BGS75], all known lower bounds were based on simulation and diagonalization arguments. These arguments work for computation relative to any oracle. But $\not= = \mathbf{NP}$ relative to TQBF, and relative to a random oracle $\not= \mathbf{NP}$. If a relativizing argument resolved $\not=$versus **NP** one way or the other, it would be a problem for logic.

The Natural Proof theorem says that any circuit class incapable of computing functions with a "natural" combinatorial property (to be defined shortly) has no pseudo-random number generators. It is a constructive argument; any natural lower bound yields an algorithm to break pseudo-random function generators for that class. Thus

classes such as $\mathcal{I}$, $\mathbf{NC}^1$, and $\mathbf{TC}^0$ which probably have pseudo-random number generators probably don't have natural proofs against them.

The remainder of this lecture will proceed as follows. We will define natural combinatorial properties and what it means for a combinatorial property to be useful against a set complexity class. Then we will see how all known non-monotone, non-uniform circuit lower bounds fit into the natural proof framework. Finally we will prove the Natural Proof theorem and conclude with corollaries in conditional and unconditional complexity theory.

83 Definitions

For this talk, think of a function $f_n \colon \{0,1\}^n \to \{0,1\}$ as a binary string of length 2^n, called the *truth table* of f_n.

Definition 83.1 *The* **set of boolean functions** *on n bits, F_n is defined as,*

$$F_n = \left\{ f_n \;\middle|\; f_n \colon \{0,1\}^n \to \{0,1\} \right\}.$$

Definition 83.2 *A* **combinatorial property of boolean functions** *is a sequence of subsets of the set of boolean functions,*

$$\left\{ C_n \subseteq F_n \;\middle|\; n \in \mathbb{N} \right\}.$$

We will somtimes write $C_n(f_n) = 1$ to indicate $f_n \in C_n$. Thinking of membership in a combinatorial property as a function on functions is central to the reasoning behind the Natural Proof Theorem.

Definition 83.3 *Let λ be a complexity class. Then a combinatorial property C_n is* **useful against** *λ if for any sequence of functions $f_n \in C_n$,*

$$f_n \notin \lambda.$$

In less formal terms, a combinatorial property is useful against a complexity class if any function exhibiting the property is not in the complexity class.

These definitions allow us to formalize the framework for proving a circuit lower bound for any complexity class λ.

Definition 83.4 *A* **standard circuit lower bound argument** *is a proof of a circuit lower bound that looks like:*

1. *Define a combinatorial property, C_n.*
2. *Prove C_n is useful against λ.*
3. *Define a family of Boolean functions, $\{f_n\}$ and prove that $f_n \in C_n$ for all n.*
4. *Conclude that $\{f_n\} \notin \lambda$.*

An example helps make these definitions tangible. Let's see how this framework plays out in the restriction method for proving $\mathbf{AC}^0$ requires large circuits to compute parity.

The complexity class $\lambda = \mathbf{AC}^0$. The combinatorial property C_n is the set of functions f_n such that setting a certain number of inputs to constants does not make the function constant. The certain number of inputs decreases from paper to paper, starting high in [FSS81] and becoming sharply refined by [H86]. The lower the number the harder it is to prove that C_n is useful against λ. Finally, the parity function $\oplus_n$ is the prototypical "function which is not constant if you set many of it's inputs to constants", so $\oplus_n \in C_n$.

Definition 83.5 *A combinatorial property C_n is called* **natural** *if there exists a combinatorial property $C_n^* \subseteq C_n$ satisfying two conditions:*

- **Constructivity:** *$C_n^*(f_n)$ is computable in time polynomial in $|f_n| = 2^n$.*
- **Largeness:** $\frac{|C_n^*|}{|F_n|} \geq \frac{1}{2^{O(n)}}$.

It appears at first glance that these conditions are too liberal, constructivity allowing exponential time, and largeness demanding an exponentially small fraction. Don't worry. It all works out. The intuition is that we are working with truth tables of functions, which are exponential sized objects.

We can generalize this definition by restricting the type of computation we use in the constructivity condition.

Definition 83.6 *Let Γ be a complexity class. Then a combinatorial property is called* **Γ-natural** *if there exists a combinatorial property $C_n^* \subseteq C_n$ satisfying two conditions:*

- **Constructivity:** $C_n^*(f_n)$ *is computable in* Γ.
- **Largeness:** $\frac{|C_n^*|}{|F_n|} \geq \frac{1}{2^{O(n)}}$.

Definition 83.7 *A* **natural proof** *is a standard circuit lower bound argument that uses a natural combinatorial property of boolean functions.*

84 Examples

All known non-monotone, non-uniform circuit lower bounds have natural proofs. We will show this for three: $\mathbf{AC}^0$ bounds for parity, $\mathbf{AC}^0[3]$ bounds for parity, and voting polynomial bounds for parity,

84.1 $\mathbf{AC}^0$ bounds for parity

We commented above that in the $\mathbf{AC}^0$ lower bounds for parity, the combinatorial property C_n is the set of functions f_n such that setting a certain number of inputs to constants does not make the function constant. Now we will show that C_n is $\mathbf{AC}^0$-natural.

Theorem 84.1 C_n *is* $\mathbf{AC}^0$*-natural.*

Proof: Take $C_n^* = C_n$.

Largeness: The intuition is that the restriction of k inputs of a random function on n inputs is a random function on $n-k$ inputs. There are lots of functions and only 2 are constants.

We can formalize this with a little probabilistic calculation, but we have to get into the details of exactly how many inputs we fix with the restriction. It is left as an exercise to the reader.

Constructivity: f_n is a truth table with length 2^n. So we have $O(2^n)$ gates to work with. We can "just do it": list all of the $\binom{n}{k}2^{n-k} = 2^{O(n)}$ restrictions. Then for each restriction there is a depth 2 circuit of size $2^{O(n)}$ which decides if the restriction leaves f_n constant. ∎

Thus, this standard circuit lower bound argument is an $\mathbf{AC}^0$-natural proof:

1. $f_n \in C_n$ if there exists no restriction leaving the appropriate number of unassigned variables which forces f_n to a constant function.
2. C_n is useful against $\mathbf{AC}^0$.
3. $\oplus_n \in C_n$.
4. Therefore, $\oplus_n \notin \mathbf{AC}^0$.

84.2 $\mathbf{AC}^0[3]$ bounds for parity

Recall Smolensky's proof that $\oplus_n$ requires "large" $\mathbf{AC}^0[3]$ circuits [S87]. There are two parts to the argument. First we argue that any function computed by a "small" $\mathbf{AC}^0[3]$ circuit can be "reasonably approximated" by a "low" degree polynomial over $\mathbb{Z}_3$. Then we argue that $\oplus_n$ cannot be approximated by a "low" degree polynomial over $\mathbb{Z}_3$.

The combinatorial property C_n in this proof is the set of f_n which can't be "reasonably" approximated by a "low" degree polynomial. Smolensky's paper proves C_n is useful. We wish to prove it is natural. But to shed light on the art of proving a combinatorial property is natural, we will make two false starts before using the correct approach.

Theorem 84.2 *C_n is natural*

Proof Attempt 1: Let's try $C_n^* = C_n$. It worked last time.

We can prove largeness by a simple counting argument.

Unfortunately, no one know how to prove or disprove constructivity. It is an open problem.

Proof Attempt 2: We need to dig deep into Smolensky's proof to come up with a core combinatorial property that we can prove is constructible.

A key move was to use the fact that every polynomial can be written in the form $\oplus_n \cdot p_1 + p_2$, where p_1 and p_2 are low degree polynomials. Maybe we can use this as our core combinatorial property.

$f_n \in C_n^*$ if $f_n \in C$ and every polynomial over $\mathbb{Z}_3$ can be written in the form $\tilde{f}_n p_1 + p_2$ where p_1 and p_2 are low degree polynomials. (Here $\tilde{f}_n$ is the polynomial representation of f_n in $\mathbb{Z}_3$.)

With this C_n^*, we have constructibility. We can calculate the value of $C_n^*(f_n)$ by calculating the rank of a $2^n \times 2^n$ matrix, which is in $\mathbf{NC}^2$.

Unfortunately, we have achieved constructibility at too great a cost to largeness. It may be that there is a sufficient fraction of F_n exhibiting this property, but no one knows. It is another open question. But we're getting closer.

Proof: If we take another look into the bowels of Smolensky's proof at exactly what property he requires of $\oplus_n$, we get a C_n^* which works.

Intuitively, we take C_n^* to be the set of functions f_n for which "most" functions can be written as $f_n p_1 + p_2$ where p_1 and p_2 are low degree polynomials.

We can formalize this by saying $f_n \in C_n$ iff the vector space of polynomials spanned by $\tilde{f}_n L + L \geq \frac{3}{4}N$, where L is the set of low degree polynomials.

Largeness: $f_n \in C_n^*$ or $(\oplus_n \wedge f_n) \in C_n^*$, so $|C_n| \geq \frac{1}{2}|F_n|$.

Constructivity: We can calculate the rank of f_n using an $\mathbf{NC}^2$ algorithm, as before. ■

Therefore, the $\mathbf{AC}^0[3]$ bound is an $\mathbf{NC}^2$-natural proof. This result extends to cover the $\mathbf{AC}^0[q]$ bounds of [S87].

In the next section, we will show that there is no $\mathbf{AC}^0$-natural proof against $\mathbf{AC}^0[q]$. But now, one more example.

84.3 Voting polynomial bounds for parity

For constant depth circuits with AND, OR, and NOT gates, and a single majority gate directly above the output, [ABFR91] proves a lower bound on the number of

gates required to compute parity.

The combinatorial property C_n used in this argument is the set of f_n that cannot be approximated by the sign of a "low" degree polynomial over the reals.

Theorem 84.3 *C_n is natural.*

Proof: Take C_n^* to be the f_n with weak degree appropriately high.

Largeness is proved in the original paper [ABFR91].

Constructivity: In the previous lecture, we commented that the strong degree and weak degree are have a complimentary relationship: $\deg_s(f_n)+\deg_w(\oplus_n \wedge f_n) = n$. We can calculate the strong degree by linear programming, and obtain the weak degree by $\deg_w(f_n) = n - \deg_s(\oplus_n \wedge f_n)$. ∎

Since checking membership of f_n in C_n^* requires linear programming, we can only say that this argument is a $\mathcal{I}$-natural proof.

85 Natural Proof Theorem

Recall the definition of a Γ-secure pseudo-random function generator: G is a pseudo-random function generator secure against Γ-statistical tests, then for every statistical test $C_n \in \Gamma$, for every random seed,

$$\left| \Pr\left[C_n(\text{random function in } F_n)] = 1\right] - Pr\left[C_n(g_{\text{random seed}}) = 1\right] \right| < \frac{1}{2^{\omega(n)}}$$

We comment that pseudo-random function generators are know to exist for limited computation models, and suspected to exist for more powerful models.

Theorem 85.1 (Nisan, [N90]) *For all d, $\mathbf{AC}^0[2]$ contains a pseudo-random function generator secure against depth d $\mathbf{AC}^0$ circuits.*

Theorem 85.2 ([GGM90], [RR97]) *If $\mathcal{I}$/poly contains a 2^{n^ϵ}-hard function then $\mathcal{I}$/poly contains a pseudo-random function generator secure against $\mathcal{I}$/poly.*

Now we are ready to state and prove the Natural Proof Theorem.

Theorem 85.3 *If a complexity class λ contains a pseudo-random function generator G which is secure against Γ-statistical tests, then there is no Γ-natural combinatorial property useful against λ.*

Proof: Let λ be a complexity class containing a pseudo-random function generator secure against all Γ-statistical tests.

Assume for contradiction that C_n is a Γ-natural combinatorial property useful against λ. Let C_n^* be the Γ-computable core of C_n.

For seed s, G generates the pseudo-random function g_s. g_s is not in C_n because it is computable in λ and C_n is useful against λ. So $g_s \notin C_n^*$.

But a $1/2^{\mathrm{O}(n)}$ fraction of $f \in F_n$ are in C_n^* because of the largeness requirement.

So C_n^* is a Γ-statistical test that breaks G. Contradiction! ∎

86 Conclusion

There are a number of conditional and unconditional corollaries to the Natural Proof Theorem.

Corollary 86.1 *If I/poly contains a 2^{n^ϵ}-hard function (and we believe it does) then there is no natural proof that* `SAT` *requires super-polynomial circuits.*

Corollary 86.2 *No $\mathbf{AC}^0$-natural proof can work against $\mathbf{AC}^0[p]$.*

Corollary 86.3 *There is no natural proof that discrete logarithm requires large circuits.*

Corollary 86.4 *If there is a natural proof that some function in* **NP** *requires n^4 gates then there is a circuit for factoring smaller than currently known.*

Theorem 86.5 (Naor, Reingold [NR95]) *If factoring is suitably hard then $\mathbf{TC}^0$ contains sufficiently strong pseudo-random function generators.*

Corollary 86.6 *If factoring is suitably hard then there is no natural proof against* $\mathbf{TC}^0$.

But enough of that. Let's be philosophical. Why do natural proofs arise? In a non-uniform lower bound there is always a useful combinatorial property. Perhaps the reason the combinatorial property has a constructible core is because it's very hard to work with non-constructible objects. Algorithmic proof techniques are certainly a staple of complexity theoretic reasoning. We can do an even more convincing piece of hand-waving regarding why the combinatorial property is large.

We will define a generalized notion of complexity

Definition 86.1 *A* **formal complexity measure** $\mu : F_n \to \mathbb{N}$ *is a function such that*

1. $\mu(f) \leq 1$ *for "simple" functions like* $x_i, \neg x_i$,
2. $\mu(f \wedge g) \leq \mu(f) + \mu(g)$ *and* $\mu(f \vee g) \leq \mu(f) + \mu(g)$ *(putting functions together is not expensive).*

Theorem 86.7 *Let* μ *be a formal complexity measure with* $\mu(g) = t$ *for some* $g \in F_n$. *Then for at least* $\frac{1}{4}$ *of* $f \in F_n$, $\mu(f) \geq t/4$.

Proof: $g = (f \wedge (\neg f \oplus g)) \vee (\neg f \wedge (f \oplus g))$. $\mu(g) = t$ so one of the four terms on the right-hand side must have $\mu(t) \geq t/4$ by the pigeon-hole principle. ∎

So where do we stand for proving important theorems, like $\mathbf{I}$ versus $\mathbf{NP}$? We've known since the 70s that diagonalization and simulation proves too little to do the job. Now, the Natural Proof Theorem says that natural proofs prove too much to do the job. There is probably not even a natural proof separating $\mathbf{TC}^0$ from $\mathbf{NC}^1$.

What can we do? We can try returning to uniform computation. We can try to dream up combinatorial properties which violate largeness. We can try to dream up combinatorial properties which violate constructiveness. Or we can take the pessimistic route and invest in independence results, and try to show the problem is undecidable in ZFC or some suitably powerful axiomatic framework.

But let's end on a high note. It is also possible that we can return to diagonalization and make it work with circuit lower bound techniques. Perhaps a diagonalization argument with some non-relativizing step based on circuit theory can do the job.

References

[ABFR91] J. Aspnes, R. Beigel, M. Furst, S. Rudich. The expressive power of voting polynomials. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, 402-409, 1991.

[BGS75] T. Baker, J. Gill, and R. Solovay, Relativizations of the $\mathcal{P} =? \mathcal{NP}$ Question, *SIAM Journal of Computing*, 4(4):431-422, 1975.

[FSS81] M. Furst, J. Saxe, and M. Sipser. Parity, Circuits, and the Polynomial-Time Hierarchy. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, 260-270, 1981.

[H86] J. Hastad. Almost optimal lower bounds for small depth circuits. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, 6-20, 1986.

[GGM90] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4):792-807, 1986.

[NR95] M. Naor and O. Reingold. Synthesizers and their application to the parallel construction of pseudo-random functions. In *36th Annual Symposium on Foundations of Computer Science*, 170-181, 1995.

[N90] Noam Nisan. Pseudorandom generators for space-bounded computation. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, 204-212, 1990.

[RR97] A. Razborov and S. Rudich. Natural proofs. *Journal of Computer and System Sciences*, 55(1):24-35, 1997.

[S87] R. Smolensky. Algebraic methods in the theory of lower bounds for Boolean circuit complexity. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, 77-82, 1987.