CS278: Computational Complexity
Spring 2001

Luca Trevisan

These are scribed notes from a graduate course on Computational Complexity offered at the University of California at Berkeley in the Spring of 2001. The notes have been only minimally edited, and there may be several errors and imprecisions. I wish to thank the students who attended this course for their enthusiasm and hard work.

Berkeley, June 12, 2001.

Luca Trevisan

Contents

Chapter 1

Introduction

January 17, 2001,

This course assumes CS170, or equivalent, as a prerequisite. We will assume that the reader is familiar with the notions of algorithm and running time, as well as with basic notions of discrete math and probability.

A main objective of theoretical computer science is to understand the amount of resources (time, memory, communication, randomness , ...) needed to solve computational problems that we care about. While the design and analysis of algorithms puts upper bounds on such amounts, computational complexity theory is mostly concerned with lower bounds; that is we look for *negative results* showing that certain problems require a lot of time, memory, etc., to be solved. In particular, we are interested in *infeasible* problems, that is computational problems that require impossibly large resources to be solved, even on instances of moderate size. It is very hard to show that a particular problem is infeasible, and in fact for a lot of interesting problems the question of their feasibility is still open. Another major line of work in complexity is in understanding the relations between different computational problems and between different "modes" of computation. For example what is the relative power of algorithms using randomness and deterministic algorithms, what is the relation between worst-case and average-case complexity, how easier can we make an optimization problem if we only look for approximate solutions, and so on. It is in this direction that we find the most beautiful, and often surprising, known results in complexity theory.

Before going any further, let us be more precise in saying what a computational problem is, and let us define some important classes of computational problems. Then we will see a particular incarnation of the notion of "reduction," the main tool in complexity theory, and we will introduce **NP**-completeness, one of the great success stories of complexity theory.

Computational Problems

In a *computational problem*, we are given an *input* that, without loss of generality, we assume to be encoded over the alphabet $\{0, 1\}$, and we want to return in *output* a solution satisfying some property: a computational problem is then described by the property that the output has to satisfy given the input.

In this course we will deal with four types of computational problems: *decision* problems, *search* problems, *optimization* problems, and *counting* problems.[1] For the moment, we will discuss decision and search problem.

In a *decision* problem, given an input $x \in \{0,1\}^*$, we are required to give a YES/NO answer. That is, in a decision problem we are only asked to verify whether the input satisfies a certain property. An example of decision problem is the 3-coloring problem: given an undirected graph, determine whether there is a way to assign a "color" chosen from $\{1,2,3\}$ to each vertex in such a way that no two adjacent vertices have the same color.

A convenient way to *specify* a decision problem is to give the set $L \subseteq \{0,1\}^*$ of inputs for which the answer is YES. A subset of $\{0,1\}^*$ is also called a *language*, so, with the previous convention, every decision problem can be specified using a language (and every language specifies a decision problem). For example, if we call 3COL the subset of $\{0,1\}^*$ containing (descriptions of) 3-colorable graphs, then 3COL is the language that specifies the 3-coloring problem. From now on, we will talk about decision problems and languages interchangeably.

In a *search* problem, given an input $x \in \{0,1\}^*$ we want to compute some answer $y \in \{0,1\}^*$ that is in some relation to x, if such a y exists. Thus, a search problem is specified by a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$, where $(x,y) \in R$ if and only if y is an admissible answer given x.

Consider for example the search version of the 3-coloring problem: here given an undirected graph $G = (V,E)$ we want to find, if it exists, a coloring $c : V \to \{1,2,3\}$ of the vertices, such that for every $(u,v) \in V$ we have $c(u) \neq c(v)$. This is different (and more demanding) than the decision version, because beyond being asked to determine whether such a c exists, we are also asked to construct it, if it exists. Formally, the 3-coloring problem is specified by the relation R_{3COL} that contains all the pairs (G,c) where G is a 3-colorable graph and c is a valid 3-coloring of G.

P and NP

In most of this course, we will study the *asymptotic* complexity of problems. Instead of considering, say, the time required to solve 3-coloring on graphs with $10,000$ nodes on some particular model of computation, we will ask what is the best asymptotic running time of an algorithm that solves 3-coloring on all instances. In fact, we will be much less ambitious, and we will just ask whether there is a "feasible" asymptotic algorithm for 3-coloring. Here feasible refers more to the rate of growth than to the running time of specific instances of reasonable size.

A standard convention is to call an algorithm "feasible" if it runs in polynomial time, i.e. if there is some polynomial p such that the algorithm runs in time at most $p(n)$ on inputs of length n.

[1] This distinction is useful and natural, but it is also arbitrary: in fact every problem can be seen as a search problem

We denote by $\mathbf{P}$ the class of decision problems that are solvable in polynomial time.

We say that a search problem defined by a relation R is a NP search problem if the relation is efficiently computable and such that solutions, if they exist, are short. Formally, R is an $\mathbf{NP}$ search problem if there is a polynomial time algorithm that, given x and y, decides whether $(x, y) \in R$, and if there is a polynomial p such that if $(x, y) \in R$ then $|y| \le p(|x|)$.

We say that a decision problem L is an NP decision problem if there is some NP relation R such that $x \in L$ if and only if there is a y such that $(x, y) \in R$. Equivalently, a decision problem L is an NP decision problem if there is a polynomial time algorithm $V(\cdot, \cdot)$ and a polynomial p such that $x \in L$ if and only if there is a y, $|y| \le p(|x|)$ such that $V(x, y)$ accepts.

We denote by $\mathbf{NP}$ the class of NP decision problems.

Reductions

Let A and B be two decision problems. We say that A reduces to B, denoted $A \le B$, if there is a polynomial time computable function f such that $x \in A$ if and only if $f(x) inB$.

Two immediate observations: if $A \le B$ and B is in $\mathbf{P}$, then also $A \in \mathbf{P}$ (conversely, if $A \le B$, and $A \notin \mathbf{P}$ then also $B \notin \mathbf{P}$); if $A \le B$ and $B \le C$, then also $A \le C$.

NP-completeness

A decision problem A is $\mathbf{NP}$-hard if for every problem $L \in \mathbf{NP}$ we have $L \le A$. A decision problem A is $\mathbf{NP}$-complete if it is $\mathbf{NP}$-complete and it belongs to $\mathbf{NP}$.

It is a simple observation that if A is $\mathbf{NP}$-complete, then A is solvable in polynomial time if and only if $\mathbf{P} = \mathbf{NP}$.

An NP-complete problem

Consider the following decision problem, that we call U: we are given in input (M, x, t, l) where M is a Turing machine, $x \in \{0,1\}^*$ is a possible input, and t and l are integers encoded in unary[2], and the problem is to determine whether there is a $y \in \{0,1\}^*$, $|y| \le l$, such that $M(x, y)$ accepts in $\le t$ steps.

It is immediate to see that U is in $\mathbf{NP}$. One can define a procedure V_U that on input (M, x, t, l) and y accepts if and only if $|y| \le l$, and $M(x, y)$ accepts in at most t steps.

Let L be an NP decision problem. Then there are algorithm V_L, and polynomials T_L and p_L, such that $x \in L$ if and only if there is y, $|y| \le p_L(|x|)$ such that $V_L(x, y)$ accepts; furthermore V_L runs in time at most $T_L(|x| + |y|)$. We give a reduction from L to U. The reduction maps x into the instance $f(x) = (V_L, x, T_L(|x| + p_L(|x|)), p_L(|x|))$. Just by applying the definitions, we can see that $x \in L$ if and only $f(x) \in U$.

[2]The "unary" encoding of an integer n is a sequence of n ones.

Chapter 2

Cook's Theorem

January 22, 2001,

This lecture is devoted to the proof that SAT, a decision problem from logics, is **NP**-complete. This result is due to Cook [Coo71]. Cook introduced the notion of reductions and **NP**-completeness, recognized the relevance of the notion of **NP**-completeness, and proved the **NP**-completeness of SAT and of a few other problems. Independently, Levin [Lev73] also developed the notions of reductions and **NP**-completeness, and proved the **NP**-completeness of some natural problems. The **NP**-completeness of SAT is often referred to as *Cook's theorem* or *Cook-Levin's theorem*. Karp [Kar72] proved the **NP**-completeness of several problems, including several optimization problems. Karp's work pointed out the potential of **NP**-completenss as a way of explaining the hardness of an extreme variety of problems. This promise was quickly fulfilled, and **NP**-completeness results came by the hundreds during the 70s.

2.1 The Problem SAT

In SAT (that stands for *CNF-satisfiability*) we are given Boolean variables $x_1, x_2, \ldots, x_n$ and a Boolean formula ϕ involving such variables; the formula is given in a particular format called *conjunctive normal form*, that we will explain in a moment. The question is whether there is a way to assing Boolean (True / False) values to the variables so that the formula is satisfied.

To complete the description of the problem we need to explain what is a Boolean formula in conjunctive normal form. First of all, Boolean formulas are constructed starting from variables and applying the operators $\vee$ (that stands for OR), $\wedge$ (that stands for AND) and $\neg$ (that stands for NOT).

The operators work in the way that one expects: $\neg x$ is True if and only if x is False; $x \wedge y$ is True if and only if both x and y are True; $x \vee y$ is True if and only at least one of x or y is True.

So, for example, the expression $\neg x \wedge (x \vee y)$ can be satisfied by setting x to False and y to True, while the expression $x \wedge (\neg x \vee y) \wedge \neg y$ is impossible to satisfy.

A *literal* is a variable or the negation of a variable, so for example $\neg x_7$ is a literal and so is x_3. A *clause* is formed by taking one or more literals and connecting them with a OR, so for example $(x_2 \vee \neg x_4 \vee x_5)$ is a clause, and so is (x_3). A *formula in conjunctive normal form* is the AND of clauses. For example

$$(x_3 \vee \neg x_4) \wedge (x_1) \wedge (\neg x_3 \vee x_2)$$

is a formula in conjunctive normal form (from now on, we will just say "CNF formula" or "formula"). Note that the above formula is satisfiable, and, for example, it is satisfied by setting all the variables to TRUE (there are also other possible assignments of values to the variables that would satisfy the formula).

On the other hand, the formula

$$x \wedge (\neg x \vee y) \wedge \neg y$$

is not satisfiable, as it has already been observed.

2.2 Intuition for the Reduction

From now on, it will be convenient to think of Boolean variables as taking values in $\{0,1\}$, with the convention that 0 stands for FALSE and 1 stands for TRUE.

It may seem that CNF formulas can only specify very simple Boolean relations, however the following result holds.

LEMMA 1
For every function $f : \{0,1\}^k \to \{0,1\}$ there is a CNF formula ϕ over variables $x_1, \ldots, x_k$ such that an assignment $b_1, \ldots, b_k$ to $x_1, \ldots, x_k$ satisfies ϕ if and only $f(b_1, \ldots, b_k) = 1$. Furthermore, the size of ϕ is at most $k2^k$.

So it is possible to represent every function using a CNF formula, although the formula promised by the above lemma could be exponentially big in the number of variables. If we could guarantee the formula to be small for all efficiently computable f, then the **NP**-completeness of SAT would be quite easy to establish.

Let L be an **NP** decision problem. Then, by definition, there is an algorithm V_L and polynomials T and p such that for every x, we have that $x \in L$ if and only if there is a y, $|y| \leq p(|x|)$ such that $V_L(x, y)$ accepts; furthermore $V_L(x, y)$ runs at most in time $T(|x| + |y|)$. Now, fix an x, call $l = p(|x|)$, and consider the function $f_x : \{0,1\}^l \to \{0,1\}$ such that $f_x(y) = 1$ if and only if $V_L(x, y)$ accepts. Suppose we could produce a CNF formula ϕ_x over l variables such that ϕ_x is satisfied only by assignments that correspond to a y such that $f_x(y) = 1$. Then ϕ_x is satisfiable if and only if $x \in L$, and the transformation mapping x in ϕ_x would be a reduction from L to SAT. Since L was arbitrary, this argument would prove the **NP**-hardness of SAT.

Unfortunately, the function f_x could be such that there is no equivalent polynomial-size CNF formula, so it may be impossible to implement this strategy in polynomial time. In fact, even very simple functions have no short CNF formula representation, and the above strategy cannot possibly work, as shown by the next example.

LEMMA 2
Let ϕ be a formula over k variables, such that ϕ is satisfied precisely by those assignments where an odd number of variables takes value 1. Then ϕ has at least 2^{k-1} clauses.

2.3 The Reduction

Fortunately, there is a way to implement the general idea given in the previous section; unfortunately, there is some technical work to do.

Let L be an **NP** decision problem, and let $V(\cdot,\cdot)$ be the algorithm and T and p be the polynomials as above. We assume that V is implemented as a single-tape Turing machine M with semi-infinite tape.

Let us also fix an input x, and let $l = p(|x|)$ and $t = T(|x| + l)$.

Our goal is to construct, in time polynomial in $|x|$, a CNF formula ϕ_x such that ϕ_x is satisfiable if and only if $x \in L$, that is, if and only there is a y, $|y| \leq l$ such that $M(x, y)$ accepts within t steps.

We will construct ϕ_x so that its variables not only represent a possible choice of y, but also give a complete representation of the whole computation of $M(x, y)$. Let Q be the set of states of M, and let Σ be the alphabet of M. Since we want to represent computations of M that take time at most t, we know that only the first t entries of the tape are relevant for the description. The global state (also called a *configuration*) of the machine can be described by giving the position of the head on the tape (in one of the possible t positions), the current state of the machine (one of the $|Q|$ possible ones), and giving the content of the relevant part of the tape (a string in Σ^t). We will choose a more redundant representation, that will be easier to deal with. For each cell of the tape, we specify a pair (q, a), where $q \in Q \cup \{\square\}$ is the state of the machine, if the head is on that particular cell, or the special symbol $\square$ otherwise, and a is the content of the cell. So the configuration is represented by a string in $(\Sigma \times (Q \cup \{\square\}))^t$. A computation of length t can be represented by a sequence of $t + 1$ such configurations. Such a $t \times (t + 1)$ table of elements of $\Sigma \times (Q \cup \{\square\})$ will be called the *tableau* of a computation.

The variables of the formula ϕ_x are meant to represent a tableau. For every $i = 0, \ldots, t$, $j = 1, \ldots, t$, $q \in Q \cup \{\square\}$, $a \in \Sigma$, we have a variable $z_{i,j,q,a}$. The intended meaning of the variable is that the variable is true if and only if the tableau contains the pair (q, a) in the entry corresponding to time i and position j on the tape. Overall, we have $t \times (t + 1) \times |\Sigma| \times (|Q| + 1)$ variables.

The clauses of ϕ_x, that we will describe in a moment, enforce the property that every assignment to $z_{i,j,q,a}$ that satisfies ϕ_x encodes the tableau of a computation $M(x, y)$ where $|y| \leq l$, and M accepts at the end.

First of all, we need to enforce that the variables are giving a consistent description of the tableau, that is, that for each fixed i and j, there is exactly one pair (q, a) such that $z_{i,j,q,a}$ is true. This is enforced by the clauses

$$(\bigvee_{q,a} z_{i,j,q,a}) \text{ for all } i = 0, \ldots, t,\ j = 1, \ldots, t$$

and the clauses

$$(\neg z_{i,j,q,a} \vee \neg z_{i,j,q',a'}) \text{ for all } i = 0, \ldots, t,\ j = 1, \ldots, t,\ (q, a) \neq (q', a')$$

We also want to make sure that, in every line, exactly one cell contains a pair (q, a) where $q \neq \square$. We will use the clauses

$$(\bigvee_{q\neq\square,a,j} z_{i,j,q,a}) \text{ for all } i$$

and the clauses

$$(\neg z_{i,j,q,a} \vee \neg z_{i,j',q',a'}) \text{ for all } i, j \neq j', q, q' \in Q, a, a' \in \Sigma$$

Next, we want to enforce that the first line corresponds the a starting configuration where the tape contains (x, y) for some y of length l. Let q_0 be the initial state, let $n = |x|$, and let $x = (x_1, \ldots , x_n)$. We will use the clauses

$$z_{0,1,q_0,x_1}$$

$$z_{0,j,\square,x_i} \text{ for all } j = 2, \ldots , n$$

$$(z_{0,j,\square,0} \vee z_{0,j,\square,1}) \text{ for all } j = n + 1, \ldots , n + l$$

$$z_{0,j,\square,\square} \text{ for all } j = n + l + 1, \ldots , t$$

Then, we want to enforce that each line of the tableau is consistent with the previous line. Note that the content of the entry (i, j) of the tableau only depends on the entries $(i-1, j-1)$, $(i-1, j)$ and $(i-1, j+1)$ of the tableau (for $i \geq 2$). We can then use Lemma 1 to claim that for each i, j, $i \geq 2$, there is a CNF formula $\phi_{i,j}$ over the variables of the form $z_{i,j,q,a}$, $z_{i-1,j-1,q,a}$, $z_{i-1,j,q,a}$ and $z_{i-1,j-1,q,a}$ that is satisfied if and only if the variables encode a transition that is consistent with behaviour of M. Each such formula $\phi_{i,j}$ relates to $4 \times (|Q| + 1) \times |\Sigma| = O(1)$ variables, and so it is of size $O(1)$. We can construct all those formulas, and then take their AND, together with all the clauses described above.

Finally, we add the clause $\vee_{j,a} z_{t,j,q_A,a}$, where q_A is the accepting state of M.

The resulting formula is ϕ_x. Looking back at the description of the clauses in ϕ_x, the construction of ϕ_x can be done in time polynomial in $|x|$.

We claim that ϕ_x is satisfiable if and only if $x \in L$; this proves that L is reducible to SAT.

Suppose $x \in L$, then there is y, $|y| \leq l$ such that $M(x, y)$ accepts in at most t steps. Write down the tableau of the computation of $M(x, y)$, then compute the corresponding assignment to variables $z_{i,j,q,a}$. One can verify that such an assignment satisfies all the clauses described above.

Suppose we have an assignment of values $b_{i,j,q,a}$ to variables $z_{i,j,q,a}$ that satisfies ϕ_x. Construct the tableau corresponding to the values $b_{i,j,q,a}$. Since all clauses of ϕ_x are satisfied, the tableau describes a computation $M(x, y)$, for some y of length l, such that $M(x, y)$ accepts in at most t steps. Then it must be the case that $x \in L$.

Since L was arbitrary, we deduce that SAT is **NP**-hard. It's easy to see that SAT is in **NP**, so we proved that SAT is **NP**-complete.

2.4 The NP-Completeness of 3SAT

3SAT is the restriction of SAT to formulas where every clause is the OR of precisely 3 literals. A CNF formula where every clause contains exactly 3 literals is also called a 3CNF formula.

We give a reduction from SAT to 3SAT, thereby showing that 3SAT is NP-complete.

Let ϕ be a CNF formula, we want to create a new formula ϕ_3 such that ϕ is satisfiable if and only ϕ_3 is satisfiable; furthermore ϕ_3 contains only clauses with exactly 3 literals.

Note that it is not possible to construct ϕ_3 so that it has the same set of variables as ϕ, and is satisfied precisely by the same assignments. Consider for example the case where ϕ contains only the clause $(x_1 \vee x_2 \vee x_3 \vee x_4)$. There is no 3CNF formula over variables x_1, x_2, x_3, x_4 that has the same set of satisfying assignments as ϕ.

Starting from an arbitrary CNF formula ϕ, we will construct ϕ_3 by substitution each clause in ϕ that contains 1, 2, 4 or more literals by a small 3CNF formula, involving new variables. Each substitution will preserve satisfiability.

A clause of the form (x) is replaced by

$$(x \vee y_1 \vee y_2) \wedge (x \vee \neg y_1 \vee y_2) \wedge (x \vee y_1 \vee \neg y_2) \wedge (x \vee \neg y_1 \vee \neg y_2)$$

where y_1, y_2 are new variables.

A clause of the form $(x_1 \vee x_2)$ is replaces by

$$(x_1 \vee x_2 \vee y) \wedge (x_1 \vee x_2 \vee \neg y)$$

where y is a new variable.

A clause of the form $(x_1 \vee \cdots \vee x_k$, $k \geq 4$ is replaced by

$$(x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee x_3 \vee y_2) \wedge \cdots \wedge (\neg y_{k-3} \vee x_{k-1} \vee x_k)$$

where $y_1, \ldots, y_{k-3}$ are new variables.

One should check that each of these substitutions do indeed preserve satisfiability, and so that the combination of all these substitutions give a reduction that produces a formula ϕ_3 that is satisfiable if and only if ϕ is satisfiable.

Chapter 3

More NP-complete Problems

January 24, 2001, Scribe: John Leen

In this lecture we will wrap up our discussion of **NP**-completeness by studying a new **NP**-complete problem, Independent Set, and showing how 3SAT reduces to it. Then we will consider non-deterministic models of computation and their relation to **NP**. Finally, we will introduce some space-bounded complexity classes and examine their relation to the time-bounded classes.

3.1 Independent Set is NP-Complete

We have already seen the proof that SAT and 3SAT are **NP**-complete. Now we will look at another **NP**-complete problem, Independent Set (IS). We can state problem IS as follows: given a graph $G = (V, E)$ and an ingeger $k > 0$, is there an independent set in G of size k? (A set of vertices $I \subseteq V$ is *independent* if no two vertices $u, v \in I$ are connected by an edge $(u, v) \in E$.)

There is an optimization version of this problem, which is to find the largest possible independent set of a given graph, but we are simply interested in the decision problem IS given above. We will show that 3SAT reduces to IS, hence IS is **NP**-complete.

THEOREM 3
3SAT $\leq$ IS.

PROOF: Say we have a problem in 3SAT which is a boolean expression ϕ with n variables $x_1, x_2, \ldots, x_n$, and m clauses. Construct a graph G with $3m$ vertices, one for each literal (an occurance of a variable in a clause). Add edges to connect vertices corresponding to literals belonging to the same clause, so that for each clause we have a triangle of vertices connecting its literal. Finally, add edges to connect all pairs of vertices which are negations of each other.

We claim that 3SAT for ϕ corresponds to IS for (G, m). First, suppose there *is* an independent set I of size m. Then it must have exactly one vertex from each triangle, since there are m triangles, and having two vertices from the same triangle would violate independence. Let $x_i = true$ if at least one x_i vertex is in I, $x_i = false$ if at least one $\overline{x_i}$ is in I, and $x_i = true$ if neither condition holds. (The first two conditions cannot both occur for the same x_i, since we constructed edges between all contradictory pairs of literals.) By construction, this satisfies ϕ, because for each clause we have made at least one of the three assignments that will satisfy it.

Conversely, if ϕ is satisfiable, pick a satisfying assignment of variables, and define I to contain, for each triangle, one arbitrarily chosen literal satisfied by the assignment. (At least one must exist, or else ϕ would not be satisfied!) This I is independent since edges only connect literals within the same clause (of which only one is in I) and literals which are negations of each other (of which only one would be consistent with our satisfying variable assignment).

This proves that ϕ can be satisfied if and only if G has an independent set of size m. G can obviously be constructed from ϕ in polynomial time. Therefore 3SAT $\leq$ IS. □

3.2 Nondeterministic Turing Machines

A Turing machine is usually described by a set of states Q, an alphabet Σ, and a state transition function

$$\delta : Q \times \Sigma \to Q \times \Sigma \times \{L, S, R\}$$

which, given the state of the machine and the symbol at the current position of the tape, tells the machine a new state to go to, a new symbol to write, and a direction to move in (left, right, or stay in the same place). A *nondeterministic* Turing machine instead has a function

$$\delta : Q \times \Sigma \to 2^{Q \times \Sigma \times \{L,S,R\}}$$

which prescribes multiple possible things for the machine to do next. We say that a nondeterministic machine M solves a problem L if for every $x \in L$, there is a possible computation of $M(x)$ leading to acceptance, and for every $x \notin L$, all computations of $M(x)$ reject.

Theorem 4
NP *is the set of decision problems solvable in polynomial time by a nondeterministic Turing machine.*

Proof: Consider $L \in \mathbf{NP}$. We know there is an algorithm $V_L(\cdot, \cdot)$ running in polynomial time $T_L(\cdot)$ and a polynomial $p(\cdot)$ such that

$$x \in L \Leftrightarrow \exists y, |y| \leq p(|x|) \text{ and } V_L(x, y) \text{ accepts}$$

We can construct a machine M that uses nondeterminism to write all possible strings $y \in \{0,1\}^{p(|x|)}$ to its tape and then simulate $V_L(x, y)$, essentially testing all possible solutions in parallel. Thus this nondeterministic machine solves L in polynomial time.

Conversely, if L is solved in polynomial time by a nondeterministic machine M, define $V(\cdot, \cdot)$ that takes as input a value x that would be given as input to M and a description of a computational path of $M(x)$ leading to acceptance. V accepts if this description is valid, and runs in polynomial time. Thus $L \in \mathbf{NP}$, establishing our result. □

3.3 Space-Bounded Complexity Classes

A machine solves a problem using space $s(\cdot)$ if for every input x, the machine outputs the correct answer and uses only the first $s(|x|)$ cells of the tape. For a standard Turing machine, we can't do better than linear space since x itself must be on the tape! So we will often consider a machine with multiple tapes: a read-only "input" tape, a read/write "work" or "memory" tape, and possibly a write-once "output" tape. Then we can say the machine uses space s if for input x, it uses only the first $s(|x|)$ cells of the work tape.

We denote by ł the set of decision problems solvable in $O(\log n)$ space. We denote by **PSPACE** the set of decision problems solvable in polynomial space.

Theorem 5
If a machine always halts, and uses $s(\cdot)$ space, with $s(n) \geq \log n$, then it runs in time $2^{O(s(n))}$.

Proof: Call the "configuration" of a machine M on input x a description of the state of M, the position of the input tape, and the contents of the work tape at a given time. Write down $c_1, c_2, \ldots, c_t$ where c_i is the configuration at time i and t is the running time of $M(x)$. No two c_i can be equal, or else the machine would be in a loop, since the c_i completely describes the present, and therefore the future, of the computation! Now the number of *possible* configurations is simply the product of the number of states, the number of positions on the input tape, and the number of possible contents of the work tape (which itself depends on the number of allowable positions on the input tape). This is

$$O(1) \cdot n \cdot |\Sigma|^{s(n)} = 2^{O(s(n)) + \log n} = 2^{O(s(n))}$$

Since we cannot visit a configuration twice during the computation, the computation must therefore finish in $2^{O(s(n))}$ steps. □

NL is the set of decision problems solvable by a non-deterministic machine using $O(\log n)$ space. **NPSPACE** is the set of decision problems solvable by a non-deterministic machine using polynomial space.

Analogously with time-bounded complexity classes, we could think that **NL** is exactly the set of decision problems that have "solutions" that can verified in log-space. If so, **NL** would be equal to **NP**, since there is a log-space algorithm V that verifies solutions to SAT. However, this is unlikely to be true, because of the surprising fact that **NL** is contained in **P**! Intuitively, not all problems with a log-space "verifier" can be simulated in **NL**, because an **NL** machine does not have enough memory to keep track of all the choices that it makes.

Theorem 6
$\mathbf{NL} \subseteq \mathbf{P}$.

Proof: Let L be a language in **NL** and let M be a non-deterministic log-space machine for L. Consider a computation of $M(x)$. As before, at any time there are $2^{O(s(n))} = n^{O(1)}$ possible configurations. Consider a directed graph in which vertices are configurations and edges indicate transitions from one state to another which the machine is allowed to make in a single step (as determined by its δ). This graph has polynomially many vertices, so in polynomial time we can do a depth-first search to see whether there is a path from the initial configuration that eventually leads to acceptance. This describes a polynomial-time algorithm for deciding L, so we're done. $\square$

Chapter 4

Savitch's Theorem

January 29, 2001, Allison Coates

Today's lecture continues the discussion of space complexity. We examine **NL**, the set of languages decidable in $O(\log n)$ space on a nondeterministic Turing machine, introduce log-space reducibility and define nl-completeness. Finally, we discuss Savitch's theorem, and state the main result to be proved next time: $\mathbf{NL} = \mathbf{coNL}$.

4.1 Reductions in NL

Last time, we introduced **NL**, the class of languages decidable in logarithmic space on a nondeterministic Turing Machine: $\mathbf{NL} = \mathbf{NSPACE}(\log n)$. Now we would like to introduce a notion of completeness in **NL** analogous to the notion of completeness that we have introduced and studied for the class **NP**. A first observation is that, in order to have a meaningful notion of completeness in **NL**, we cannot use polynomial-time reductions, otherwise any **NL** problem having at least a YES instance and at least a NO instance would be trivially **NL**-complete. To get a more interesting notion of **NL** completeness we need to turn to weaker reductions. In particular, we define *log space* reductions as follows:

Definition 1 *Let A and B be decision problems. We say A is log space reducible to B, $A \leq_{\log} B$, if $\exists$ a function f computable in log space such that $x \in A$ iff $f(x) \in B$, and $B \in L$.*

Theorem 7
If $B \in \mathbf{L}$, and $A \leq_{\log} B$, then $A \in \mathbf{L}$.

Proof: We consider the concatenation of two machines: M_f to compute f, and M_B to solve B. If our resource bound was polynomial time, then we would use $M_f(x)$ to compute $f(x)$, and then run M_B on $f(x)$. The composition of the two procedures would given an algorithm for A, and if both procedures run in polynomial time then their composition is also polynomial time. To prove the theorem, however, we have to show that if M_f and M_B are log space machines, then their composition can also be computed in log space.

Recall the definition of a Turing machine M that has a log space complexity bound: M has one read-only input tape, one write-only output tape, and uses a log space work tape. A naive implementation of the composition of M_f and M_B would be to compute $f(x)$, and then run M_B on input $f(x)$; however $f(x)$ needs to be stored on the work tape, and this implementation does not produce a log space machine. Instead we modify M_f so that on input x and i it returns the i-th bit of $f(x)$ (this computation can still be carried out in logarithmic space). Then we run a simulation of the computation of $M_B(f(x))$ by using the modified M_f as an "oracle" to tell us the value of specified positions of $f(x)$. In order to simulate $M_B(f(x))$ we only need to know the content of one position of $f(x)$ at a time, so the simulation can be carried with a total of $O(\log |x|)$ bits of work space. □

Using the same proof technique, we can show the following:

THEOREM 8
if $A \leq_{\log} B, B \leq_{\log} C$, *then* $A \leq_{\log} C$.

4.2 NL Completeness

Armed with a definition of log space reducible, we can define **NL**-completeness.

DEFINITION 2 *A is* **NL***-hard if* $\forall\, B \in \mathbf{NL}, B \leq_{\log} A$. *$A$ is* **NL***-complete if* $A \in \mathbf{NL}$ *and A is* **NL***-hard.*

We now introduce a problem STCONN (s,t-connectivity) that we will show is **NL**-complete. In STCONN, given in input a directed graph $G(V, E)$ and two vertices $s, t \in V$, we want to determine if there is a directed path from s to t.

THEOREM 9
STCONN is **NL***-complete.*

PROOF:

1. STCONN $\in$ **NL**.

   On input *G(V,E), s,t*, set p to s. For $i = 1$ to $|V|$, nondeterminsitically, choose a neighboring vertex v of p. Set $p = v$. If $p = t$, accept and halt. Reject and halt if the end of the *for* loop is reached. The algorithm only requires $O(\log n)$ space.

2. STCONN is **NL**-hard.

   Let $A \in \mathbf{NL}, and let M_A$ be a non-deterministic logarithmic space Turing Machine for A. On input x, construct a directed graph G with one vertex for each configuration of $M(x)$, and an additional vertex t. Add edges (c_i, c_j) if $M(x)$ can move in one step from c_i to c_j. Add edges (c, t) from every configuration that is accepting, and let s be the start configuration. M accepts x iff some path from s to t exists in G. The above graph can be constructed from x in log space, because listing all nodes requires $O(\log n)$ space, and testing valid edges is also easy.

□

4.3 Savitch's Theorem

What kinds of tradeoffs are there between memory and time? STCONN can be solved deterministically in linear time and linear space, using depth-first-search. Is there some sense in which this is optimal? Nondeterministically, we can search using less than linear space. Can searching be done deterministically in less than linear space?

We will use Savitch's Theorem to show that STCONN can be solved deterministically in $O(\log^2 n)$, and that every **NL** problem can be solved deterministically in $O(\log^2 n)$ space. In general, if A is a problem that can be solved nondeterministically with space $s(n) \geq \log n$, then it can be solved deterministically with $O(s^2(n))$space.

Theorem 10
Every problem in **NL** *can be solved deterministically in* $O(\log^2 n)$ *space.*

Proof: Consider a graph $G(V, E)$, and vertices s, t. We define a recursive function REACH(u, v, k) that accepts and halts iff v can be reached from u in $\leq k$ steps. If $k = 1$, then REACH accepts iff (u, v) is an edge. If $k \geq 2, \forall w \in V - \{u, v\}$, compute REACH$(u, w, \lfloor \mathrm{k}/2 \rfloor)$ and REACH$(w, v, \lceil \mathrm{k}/2 \rceil)$. If both accept and halt, accept. Else, reject.

Let $S(k)$ be the worst-case space use of REACH$(\cdot, \cdot, k)$. The space required for the base case $S(1)$ is a counter for tracking the edge, so $S(1) = O(\log n)$. In general, $S(k) = O(\log n) + S(k/2)$ for calls to REACH and for tracking w. So, $S(k) = O(\log k * \log n)$. Since $k \leq n$, the worst-case space use of REACH is $O(\log^2 n)$. $\square$

Comparing to depth-first-search, we find that we are exponentially better in space requirements, but we are no longer polynomial in time.

Examining the time required, if we let $t(k)$ be the worst-case time used by REACH$(\cdot, \cdot, k)$, we see $t(1) = O(n + m)$, and $t(k) = n(2 * T(k/2))$, which solves to $t(k) = n^{O(\log k)} = O(n^{O(\log n)})$, which is super-polynomial. Savitch's algorithm is still the one with the best known space bound. No known algorithm achieves polynomial log space and polynomial time simultaneously. However, there are interesting results for STCONN in undirected graphs. There exists an algorithm running in polynomial time and $O(\log^2 n)$ space (but the polynomial has very high degree), due to Nisan [Nis94]. There is also an algorithm that has $O(\log^{4/3} n)$ space complexity and superpolynomial time complexity, due to Armoni, Ta-Shma, Nisan and Wigderson [ATSWZ97], improving on a previous algorithm by Nisan, Szemeredy and Wigderson [NSW92].

Theorem 11 (Savitch's Theorem)
For every function $s(n)$ computable in space $O(s(n))$, $\mathbf{NSPACE}(s) = \mathbf{SPACE}(O(s^2))$

Proof: We begin with a nondeterministic machine M, which on input x uses $s(|x|)$ space. We define $REACH(c_i, c_j, k)$, as above, which accepts and halts iff $M(x)$ can go from c_i to c_j in $\leq k$ steps. We compute $REACH(c_0, c_{\mathrm{acc}}, 2^O(s|x|))$ for all accepting configurations c_{acc}. If there is a call of REACH which accepts and halts, then M accepts. Else, M rejects. If REACH accepts and halts, it will do so in $\leq 2^{O(|x|)}$ steps.

Let $S_R(k)$ be the worst-case space used by REACH$(\cdot, \cdot, k)$: $S_R(1) = O(s(n)), S_R(k) = O(s(n)) + S_R(k/2)$. This solves $S_R = s(n) * \log k$, and, since $k = 2^O(s(n))$, we have $S_R = O(s^2(n))$. $\square$

4.4 coNL

We now show interesting results for **NL** and **coNL**. Let $\overline{A}$ denote the complement of A — that is, if A is a set of strings for which M_A accepted, then $\overline{A}$ consists of exactly those strings for which M_A did not accept. For a complexity class $\mathcal{C}$, we define the *complement class* $\mathbf{co}\mathcal{C}$ as follows: a decision problem $A \in \mathbf{co}\mathcal{C}$ iff $\overline{A} \in \mathcal{C}$.

Observe that, for deterministic classes, the complement of a class is the class itself. However, for nondeterministic classes, complementarity involves an exchange of the quantification: one goes from $\exists$ something satisfying such a condition to $\forall$ cases, there is nothing satisfying that condition. If $A \in \mathbf{NP}$, $X \in A \Leftrightarrow y, |y| \leq p(x).V(x, y)$ accepts. If $A \in \mathbf{coNP}$, $x \in A \Leftrightarrow \forall\, y, |y| \leq p(x).V(x, y)$ rejects. It considered very unlikely that $\mathbf{NP} = \mathbf{coNP}$, however the situation is different for space-bounded nondeterministic classes. The following theorem is due to Immerman [Imm88] and Sezelepscenyi [Sze88], and we will prove it next time.

THEOREM 12
$\mathbf{NL} = \mathbf{coNL}$

For the time being, let us see how the theorem reduces to finding a nondeterminisitc log space procedure for the complement of the STCONN problem.

LEMMA 13
Suppose $\overline{STCONN} \in \mathbf{NL}$*, then* $\mathbf{NL} = \mathbf{coNL}$.

PROOF: Consider a generic problem A in **coNL**. $\overline{A} \in \mathbf{NL}$, so $\overline{A} \leq_{\log} STCONN$. This means that there is a function f computable in logarithmic space such that $x \in \overline{A} \Leftrightarrow f(x) \in STCONN$. But then we also have $x \in A \Leftrightarrow f(x) \in \overline{STCONN}$, and so $A \leq_{\log} \overline{STCONN}$. Since, by assumption, we have $\overline{STCONN} \in \mathbf{NL}$, and **NL** is closed under logarithmic space reductions, we conclude $A \in \mathbf{NL}$, and since A was arbitrary, we have $\mathbf{coNL} \subseteq \mathbf{NL}$.

Take now any problem in **NL**: $A \in \mathbf{NL}$, hence $\overline{A} \in \mathbf{coNL}$ but by above, $\overline{A} \in \mathbf{NL}$, $A \in \mathbf{coNL}$, so $\mathbf{NL} \subseteq \mathbf{coNL}$. $\Box$

Chapter 5

NL=coNL, and the Polynomial Hierarchy

January 31, 2001, Scribe: Chris Harrelson

Today we will prove that **NL** = **coNL** and discuss the polynomial hierarchy.

5.1 NL = coNL

In order to prove that these two classes are the same, we will show that there is an **NL** Turing machine which solves $\overline{\text{STCONN}}$. $\overline{\text{STCONN}}$ is the problem of deciding, given a directed graph G, together with special vertices s and t, whether t is *not* reachable from s. Note that $\overline{\text{STCONN}}$ is **coNL**-complete.

Once we have the machine, we know that **coNL** $\subseteq$ **NL**, since any language A in **coNL** can be reduced to $\overline{\text{STCONN}}$, and since $\overline{\text{STCONN}}$ has been shown to be in **NL** (by the existence of our machine), so is A. Also, **NL** $\subseteq$ **coNL**, since if $\overline{\text{STCONN}} \in$ **NL**, by definition STCONN $\in$ **coNL**, and since STCONN is **NL**-complete, this means that any problem in **NL** can be reduced to it and so is also in **coNL**. Hence **NL** = **coNL**.

5.1.1 A simpler problem first

Now all that remains to be shown is that this Turing machine exists. First we will solve a simpler problem than $\overline{\text{STCONN}}$. We will assume that in addition to the usual inputs G, s and t, we also have an input r, which we will assume is equal to the number of vertices reachable from s in G, including s.

Given these inputs, we will construct a non-deterministic Turing machine which decides whether t is reachable from s by looking at all subsets of r vertices in G, halting with YES if it sees a subset of vertices which are all reachable from s but do not include t, and halting with NO otherwise. Here is the algorithm:

input: $G = (V, E)$, s, t, r
output: YES if it discovers that t is not reachable from s, and NO otherwise

assumption: there are exactly r distinct vertices reachable from s

$c := 0$
for all $v \in (V - \{t\})$ **do**
  non-deterministically guess if v is reachable from s in k steps
  if guess = YES **then**
    $p := s$
    for $i := 1$ to k **do**
      non-deterministically pick a neighbor q of p
      p := q
    if p is not equal to v, reject
    $c := c + 1$

if $c = r$ **then** return YES, otherwise return NO

Exercise: verify that this algorithm is indeed in **NL**.

Notice that in the algorithm above, c can only be incremented for a vertex v that is actually reachable from s. Since there are assumed to be exactly r such vertices, c can be at most r at the end of the algorithm, and if it is exactly r, that means that there are r vertices other than t which are reachable from s, meaning that t by assumption cannot be reachable form s. Hence the algorithm accepts if and only if it discovers that t is not reachable from s.

5.1.2 Finding r

Now we need to provide an **NL**-algorithm that finds r. Let's first try this algorithm:

input: $G = (V, E)$, s
output: the number of vertices reachable from s (including s in this count)

$c := 0$
for all $v \in V$ **do**
  non-deterministically guess if v is reachable from s in k steps
  if guess = YES **then**
    $p := s$
    for $i := 1$ to k **do**
      non-deterministically guess a neighbor q of p (possibly not moving at all)
      $p := q$
    if $p \neq v$ reject
    $c := c + 1$

return c

This algorithm has a problem. It will only return a number c which is at most r, but we need it to return *exactly* r. We need a way to force it to find all vertices which are reachable from s. Towards this goal, let's define r_i to be the set of vertices reachable from s in at most i steps. Then $r = r_{n-1}$, where n is the number of vertices in G. The idea is to try to compute r_i from r_{i-1} and repeat the procedure $n-2$ times. Now here is another try at an algorithm:

input: $G = (V, E)$, s, i, r_{i-1}
output: the number of vertices reachable from s in at most i steps (including s in this count)
assumption: r_{i-1} is the exact number of vertices reachable from s in at most $i-1$ steps

```
c := 0
for all v ∈ V do
        d := 0
        for all w ∈ V do
                non-deterministically guess if w is reachable from s in at most k − 1 steps
                if guess = YES then
                  p := s
                  for i := 1 to k − 1 do
                      non-deterministically pick a neighbor q of p (possibly not moving at all)
                      p := q
                  if p ≠ v then reject
                  d := d + 1
                  if v is a neighbor of w, or if v = w then
                    c := c + 1
                    break out of the inner loop and start the next iteration of the outer loop
        if d < r_{i−1} reject

return c
```

Here is the idea behind the algorithm: for each vertex v, we need to determine if it is reachable from s in at most i steps. To do this, we can loop over all vertices which are a distance at most $i-1$ from s, checking to see if v is either equal to one of these vertices or is a neighbor of one of them (in which case it would be reachable in exactly i steps). The algorithm is able to force all vertices of distance at most $i-1$ to be considered because it is given r_{i-1} as an input.

Now, putting this algorithm together with the first one listed above, we have shown that $\overline{\text{STCONN}} \in \mathbf{NL}$, implying that $\mathbf{NL} = \mathbf{coNL}$. In fact, the proof can be generalized to show that if a decision problem A is solvable in non-deterministic space $s(n) = \Omega(\log n)$, then $\overline{A}$ is solvable in non-deterministic space $O(s(n))$.

5.2 The polynomial hierarchy

One way to look at the difference between **NP** and **coNP** is that a decision problem in **NP** is asking a sort of "does there exist" question, where the existence of the answer can by definition be efficiently represented. On the other hand, **coNP** asks "is it true for all" questions, which do not seem to have simple, efficient proofs. In this way, complexity theorists think of there being a $\exists$ associated with **NP** and a $\forall$ associated with **coNP**.

5.2.1 Stacks of quantifiers

Now suppose you had a decision problem A which asked something of the following form:

$$x \in A \Leftrightarrow \exists\, y_1 \text{ s.t. } |y_1| \leq p(|x|)\ \forall\, y_2 \text{ s.t. } |y_2| \leq p(|x|)\ V(x, y_1, y_2)$$

In other words, a Turing machine solving problem A should return YES on an input x if an only if there exists some string y_1 such that for all strings y_2 (both of polynomial length), the predicate $V(x, y_1, y_2)$ holds. An example of such a problem is this: given a Boolean formula ϕ over variables $x_1, \ldots, x_n$, is there a formula ϕ' which is equivalent to ϕ and is of size at most k? In this case, y_1 is the formula ϕ', y_2 is an arbitrary assignment to the variables $x_1, \ldots, x_n$, and $V(x, y_1, y_2)$ is the predicate which is true if and only if $x[y_2]$ and $y_1[y_2]$ are both true or both false, meaning that under the variable assignment y_2, ϕ and ϕ' agree. Notice that ϕ' is equivalent to ϕ if and only if it agrees with ϕ under all assignments of Boolean values to the variables.

As we will see, the problem A is a member of the class Σ_2 in the second level of the polynomial hierarchy.

5.2.2 The hierarchy

The polynomial hierarchy starts with familiar classes on level one: $\Sigma_1 = \mathbf{NP}$ and $\Pi_1 = \mathbf{coNP}$. For all $i \geq 1$, it includes two classes, Σ_i and Π_i, which are defined as follows:

$$A \in \Sigma_i \Leftrightarrow \exists y_1.\ \forall y_2.\ \ldots\ .Qy_i.\ V_A(x, y_1, \ldots, y_i)$$

and

$$B \in \Pi_i \Leftrightarrow \forall y_1.\ \exists y_2.\ \ldots\ .Q'y_i.\ V_B(x, y_1, \ldots, y_i)$$

where the predicates V_A and V_B depend on the problems A and B, and Q and Q' represent the appropriate quantifiers, which depend on whether i is even or odd (for example, if $i = 10$ then the quantifier Q for Σ_{10} is $\forall$, and the quantifier Q' for Π_{10} is $\exists$). For clarity, we have also omitted the $p(\cdot)$ side conditions, but they are still there.

One thing that is easy to see is that $\Pi_k = \text{co}\Sigma_k$. Also, note that, for all $i \leq k - 1$, $\Pi_i \subseteq \Sigma_k$ and $\Sigma_i \subseteq \Sigma_k$. These subset relations hold for Π_k as well. This can be seen by noticing that the predicates V do not need to "pay attention to" all of their arguments, and so can represent classes lower on the hierarchy which have a smaller number of them.

Here are some more facts about the polynomial hierarchy (proof left to the reader):

1. Π_i and Σ_i have complete problems for all i.
2. A Σ_i-complete problem is not in Π_j, $j \leq i - 1$, unless $\Pi_j = \Sigma_i$, and it is not in Σ_j unless $\Sigma_j = \Sigma_i$.
3. Suppose that $\Sigma_i = \Pi_i$ for some i. Then $\Sigma_j = \Pi_j = \Sigma_i = \Pi_i$ for all $j \geq i$.
4. Suppose that $\Sigma_i = \Sigma_{i+1}$ for some i. Then $\Sigma_j = \Pi_j = \Sigma_i$ for all $j \geq i$.
5. Suppose that $\Pi_i = \Pi_{i+1}$ for some i. then $\Sigma_j = \Pi_j = \Pi_i$ for all $j \geq i$.

While it seems like an artificial construction right now, in future lectures we will see that the polynomial hierarchy helps us to understand other complexity classes.

5.2.3 An alternate characterization

The polynomial hierarchy can also be characterized in terms of "oracle machines." The idea here is that, instead of a standard Turing machine, we consider one which is augmented with an oracle of a certain power which can be consulted as many times as desired, and using only one computational step each time. Syntactically, this can be written as follows:

Let A be some decision problem and $\mathcal{M}$ be a class of Turing machines. Then $\mathcal{M}^A$ is defined to be the class of machines obtained from $\mathcal{M}$ by allowing instances of A to be solved in one step. For example, $\mathbf{NP}^{\text{3SAT}} = \Sigma_2$.

In general, let L be some Σ_{i-1}-complete problem. Then it can be verified that $\Sigma_i = \mathbf{NP}^L$ and $\Pi_i = \mathbf{coNP}^L$.

Chapter 6

Circuits

February 7, 2001, Scribe: François Labelle

This lecture is on boolean circuit complexity. We first define circuits and the function they compute. Then we consider families of circuits and the language they define. In Section 6.2, we see how circuits relate to other complexity classes by a series of results, culminating with the Karp-Lipton theorem which states that if **NP** problems can be decided with polynomial-size circuits, then $\mathbf{PH} = \Sigma_2$.

6.1 Circuits

A circuit C has n inputs, m outputs, and is constructed with AND gates, OR gates and NOT gates. Each gate has in-degree 2 except the NOT gate which has in-degree 1. The out-degree can be any number. A circuit must have no cycle. See Figure 6.1.

A circuit C with n inputs and m outputs computes a function $f_C : \{0,1\}^n \to \{0,1\}^m$. See Figure 6.2 for an example.

Define $\mathbf{SIZE}(C) = \#$ of AND and OR gates of C. By convention, we do *not* count the NOT gates.

To be compatible with other complexity classes, we need to extend the model to arbitrary input sizes:

DEFINITION 3 *A language L is solved by a family of circuits $\{C_1, C_2, \ldots, C_n, \ldots\}$ if for every $n \geq 1$ and for every x s.t. $|x| = n$,*

$$x \in L \Leftrightarrow f_{C_n}(x) = 1.$$

DEFINITION 4 *Say $L \in \mathbf{SIZE}(s(n))$ if L is solved by a family $\{C_1, C_2, \ldots, C_n, \ldots\}$ of circuits, where C_i has at most $s(i)$ gates.*

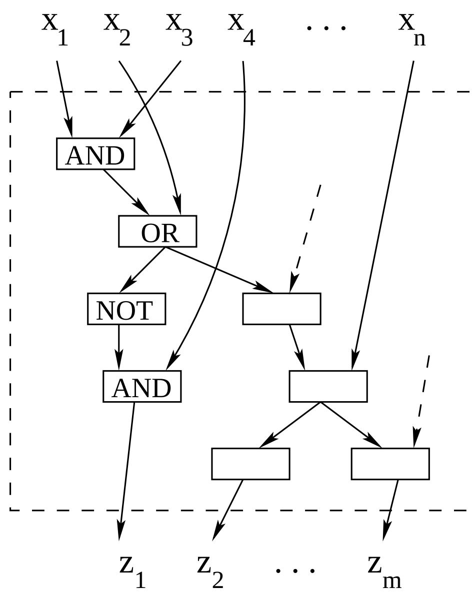

Figure 6.1: A Boolean circuit.

6.2 Relation to other complexity classes

PROPOSITION 14
For every language L, $L \in \mathbf{SIZE}(O(2^n))$. In other words, exponential size circuits contain all languages.

PROOF: We need to show that for every 1-output function $f : \{0,1\}^n \to \{0,1\}$, f has circuit size $O(2^n)$.

Use the identity $f(x_1 x_2 \ldots x_n) = (x_1 \wedge f(1x_2 \ldots x_n)) \vee (\overline{x}_1 \wedge f(0x_2 \ldots x_n))$ to recursively construct a circuit for f, as shown in Figure 6.3.

The recurrence relation for the size of the circuit is: $s(n) = 3 + 2s(n-1)$ with base case $s(1) = 1$, which solves to $s(n) = 2 \cdot 2^n - 3 = O(2^n)$. $\Box$

PROPOSITION 15
If $L \in \mathbf{DTIME}(t(n))$, then $L \in \mathbf{SIZE}(O(t^2(n)))$.

PROOF: Let L be a decision problem solved by a machine M in time $t(n)$. Fix n and x s.t. $|x| = n$, and consider the $t(n) \times t(n)$ tableau of the computation of $M(x)$. See Figure 6.4.

Assume that each entry (a, q) of the tableau is encoded using k bits. By Proposition 14, the transition function $\{0,1\}^{3k} \to \{0,1\}^k$ used by the machine can be implemented by a "next state circuit" of size $k \cdot O(2^k)$, which is exponential in k but constant in n. This building block can be used to create a circuit of size $O(t^2(n))$ that computes the complete tableau, thus also computes the answer to the decision problem. This is shown in Figure 6.5.
$\Box$

COROLLARY 16
$\mathbf{P} \subseteq \mathbf{SIZE}(n^{O(1)})$.

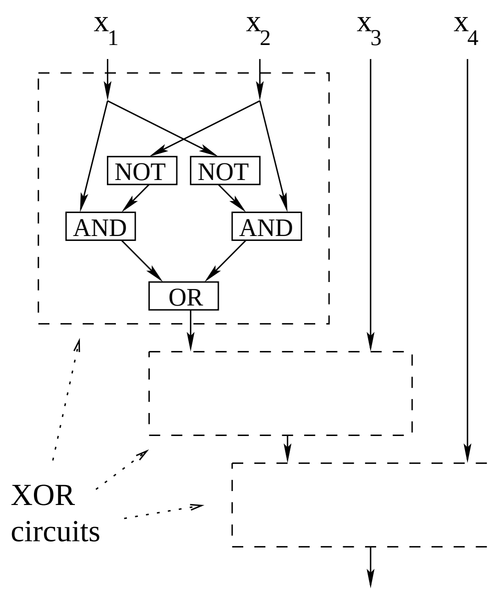

Figure 6.2: A circuit computing the boolean function $f_C(x_1x_2x_3x_4) = x_1 \oplus x_2 \oplus x_3 \oplus x_4$.

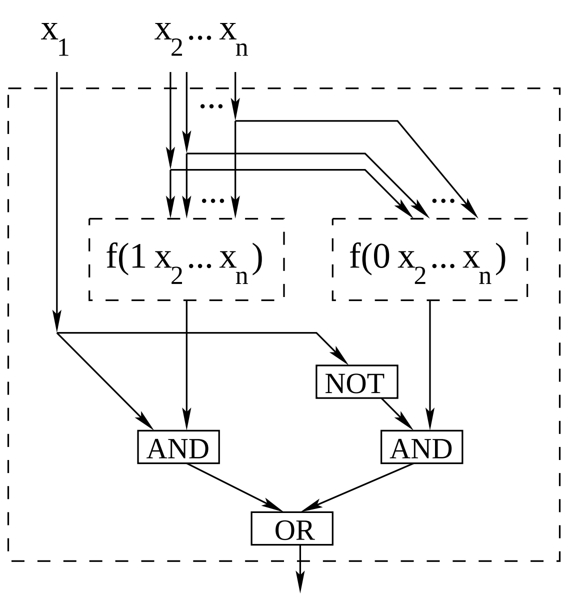

Figure 6.3: A circuit computing any function $f(x_1x_2 \ldots x_n)$ of n variables assuming circuits for two functions of $n-1$ variables.

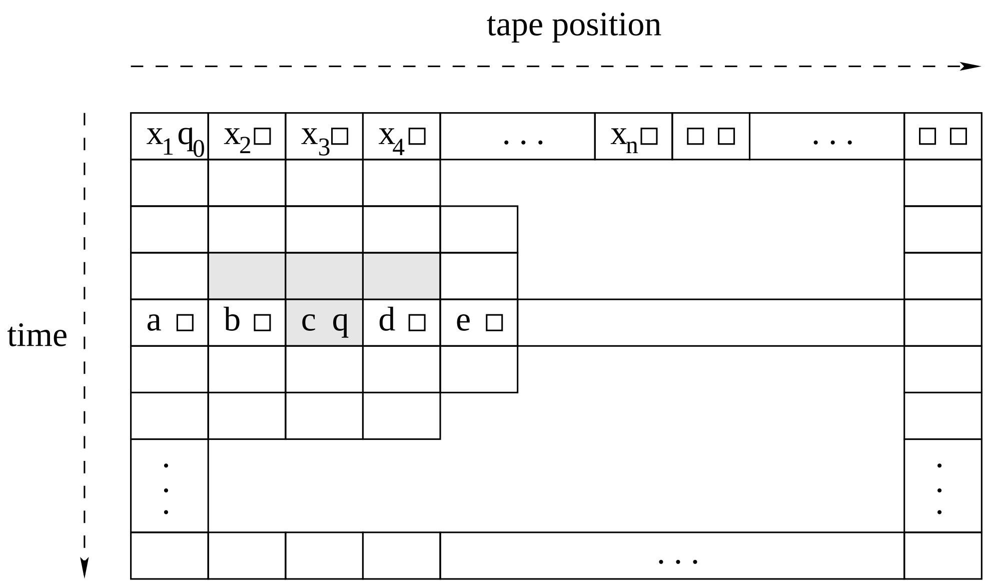

Figure 6.4: $t(n) \times t(n)$ tableau of computation. The left entry of each cell is the tape symbol at that position and time. The right entry is the machine state or a blank symbol, depending on the position of the machine head.

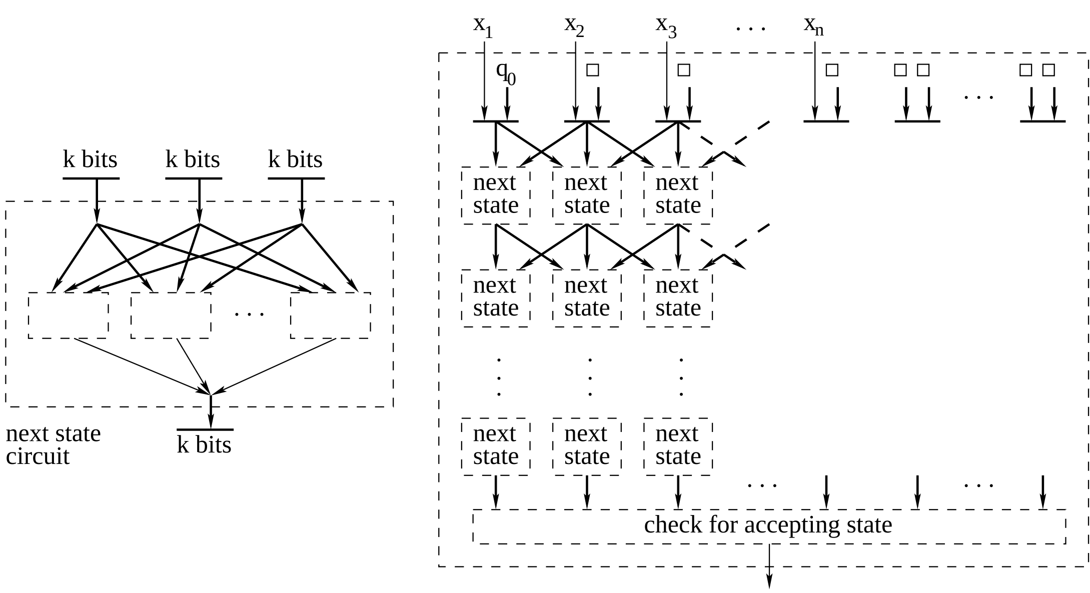

Figure 6.5: Circuit to simulate a Turing machine computation by constructing the tableau.

Exercise 1 $\mathbf{P} \neq \mathbf{SIZE}(n^{O(1)})$.

Proposition 17
There are languages L such that $L \in \mathbf{SIZE}(2^{o(n)})$. In other words, for every n, there exists $f : \{0,1\}^n \to \{0,1\}$ that cannot be computed by a circuit of size $2^{o(n)}$.

Proof: This is a counting argument. There are 2^{2^n} functions $f : \{0,1\}^n \to \{0,1\}$, and we claim that the number of circuits of size s is at most $2^{O(s \log s)}$, assuming $s \geq n$. To bound the number of circuits of size s we create a compact binary encoding of such circuits. Identify gates with numbers $1, \ldots, s$. For each gate, specify where the two inputs are coming from, whether they are complemented, and the type of gate. The total number of bits required to represent the circuit is

$$s(2\log(n+s)+3) \leq s(2\log 2s + 3) = s(2\log 2s + 5).$$

So the number of circuits of size s is at most $2^{2s\log s + 5s}$, and this is not sufficient to compute all possible functions if

$$2^{2s\log s+5s} < 2^{2^n}.$$

This is satisfied if $s = 2^{o(n)}$. $\Box$

Exercise 2 Show that there is a language in $\mathbf{SPACE}(2^{n^{O(1)}})$ that does not belong to $\mathbf{SIZE}(2^{o(n)})$.

Theorem 18 (Karp-Lipton)
If $\mathbf{NP} \subseteq \mathbf{SIZE}(n^{O(1)})$ then $\mathbf{PH} = \Sigma_2$. In other words, the polynomial hierarchy would collapse to its second level.

Proof: We will show that if $\mathbf{NP} \subseteq \mathbf{SIZE}(n^{O(1)})$ then $\Pi_2 \subseteq \Sigma_2$. By a result in a previous lecture, this implies that $\mathbf{PH} = \Sigma_2$. Recall the definitions:

L is in Σ_2 if there is a polynomial time verifier V and polynomials p_1, p_2 s.t.

$$x \in L \;\Leftrightarrow\; \exists y_1\, |y_1| \leq p_1(|x|) \cdot \forall y_2\, |y_2| \leq p_2(|x|) \cdot V(x, y_1, y_2) = 1.$$

Similarly, L is in Π_2 [...]

$$x \in L \;\Leftrightarrow\; \forall y_1\, |y_1| \leq p_1(|x|) \cdot \exists y_2\, |y_2| \leq p_2(|x|) \cdot V(x, y_1, y_2) = 1.$$

Assume $\mathbf{NP} \subseteq \mathbf{SIZE}(n^{O(1)})$ and let L be in Π_2. For every n, there is a circuit C_n such that $C_n(x, y_1) = 1$ iff $\exists y_2$ such that $V(x, y_1, y_2) = 1$, where $|x| = n$, $|y_1| = p_1(n)$, $|y_2| = p_2(n)$. So,

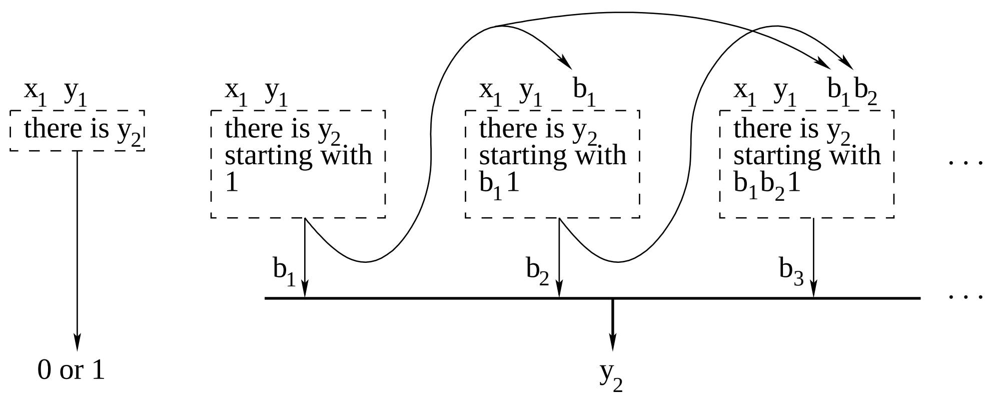

Figure 6.6: How to use decision problem solvers to find a witness to a search problem.

$$
\begin{aligned}
x \in L \quad \Leftrightarrow \quad & \exists C_n \cdot (\forall y_1 \cdot C_n(x, y_1) = 1) \\
& \text{and } C_n \text{ is the circuit that decides whether } \exists y_2 \cdot V(x, y_1, y_2) = 1 \\
\Leftrightarrow \quad & \exists C_n \cdot (\forall y_1 \cdot C_n(x, y_1) = 1) \\
& \wedge \forall y_1 \{(\exists y_2 \cdot V(x, y_1, y_2) = 1) \Leftrightarrow C_n(x, y_1) = 1\} \quad \text{(checking specifications)}
\end{aligned}
$$

... which is a Σ_3 formula, we are not in good shape. We need the circuit for the **NP** problem to not only answer the decision problem, but also to report a witness if the answer is YES.

(Second try.) There is a circuit C_n such that

$$
C_n(x, y_1) = \begin{cases} 0 & \text{if there is no } y_2 \text{ such that } V(x, y_1, y_2) = 1 \\ 1y_2 & \text{otherwise, where } y_2 \text{ is such that } V(x, y_1, y_2) = 1. \end{cases}
$$

See Figure 6.6 if you can't immediately see how a decision problem solver can always be converted into a witness finder.

Let C be a circuit. The formula for checking specifications is

$$
\begin{aligned}
\forall x \forall y_1 \forall y_2 \cdot \quad & (V(x, y_1, y_2) = 1) \Rightarrow (C(x, y_1) \neq 0) \\
& \wedge (C(x, y_1) \neq 0) \Rightarrow (V(x, y_1, C(x, y_1)) = 1)
\end{aligned}
$$

which can be used to build a Σ_2 formula for L as follows:

$$
\begin{aligned}
x \in L \;\Leftrightarrow\; \exists C_n \cdot \forall z \forall y_1 \forall y_2 \cdot \quad & (C_n(x, y_1) \neq 0) \\
& \wedge (V(z, y_1, y_2) = 1) \Rightarrow (C_n(z, y_1) \neq 0) \\
& \wedge (C_n(z, y_1) \neq 0) \Rightarrow (V(z, y_1, C_n(z, y_1)) = 1).
\end{aligned}
$$

□

Chapter 7

Probabilistic Complexity Classes

February 12, 2001, Scribe: Iordanis Kerenidis

In this lecture we will define the probabilistic complexity classes **BPP**, **RP**, **ZPP** and we will see how they are related to each other, as well as to other deterministic or circuit complexity classes. Further, we will describe a probabilistic algorithm for the primality testing.

7.1 Probabilistic complexity classes

First we are going to describe the probabilistic model of computation. In this model an algorithm A gets as input a sequence of random bits r and the "real" input x of the problem. The output of the algorithm is the correct answer for the input x with some probability.

DEFINITION 5 *An algorithm A is called a polynomial time probabilistic algorithm if the size of the random sequence $|r|$ is polynomial in the input $|x|$ and $A()$ runs in time polynomial in $|x|$.*

If we want to talk about the correctness of the algorithm, then informally we could say that for every input x we need $\mathbf{Pr}[A(x,r) = \text{correct answer for} x] \geq \frac{2}{3}$. That is, for every input the probability distribution over all the random sequences must be some constant bounded away from $\frac{1}{2}$. Let us now define the class **BPP**.

DEFINITION 6 *A decision problem L is in* **BPP** *if there is a polynomial time algorithm A and a polynomial $p()$ such that :*

$$\forall x \in L \quad \mathbf{Pr}_{r \in \{0,1\}^{p(|x|)}}[A(x,r) = 1] \geq 2/3$$

$$\forall x \notin L \quad \mathbf{Pr}_{r \in \{0,1\}^{p(|x|)}}[A(x,r) = 1] \leq 1/3$$

We can see that in this setting we have an algorithm with two inputs and some constraints on the probabilities of the outcome. In the same way we can also define the class **P** as:

Definition 7 *A decision problem L is in* **P** *if there is a polynomial time algorithm A and a polynomial $p()$ such that :*

$$\forall x \in L \quad : \mathbf{Pr}_{r\in\{0,1\}^{p(|x|)}}[A(x,r)=1]=1$$

$$\forall x \notin L : \mathbf{Pr}_{r\in\{0,1\}^{p(|x|)}}[A(x,r)=1]=0$$

Similarly, we define the classes **RP** and **ZPP**.

Definition 8 *A decision problem L is in* **RP** *if there is a polynomial time algorithm A and a polynomial $p()$ such that :*

$$\forall x \in L \quad \mathbf{Pr}_{r\in\{0,1\}^{p(|x|)}}[A(x,r)=1] \geq 1/2$$
$$\forall x \notin L \quad \mathbf{Pr}_{r\in\{0,1\}^{p(|x|)}}[A(x,r)=1] \leq 0$$

Definition 9 *A decision problem L is in* **ZPP** *if there is a polynomial time algorithm A whose output can be $0, 1, ?$ and a polynomial $p()$ such that :*

$$\forall x \quad \mathbf{Pr}_{r\in\{0,1\}^{p(|x|)}}[A(x,r)=?] \leq 1/2$$

$$\forall x, \forall r \quad \textit{such that} \quad A(x,r) \neq ? \quad \textit{then} \quad A(x,r)=1 \quad \textit{if and only if} \quad x \in L$$

7.2 Relations between complexity classes (or where do probabilistic classes stand?)

After defining these probabilistic complexity classes, let's see how they are related to other complexity classes and with each other.

Theorem 19
RP $\subseteq$**NP**

Proof: Suppose we have an algorithm for **RP**. Then this algorithm is also in **NP**. If $x \in L$ then there is a random sequence r, for which the algorithm answers yes. On the other hand, if $x \notin L$ then there is no witness. □

Most natural probabilistic algorithms belong to the class **RP**. We can also show that the class **ZPP** is more restricted than **RP**.

Theorem 20
ZPP $\subseteq$**RP**

Proof: We are going to convert a **ZPP** algorithm into an **RP** algorithm. The construction consists of running the **ZPP** algorithm and anytime it outputs ?, the new algorithm will answer 0. In this way, if the right answer is 0, then the algorithm will answer 0 with probability 1. On the other hand, when the right answer is 1, then the algorithm will give the wrong answer with probability less than 1/2, since the probability of the **ZPP** algorithm giving the output ? is less than 1/2. □

Another interesting property of the class **ZPP** is that it's equivalent to the class, which contains all languages for which there is an average polynomial time algorithm that always gives the right answer. More formally,

THEOREM 21
A language L is in the class **ZPP** *if and only if L has an average polynomial time algorithm that always gives the right answer.*

PROOF: First let us clarify what we mean by average time. For each input x we take the average time of $A(x, r)$ over all random sequences r. Then for size n we take the worst time over all possible inputs x of size $|x| = n$. In order to construct an algorithm that always gives the right answer we run the **ZPP** algorithm and if it outputs a ?, then we run it again. Suppose that the running time of the **ZPP** algorithm is T, then the average running time of the new algorithm is:

$$T_{avg} = \frac{1}{2} \cdot T + \frac{1}{4} \cdot 2T + \ldots + \frac{1}{2^k} \cdot kT \approx O(T)$$

Now, we want to prove that if the language L has an algorithm that runs in polynomial average time $t(|x|)$, then this is in **ZPP**. What we do is run the algorithm for time $2t(|x|)$ and output a ? if the algorithm hs not yet stopped. It is straightforward to see that this belongs to **ZPP**. First of all, the worst running time is polynomial, actually $2t(|x|)$. Moreover, the probability that our algorithm outputs a ? is less than 1/2, since the original algorithm has an average running time $t(|x|)$ and so it must stop before time $2t(|x|)$ at least half of the times. □

EXERCISE 3 **ZPP =RP ∩coRP**

Let us now prove the fact that **RP** is contained in **BPP**.

THEOREM 22
RP ⊆BPP

PROOF: We will convert an **RP** algorithm into a **BPP** algorithm. In the case that the input x does not belong to the language then the **RP** algorithm always gives the right answer, so this is definitely in **BPP** as well. In the case that the input x does belong to the language then we need to boost the probability of a correct answer from at least 1/2 to at least 2/3.

More formally, let A be an **RP** algorithm for a decision problem L. We fix some number k and define the following algorithm:

$A^{(k)}$

input: x,
  pick $r_1, r_2, \ldots, r_k$
  if $A(x, r_1) = A(x, r_2) = \ldots = A(x, r_k) = 0$ **then** **return** 0
  else return 1

Let us now consider the correctness of the algorithm.In case the correct answer is 0 the output is always right, though in the case where the right answer is 1 the output is right except when all $A(x, r_i) = 0$.

$$\begin{array}{lll} \text{if} & x \notin L & \mathbf{Pr}_{r_1,\dots,r_k}[A^k(x, r_1, \dots, r_k) = 1] = 0 \\ \text{if} & x \in L & \mathbf{Pr}_{r_1,\dots,r_k}[A^k(x, r_1, \dots, r_k) = 1] \geq 1 - \left(\frac{1}{2}\right)^k \end{array}$$

It is easy to see that by choosing an appropriate k the second probability can go arbitrarily close to 1 and therefore become larger than 2/3, which is what is required by the definition of **BPP**. In fact, by choosing k to be a polynomial in $|x|$, we can make the probability exponentially close to 1. This enables us to change the definition of **RP** and instead of the bound of 1/2 for the probability of a correct answer when the input is in the language L, we can have a bound of $1 - \left(\frac{1}{2}\right)^{q(|x|)}$, for a fixed polynomial q. □

Let, now, A be a **BPP** algorithm for a decision problem L. Then, we fix k and define the following algorithm:

$A^{(k)}$

input: x,
  pick $r_1, r_2, \dots, r_k$
  $c = \sum_{i=1}^{k} A(x, r_i)$
  if $c \geq \frac{k}{2}$ **then return** 1
  else return 0

In a **BPP** algorithm we expect the right answer to come up with probability more than 1/2. So, by running the algorithm many times we make sure that this slightly bigger than 1/2 probability will actually show up in the results. More formally let us define the Chernoff bounds.

Theorem 23
(Chernoff Bound)
Suppose $X_1, \dots, X_k$ *are independent random variables with values in* $\{0,1\}$ *and for every* i, $\mathbf{Pr}[X_i = 1] = p$. *Then:*

$$\mathbf{Pr}[\tfrac{1}{k}\sum_{i=1}^{k} X_i - p > \epsilon] < e^{-\epsilon^2 \frac{k}{2p(1-p)}}$$

$$\mathbf{Pr}[\tfrac{1}{k}\sum_{i=1}^{k} X_i - p < -\epsilon] < e^{-\epsilon^2 \frac{k}{2p(1-p)}}$$

The Chernoff bounds will enable us to bound the probability that our result is far from the expected. Indeed, these bounds say that this probability is exponentially small in respect to k.

Let us now consider how the Chernoff bounds apply to the algorithm we described previously. We fix the input x and call $p = \mathbf{Pr}_r[A(x,r) = 1]$ over all possible random sequences. We also define the independent random variables $X_1, \ldots, X_k$ such that $X_i = A(x, r_i)$.

First, suppose $x \in L$. Then the algorithm $A^{(k)}(x, r_1, \ldots, r_k)$ outputs the right answer 1, when $\frac{1}{k}\sum_i X_i \geq \frac{1}{2}$. So, the algorithm makes a mistake when $\frac{1}{k}\sum_i X_i < \frac{1}{2}$.

We now apply the Chernoff bounds to bound this probability.

$$\begin{aligned}\mathbf{Pr}[A^{(k)}\text{outputs the wrong answer on } \ x] &= \mathbf{Pr}[\tfrac{1}{k}\sum_i X_i < \tfrac{1}{2}] \\ &\leq \mathbf{Pr}[\tfrac{1}{k}\sum_i X_i - p \leq -\tfrac{1}{6}]\end{aligned}$$

since $p \geq \frac{2}{3}$.

$$\leq e^{-\frac{k}{72p(1-p)}} = 2^{-\Omega(k)}$$

The probability is exponentially small in k. The same reasoning applies also for the case where $x \notin L$. Further, it is easy to see that by choosing k to be a polynomial in $|x|$ instead of a constant, we can change the definition of a **BPP** algorithm and instead of the bound of $\frac{1}{3}$ for the probability of a wrong answer, we can have a bound of $2^{-q(|x|)}$, for a fixed polynomial q.

Next, we are going to see how the probabilistic complexity classes relate to circuit complexity classes and specifically prove that the class **BPP** has polynomial size circuits.

Theorem 24
BPP $\subseteq$**SIZE**$(n^{O(1)})$

Proof: Let L be in the class **BPP**. Then by definition, there is a polynomial time algorithm A and a polynomial p, such that for every input x

$$\mathbf{Pr}_{r \in \{0,1\}^{p(|x|)}}[A(x,r) = \text{wrong answer for } \ x] \leq 2^{-(n+1)}$$

This follows from our previous conclusion that we can replace $\frac{1}{3}$ with $2^{-q(|x|)}$. We now fix n and try to construct a family of circuits C_n, that solves L on inputs of length n.

Lemma 25
There is a random sequence $r \in \{0,1\}^{p(n)}$ such that for very $x \in \{0,1\}^n$ $A(x,r)$ is correct.

Proof: Informally, we can see that for each input x the number of random sequences r that give the wrong answer is exponentially small. Therefore, even if we assume that these sequences are different for every input x, their sum is still less than the total number of random sequences. Formally, let's consider the probability over all sequences that the algorithm gives the right answer for all input. If this probability is greater than 0, then the claim is proved.

$$\mathbf{Pr}_r[\text{for every } \ x, A(x,r) \ \text{ is correct}] = 1 - \mathbf{Pr}_r[\exists x, A(x,r) \ \text{ is wrong}]$$

the second probability is the union of 2^n possible events for each x. This is bounded by the sum of the probabilities.

$$
\begin{aligned}
&\leq 1 - \sum_{x \in \{0,1\}^n} \mathbf{Pr}_r[A(x,r)\text{is wrong}] \\
&\leq 1 - 2^n \cdot 2^{-(n+1)} \\
&\leq \frac{1}{2}
\end{aligned}
$$

□

So, we proved that at least half of the random sequences are correct for all possible input x. Therefore, it is straightforward to see that we can simulate the algorithm $A(\cdot,\cdot)$, where the first input has length n and the second $p(n)$, by a circuit of size polynomial in n.

All we have to do is find a random sequence which is always correct and build it inside the circuit. Hence, our circuit will take as input only the input x and simulate A with input x and r for this fixed r. Of course, this is only an existential proof, since we don't know how to find this sequence efficiently. □

In conclusion, let us briefly describe some other relations between complexity classes. Whether **BPP** $\subseteq$ **NP** or not is still an open question. What we know is that it's unlikely that **NP** is contained in **BPP**, since then by the previous result **NP** would have polynomial size circuits and hence by the result of Karp and Lipton the polynomial hierarchy would collapse.

7.3 A BPP algorithm for primality testing

We are going to describe a **BPP** algortihm for the problem of deciding whether a number is prime or not. Note that there are also algorithms for primality testing which belong to **RP** or **ZPP**.

We'll begin by some known facts of number theory.

Definition 10 *A number $a \in \{0,\dots,n-1\}$ is a quadratic residue $(\mod n)$ if there is a number r, such that $a \equiv r^2 (\mod n)$.*

Theorem 26
If n is a prime and a is a quadratic residue $(\mod n)$ then a has exactly two square roots $(\mod n)$ in $\{0,\dots,n-1\}$.

Proof: We know that a has at least two roots, namely $r, -r$. We also know that there can't be more than two roots since $r^2 - a$ is a polynomial of degree 2. Hence, there are exactly two square roots. □

If now n is not a prime we can prove the following

Theorem 27
If $n \geq 3$, not a prime power, and a is a quadratic residue, then a has at least four roots in $\{0,\dots,n-1\}$.

PROOF: Suppose $n = pq$, where p, q have no common factors. Then if r is a root of a, then we can construct the following four systems of equations.

$$\begin{aligned} x = r(\mod p) \quad &, \quad x = r(\mod q) \\ x = r(\mod p) \quad &, \quad x = -r(\mod q) \\ x = -r(\mod p) \quad &, \quad x = r(\mod q) \\ x = -r(\mod p) \quad &, \quad x = -r(\mod q) \end{aligned}$$

Using the Chinese Remainder Theorem, it is easy to see that each one of these systems has at least one solution which is different in each case. Each one of these solutions is also a root of a, hence a has at least four square roots. $\square$

We also know that there is a probabilistic polynomial time algorithm R that on input a prime n and a quadratic residue a, it finds two roots of a. If the input is not correct then the algorithm outputs arbitrarily. It is easy to see that this algorithm can only work for a prime, since otherwise it would also be an algorithm for factorization.

We are now ready to describe our algorithm for primality testing:

input: n (n odd and square free),
pick $r \in \{1, \ldots, n-1\}$
  compute $R(n, r^2 \mod n)$
  if output of R not $\pm r$ **then** **return** "not prime"
  else return "prime"

In this algorithm, we select a random number, square it and ask for the roots. When n is a prime, then the algorithm R will find the two roots, which are $\pm r$ with high probability, hence our algorithm will answer "prime". On the other hand, when n is not prime, then since there are at least four roots, the algorithm R can return $\pm r$ with at most half probability. More precisely, the algorithm R is probabilistic and not exact, but there will still be a gap in the probabilities of the right and wrong answer. By amplifying this gap, we have that the algorithm is in **BPP**.

Chapter 8

Probabilistic Algorithms

February 14, 2001, Scribe: Steve Chien

Today we discuss checking equivalence of branching programs.

8.1 Branching Programs

We begin by introducing another model of computation, that of branching programs. A branching program accepts or rejects input strings $x_1, \ldots, x_n$ and can be described as a directed acyclic graph with a single root and two final states, 0 and 1. The computational path begins at the root. Each non-final vertex is labeled with some single variable x_i and has two outgoing edges, one labeled 0 and the other 1. The path follows edge b such that $x_i = b$, and the program accepts the input if and only if the path ends at the final state 1.

An illustrative example is the branching program that considers input string $x_1x_2x_3$ and outputs 1 if $x_1 \oplus x_2 \oplus x_3 = 1$. This is shown in figure 8.1.

Branching programs are a reasonably powerful model of computation; for example we can show that **L** can be computed by branching programs of polynomial size.

8.2 Testing Equivalence

We would like to be able to consider two branching programs and determine if they compute the same function. In general, testing equivalence for any model of computation is difficult: for Turing machines, the problem is undecidable, and for circuits, the problem is **coNP**-complete. A similar result is true for branching programs– here, testing equivalence is also **coNP**-complete, through a reduction from co-3SAT.

Happily, the problem becomes easier once we restrict it to *read-once* branching programs, in which every path from the root to a final state tests each variable at most once. We will show that in this case testing equivalence is in **coRP**. Our strategy will be as follows: Given two read-once branching programs B_1 and B_2, we will define two polynomials p_1 and p_2 of degree n over $x_1, \ldots, x_n$ that are equivalent if and only if B_1 and B_2 are also equivalent. We will then use a **coRP** algorithm to test the equivalence of the two polynomials. The

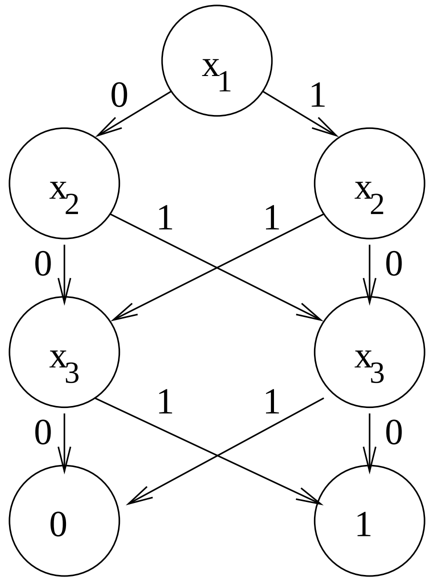

Figure 8.1: A branching program.

polynomials will be represented in compact form, but will have an exponential number of monomials and thus we cannot simply directly compare their coefficients.

Our polynomials will be determined as follows:

- We assign each vertex v_i in the branching program from the root to the final states a polynomial $p_i(x_1, \ldots, x_n)$, beginning with the root vertex which gets the constant polynomial 1.

- Once a vertex has been assigned a polynomial $p(x_1, \ldots, x_n)$, its outgoing edges are then given polynomials in the following manner: If the vertex performed a test on x_j, the outgoing 0 edge receives the polynomial $(1 - x_j)p(x_1, \ldots, x_n)$ and the outgoing 1 edge receives $(x_j)p(x_1, \ldots, x_n)$.

- Once all of a vertex's incoming edges have been appropriately labeled, that vertex's polynomial becomes the sum of the polynomials of the incoming edges.

- The polynomial associated with the final state 1 is the polynomial of the branching program.

These rules are illustrated in figure 8.2. The idea is that the polynomial at each vertex is nonzero on a set of inputs only if that set of inputs will lead the branching program to that vertex.

We can see that when the branching programs are read-once, the degree of a variable in any monomial in the resulting polynomial will have power no larger than 1, and hence the polynomial will be of degree at most n.

We now observe that given read-once branching programs B_1 and B_2, their corresponding polynomials p_1 and p_2 are equivalent if and only if B_1 and B_2 are equivalent. To see

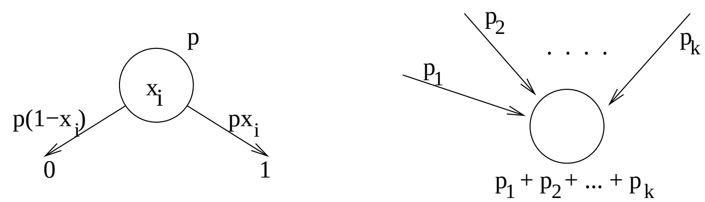

Figure 8.2: Constructing the polynomial of a branching program.

this, if any monomial m in p_1 or p_2 does not contain a variable x_i we replace m with $mx_i + m(1 - x_i)$. We can then write p_1 and p_2 as sums of terms of the form $\prod_{i=1}^{n} y_i$, where y_i is either x_i or $(1 - x_i)$. Each term corresponds to a path in the branching program that ends in final state 1. From this, some extra thought reveals that equivalence of the polynomials is equivalent to equivalence of the branching programs.

Notice also that if the branching programs are polynomial size, we can evaluate the polynomials efficiently, even though they may contain an exponential number of coefficients. All we need now is a method of testing equivalence of polynomials efficiently.

8.3 The Schwartz-Zippel Lemma

The tool we will use for this is the Schwartz-Zippel lemma, which states:

Lemma 28
If $p(x_1, \ldots, x_n)$ is an n-variate nonzero polynomial of degree d over a finite field $\mathbb{F}$, then p has at most $d\mathbb{F}^{n-1}$ roots. Equivalently, $\mathbf{Pr}[p(a_1, \ldots, a_n) = 0] \leq d/\mathbb{F}$.

Notice that we use $\mathbb{F}$ to be both the field and its size.

Proof: The proof proceeds by induction on n. The base case of $n = 1$ is simply that of a univariate polynomial and follows immediately.

Now consider $p(x_1, \ldots, x_n)$ of degree at most d. We can write

$$p(x_1, \ldots, x_n) = p_0(x_2, \ldots, x_n) + x_1 p_1(x_2, \ldots, x_n) + x_1^2 p_2(x_2, \ldots, x_n) + \ldots + x_1^k p_k(x_2, \ldots, x_n)$$

where k is the largest value for which p_k is not identically zero.

We now observe that

$$|\{a_1, \ldots, a_n\} : p(a_1, \ldots, a_n) \neq 0| \geq |\{a_1, \ldots, a_n\} : p(a_1, \ldots, a_n) \neq 0 \text{ and } p_k(a_2, \ldots, a_n) \neq 0|$$

By induction, p_k is nonzero on at least $\mathbb{F}^{n-1} - (d-k)\mathbb{F}^{n-2}$ points. Further, for each such point $a_2, \ldots, a_n$, $p(x_1, a_2, \ldots, a_n)$ is a nonzero univariate polynomial and hence there are at least $\mathbb{F} - k$ values of a_1 such that $p(a_1, \ldots, a_n) \neq 0$.

Putting this together, we can bound the number of nonzero points from below by

$$\mathbb{F}^{n-1}(1 - \frac{d-k}{\mathbb{F}})\mathbb{F}(1 - \frac{k}{\mathbb{F}} \geq \mathbb{F}^n(1 - \frac{d}{\mathbb{F}}) = \mathbb{F}^n - d\mathbb{F}^{n-1}$$

This finishes the proof. $\square$

An algorithm for testing equivalence of polynomials and hence read-once branching programs is now immediate. We choose a field of size at least $3d$, and then choose random $a_1, \ldots, a_n$ from $\mathbb{F}^n$. We accept if $p_1(a_1, \ldots, a_n) = p_2(a_1, \ldots, a_n)$.

If the two polynomials are equivalent, then we always accept; otherwise reject with probability at least $2/3$, making this a **coRP** algorithm.

Chapter 9

BPP versus PH, and Counting Problems

February 21, 2001, Scribe: Lawrence Ip

9.1 BPP $\subseteq \Sigma_2$

This result was first shown by Sipser and Gacs. Lautemann gave a much simpler proof which we give below.

Lemma 29
If L is in **BPP** *then there is an algorithm A such that for every x,*

$$\mathbf{Pr}_r(A(x,r) = \textit{right answer}) \geq 1 - \tfrac{1}{3m},$$

where the number of random bits $|r| = m = |x|^{O(1)}$ and A runs in time $|x|^{O(1)}$.

Proof: Let $\hat{A}$ be a **BPP** algorithm for L. Then for every x, $\mathbf{Pr}_r(\hat{A}(x,r) = \text{wrong answer}) \leq \frac{1}{3}$. and $\hat{A}$ uses $\hat{m}(n)$ random bits where $n = |x|$.

Do $k(n)$ repetitions of $\hat{A}$ and accept if and only if at least $\dfrac{k(n)}{2}$ executions of $\hat{A}$ accept. Call the new algorithm A. Then A uses $k(n)\hat{m}(n)$ random bits and $\mathbf{Pr}_r(A(x,r) = \text{wrong answer}) \leq 2^{-ck(n)}$. We can then find $k(n)$ with $k(n) = \Theta(\log \hat{m}(n))$ such that $\frac{1}{2^{ck(n)}} \leq \frac{1}{3k(n)\hat{m(n)}}$. $\square$

Theorem 30
BPP $\subseteq \Sigma_2$.

Proof: Let L be in **BPP** and A as in the lemma. Then we want to show that

$$x \in L \Leftrightarrow \exists y_1, \ldots, y_m \in \{0,1\}^m \forall z \in \{0,1\}^m \bigvee_{i=1}^{m} A(x, y_i \oplus z) = 1$$

where m is the number of random bits used by A on input x.
Suppose $x \in L$. Then

$$\begin{aligned}
&\mathbf{Pr}_{y_1,\dots,y_m}(\exists z A(x, y_1 \oplus z) = \cdots = A(x, y_m \oplus z) = 0) \\
&\leq \sum_{z \in \{0,1\}^m} \mathbf{Pr}_{y_1,\dots,y_m}(A(x, y_1 \oplus z) = \cdots = A(x, y_m \oplus z) = 0) \\
&\leq 2^m \frac{1}{(3m)^m} \\
&< 1.
\end{aligned}$$

So

$$\begin{aligned}
\mathbf{Pr}_{y_1,\dots,y_m}(\forall z \bigvee_i A(x, y_i \oplus z)) &= 1 - \mathbf{Pr}_{y_1,\dots,y_m}(\exists z A(x, y_1 \oplus z) = \cdots = A(x, y_m \oplus z) = 0) \\
&> 0.
\end{aligned}$$

So $(y_1, \dots, y_m)$ exists.
Conversely suppose $x \notin L$. Then

$$\begin{aligned}
\mathbf{Pr}_z \left(\bigvee_i A(x, y_i \oplus z) \right) &\leq \sum_i \mathbf{Pr}_z \left(A(x, y_i \oplus z) = 1 \right) \\
&\leq m \cdot \frac{1}{3m} \\
&= \frac{1}{3}.
\end{aligned}$$

So

$$\begin{aligned}
\mathbf{Pr}_z(A(x, y_1 \oplus z) = \cdots = A(x, y_m \oplus z) = 0) &= \mathbf{Pr}_z \left(\bigvee_i A(x, y_i \oplus z) \right) \\
&\geq \frac{2}{3} \\
&> 0.
\end{aligned}$$

So there is a z such that $\bigvee_i A(x, y_i \oplus z) = 0$ for all $y_1, \dots, y_m \in \{0,1\}^m$. $\square$

9.2 Counting Classes

Definition 11 *R is an* **NP**-relation, *if there is a polynomial time algorithm A such that $(x, y) \in R \Leftrightarrow A(x, y) = 1$ and there is a polynomial p such that $(x, y) \in R \Rightarrow |y| \leq p(|x|)$.*

$\#R$ is the problem that, given x, asks how many y satisfy $(x, y) \in R$.

Definition 12 *$\#\mathbf{P}$ is the class of all problems of the form $\#R$, where R is an* **NP***-relation.*

Unlike for decision problems there is no canonical way to define reductions for counting classes. There are two common definitions.

Definition 13 *We say there is a* parsimonious reduction *from $\#A$ to $\#B$ (written $\#A \leq_{par} \#B$) if there is a polynomial time transformation f such that for all x, $|\{y, (x, y) \in A\}| = |\{z : (f(x), z) \in B\}|$.*

Often this definition is a little too restrictive and we use the following definition instead.

Definition 14 *$\#A \leq \#B$ if there is a polynomial time algorithm for $\#A$ given an oracle that solves $\#B$.*

#CIRCUITSAT is the problem where given a circuit, we want to count the number of inputs that make the circuit output 1.

Theorem 31
*#CIRCUITSAT is #**P**-complete under parsimonious reductions.*

Proof: Let $\#R$ be in $\#\mathbf{P}$ and A and p be as in the definition. Given x we want to construct a circuit C such that $|\{z : C(z)\}| = |\{y : |y| \leq p(|x|), A(x, y) = 1\}|$. We then construct $\hat{C}_n$ that on input x, y simulates $A(x, y)$. From earlier arguments we know that this can be done with a circuit with size about the square of the running time of A. Thus $\hat{C}_n$ will have size polynomial in the running time of A and so polynomial in x. Then let $C(y) = \hat{C}(x, y)$. □

Theorem 32
*#3SAT is #**P**-complete.*

Proof: We show that there is a parsimonious reduction from #CIRCUITSAT to #3-SAT. That is, given a circuit C we construct a Boolean formula ϕ such that the number of satisfying assignments for ϕ is equal to the number of inputs for which C outputs 1. Suppose C has inputs $x_1, \ldots, x_n$ and gates $1, \ldots, m$ and ϕ has inputs $x_1, \ldots, x_n, g_1, \ldots, g_m$, where the g_i represent the output of gate i. Now each gate has two input variables and one output variable. Thus a gate can be complete described by mimicking the output for each of the 4 possible inputs. Thus each gate can be simulated using at most 4 clauses. In this way we have reduced C to a formula ϕ with $n + m$ variables and $4m$ clauses. So there is a parsimonious reduction from #CIRCUITSAT to #3SAT. □

Chapter 10

Approximate Counting

March 6,2001, Scribe: Jittat Fakcharoenphol

In this lecture, we will prove the theorem concerning the connections between counting problems and the power of randomization.

10.1 Complexity of counting problems

We will prove the following theorem:

THEOREM 33
*For every counting problem #A in #**P**, there is an algorithm C that on input x, compute with high probability a value v such that*

$$(1-\epsilon)\#A(x) \leq v \leq (1+\epsilon)\#A(x)$$

in time polynomial in $|x|$ and in $\frac{1}{\epsilon}$, using an oracle for **NP**.

The theorem says that #**P** can be approximate in $\mathbf{BPP}^{\mathbf{NP}}$. We have a remark here that approximating #3SAT is **NP**-hard. Therefore, to compute the value we need at least the power of **NP**, and this theorem states that the power of **NP** and randomization is sufficient.

Another remark is concerning on Toda's theorem which states that $\mathbf{PH} \subseteq \mathbf{P}^{\#\mathbf{P}}$. This implies that #3SAT is Σ_k-hard for every k, i.e., #3SAT lies outside **PH**. Recall that **BPP** lies inside Σ_3, and hence approximating #3SAT can be done in Σ_3. Therefore, approximating #3SAT cannot be equivalent to computing #3SAT exactly, unless the polynomial hierarchy collapses.

We first make some observations so that we can reduce the proof to an easier one.

- It is enough to prove the theorem for #3SAT.

  If we have an approximation algorithm for #3SAT, we can extend it to any #A in #**P** using the parsimonious reduction from #A to #3SAT.

- It is enough to give a polynomial time $O(1)$-approximation for #3SAT.

  Suppose we have an algorithm C and a constant c such that

  $$\frac{1}{c}\#3\mathrm{SAT}(\varphi) \leq C(\varphi) \leq c\#3\mathrm{SAT}(\varphi).$$

  Given φ, we can construct $\varphi^k = \varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_k$. If φ has t satisfying assignments, φ^k has t^k satisfying assignments. Then, giving φ^k to the algorithm we get

  $$\frac{1}{c}t^k \leq C(\varphi^k) \leq ct^k$$
  $$\left(\frac{1}{c}\right)^{1/k} t \leq C(\varphi^k)^{1/k} \leq c^{1/k}t.$$

  If c is a constant and $k = O(\frac{1}{\epsilon})$, $c^{1/k} = 1 + \epsilon$.

- For a formula φ that has $O(1)$ satisfying assignments, $\#3\mathrm{SAT}(\varphi)$ can be found in $\mathbf{P}^{\mathbf{NP}}$.

  This can be done by iteratively asking the oracle the questions of the form: "Are there k assignments satisfying this formula?" Notice that these are **NP** questions, because the algorithm can guess these k assignments and check them.

10.2 An approximate comparison procedure

Consider the following approximate comparison procedure `a-comp` defined as:

$$\texttt{a-comp}(\varphi, t) = \begin{cases} \text{YES} & \text{if } \#3\mathrm{SAT}(\varphi) \geq 2t \\ \text{NO} & \text{if } \#3\mathrm{SAT}(\varphi) \leq t/2 \\ \text{whatever} & \text{otherwise} \end{cases}$$

Given `a-comp`, we can construct an algorithm that 8-approximates #3SAT as follows:

Input: φ

compute:
  $\texttt{a-comp}(\varphi, 1)$
  $\texttt{a-comp}(\varphi, 2)$
  $\texttt{a-comp}(\varphi, 4)$
  $\vdots$
  $\texttt{a-comp}(\varphi, 2^n)$

if `a-comp` outputs NO from the first time **then**
  // The value is either 0 or 1.
  // The answer can be checked by one more query to the **NP** oracle.
  Query to the oracle and output an exact value.

else
Suppose that it outputs YES for $t = 1, \ldots, 2^i$ and NO for $t = 2^{i+1}$
Output $t = 2^{i+1}$

We need to show that this algorithm approximates #3SAT with in a factor of 8. If `a-comp` answers NO from the first time, the algorithm outputs the right answer because it checks for the answer explicitly. Now suppose `a-comp` says YES for all $t = 1, 2, \ldots, 2^i$ and says NO for $t = 2^{i+1}$. Since `a-comp`$(\varphi, 2^i)$ outputs YES, $\#3\mathrm{SAT}(\varphi) > \frac{1}{2} \cdot 2^i$, and also since `a-comp`$(\varphi, 2^{i+1})$ outputs NO, $\#3\mathrm{SAT}(\varphi) < 2 \cdot 2^{i+1}$. The algorithm outputs $t = 2^{i+1}$. Hence,

$$\frac{1}{4} \cdot t < \#3\mathrm{SAT}(\varphi) < 2 \cdot t,$$

or

$$2 \cdot \#3\mathrm{SAT}(\varphi) < t < 4 \cdot \#3\mathrm{SAT}(\varphi),$$

and the algorithm outputs the correct answer with in a factor of 8.

Thus, to establish the theorem, it is enough to give a $\mathbf{BPP}^{\mathbf{NP}}$ implementation of the `a-comp`.

10.3 Constructing `a-comp`

In order to construct the procedure, we will use the property of a special kind of families of hash functions, namely, the families of pairwise independent hash function.

DEFINITION 15 *Let H be a family of functions $h \in H, h : \{0,1\}^n \to \{0,1\}^m$. H is a collection of pairwise independent hash functions if, for every $a, b \in \{0,1\}^n, a \neq b$, and for every v_1, v_2 in $\{0,1\}^m$,*

$$\mathbf{Pr}_{h \in H}[h(a) = v_1 \wedge h(b) = v_2] = \frac{1}{2^{2m}},$$

or, equivalently, for a random $h \in H$, $h(a)$ and $h(b)$ are independently and uniformly distributed in $\{0,1\}^m$.

A family of pairwise independent hash functions exists. Furthermore, if we take H to be the set of all functions $h_A : \{0,1\}^n \to \{0,1\}^m$ where $h_A(x) = Ax^T$ and $A \in \{0,1\}^{n \times m}$, then H is a family of pairwise independent hash functions. Notice that it is easy to pick a hash function from this family and it is easy to compute the function.

We also use the Leftover Hash Lemma proved by Impagliazzo, Levin, and Luby in 1989:

THEOREM 34
Let H be a family of pairwise independent hash functions $h : \{0,1\}^n \to \{0,1\}^m$. Let $S \subset 0, 1^n$, $|S| \geq \frac{4 \cdot 2^m}{\epsilon^2}$. Then,

$$\mathbf{Pr}_{h \in H}\left[\left| |\{a \in S : h(a) = 0\}| - \frac{|S|}{2^m} \right| \geq \epsilon \frac{|S|}{2^m}\right] \leq \frac{1}{4}.$$

In practice, we use hash function in a data structure for representing a subset $S \subseteq \{1,0\}^n$ by mapping each member of S to a slot on the smaller set $\{0,1\}^m$ and maintaining, for each slot, a linked list of items mapped to it. For this scheme to work efficiently, a small number of elements from S should be mapped to each slot. This theorem says that the number is about $\frac{|S|}{2^m}$. From this, `a-comp` can be constructed as follows.

input: φ, t

if $t \leq 32$ **then** check exactly whether $\#3\text{SAT}(\varphi) \geq t$.
if $t \geq 32$,
  pick h from a set of pairwise independent hash functions $h : \{0,1\}^n \to \{0,1\}^m$,
    where m is such that $2^{m+1} \geq \frac{t}{16} \geq 2^m$.
  answer YES iff there are more then $\frac{t}{2^m}$ assignments a to φ such that
    a satisfies φ and $h(a) = 0$.

Notice that since $\frac{t}{2^m} \leq 32$, the test at the last step can be done with one access to an oracle to **NP**. We will show that the algorithm is in $\mathbf{BPP}^{\mathbf{NP}}$. Let $S \subseteq \{0,1\}^n$ be the set of satisfying assignments for φ. There are 2 cases.

- If $|S| \geq 2t$, by the Leftover Hash Lemma we have:

$$\mathbf{Pr}_{h \in H}\left[\left| |\{a \in S : h(a) = 0\}| - \frac{|S|}{2^m} \right| \geq \frac{1}{2} \cdot \frac{|S|}{2^m}\right] \leq \frac{1}{4},$$

  (set $\epsilon = \frac{1}{2}$, and $|S| \geq \frac{4 \cdot 2^m}{\epsilon^2} = 16 \cdot 2^m$, because $|S| > t \geq 16 \cdot 2^m$)

$$\mathbf{Pr}_{h \in H}\left[|\{a \in S : h(a) = 0\}| \leq \frac{1}{2} \cdot \frac{|S|}{2^m}\right] \leq \frac{1}{4},$$

$$\mathbf{Pr}_{h \in H}\left[|\{a \in S : h(a) = 0\}| \leq \frac{t}{2^m}\right] \leq \frac{1}{4},$$

  which is the failure probability of the algorithm. Hence, the algorithm succeeds with probability at least 3/4.

- If $|S| \leq t/2$:

  Let S' be a superset of S of size $t/2$. We have

$$\begin{aligned}\mathbf{Pr}_{h \in H}[\text{answer YES}] &= \mathbf{Pr}_{h \in H}[|\{a \in S : h(s) = 0\}| > \tfrac{t}{2^m}] \\ &\leq \mathbf{Pr}_{h \in H}[\left|\left\{a \in S' : h(s) = 0\right\}\right| > \tfrac{t}{2^m}] \\ &\leq \mathbf{Pr}_{h \in H}\left[\left|\left|\left\{a \in S' : h(s) = 0\right\}\right| - \tfrac{|S'|}{2^m}\right| > \tfrac{|S'|}{2^m}\right] \\ &\leq \frac{1}{4}\end{aligned}$$

  (by L.H.L. with $\epsilon = 1, |S'| \geq 4 \cdot 2^m$, and $t \geq 8 \cdot 2^m$.)

Therefore, the algorithm will give the correct answer with probability at least 3/4.

10.4 The proof of the Leftover Hash Lemma

We finish the lecture by proving the Leftover Hash Lemma.

PROOF: We will use Chebyshev's Inequality to bound the failure probability. Let $S = \{a_1, \ldots, a_k\}$, and pick a random $h \in H$. We define random variables $X_1, \ldots, X_k$ as

$$X_i = \begin{cases} 1 & \text{if } h(a_i) = 0 \\ 0 & \text{otherwise.} \end{cases}$$

Clearly, $|\{a \in S : h(a) = 0\}| = \sum_i X_i$.

We now calculate the expectations. For each i, $\mathbf{Pr}[X_i = 1] = \frac{1}{2^m}$ and $\mathbf{E}[X_i] = \frac{1}{2^m}$. Hence,

$$\mathbf{E}[\sum_i X_i] = \frac{|S|}{2^m}.$$

Also we calculate the variance

$$\begin{aligned} \mathbf{Var}[X_i] &= \mathbf{E}[X_i^2] - \mathbf{E}[X_i]^2 \\ &\leq \mathbf{E}[X_i^2] \\ &= \mathbf{E}[X_i] = \frac{1}{2^m}. \end{aligned}$$

Because $X_1, \ldots, X_k$ are pairwise independent,

$$\mathbf{Var}\left[\sum_i X_i\right] = \sum_i \mathbf{Var}[X_i] \leq \frac{|S|}{2^m}.$$

Using Chebyshev's Inequality, we get

$$\begin{aligned} \mathbf{Pr}\left[\left||\{a \in S : h(a) = 0\}| - \frac{|S|}{2^m}\right| \geq \epsilon \frac{|S|}{2^m}\right] &= \mathbf{Pr}\left[\left|\sum_i X_i - \mathbf{E}[\sum_i X_i]\right| \geq \epsilon \mathbf{E}[\sum_i X_i]\right] \\ &\leq \frac{\mathbf{Var}[\sum_i X_i]}{\epsilon^2 \mathbf{E}[\sum_i X_i]^2} \leq \frac{\frac{|S|}{2^m}}{\epsilon^2 \frac{|S|^2}{(2^m)^2}} \\ &= \frac{2^m}{\epsilon^2 |S|} \leq \frac{1}{4}. \end{aligned}$$

□

Chapter 11

Valiant-Vazirani, and Introduction to Cyptography

March 7, 2001, Scribe: Dror Weitz

In the first part of this lecture we will show that solving **NP** problems in randomized polynomial time can be reduced to solving the problem Unique SAT in randomized polynomial time. In the second part of the lecture we will start an introduction to cryptography.

11.1 Unique SAT and NP

In the last lecture when we proved that $\#\mathbf{P} \subseteq \mathbf{BPP}^{\mathbf{NP}}$, we saw how we can use hash functions in order to reduce the number of satisfying assignments to a constant number. We will next see a similar, but still different, argument in order to reduce 3SAT to USAT (Unique SAT).

The problem USAT is defined as follows. Given a 3CNF formula φ,

- if φ is unsatisfiable, output NO.
- if φ has exactly one satisfying assignment, output YES.
- otherwise, any answer is acceptable.

Exercise 4 Show that if $USAT \in \mathbf{BPP}$ then $USAT \in \mathbf{RP}$.

Theorem 35 (Valiant-Vazirani)
If $USAT \in \mathbf{BPP}$ *then* $\mathbf{NP} = \mathbf{RP}$.

Open question: can the reduction be derandomized? I.e., is the following implication true: $USAT \in \mathbf{P} \Rightarrow \mathbf{P} = \mathbf{NP}$?

We now recall a definition of from the last lecture.

Definition 16 *A family of functions* H_k*, where* $h : \{0,1\}^n \to \{0,1\}^k$ *for each* $h \in H_k$*, is a family of pairwise independent hash functions if for every* $a, b \in \{0,1\}^n$ *s.t.* $a \neq b$*, for a random* $h \in H_k$*,* $h(a)$ *and* $h(b)$ *are independent, uniformly distributed in* $\{0,1\}^k$.

The following lemma was given in the last lecture:

LEMMA 36
$\{h_A\}_{A\in\{0,1\}^{n\times k}}$ where $h_A(x) = A \cdot x^T$ is a family of pairwise independent hash functions.

LEMMA 37
Let φ be a 3CNF formula with n variables and t satisfying assignments. Let k be s.t. $2^k \le t \le 2^{k+1}$. Let H_{k+2} be a family of pairwise independent hash functions where $h : \{0,1\}^n \to \{0,1\}^{k+2}$ for each $h \in H_{k+2}$. Then

$$\mathbf{Pr}\left[\left|\left\{a : a \text{ satisfies } \varphi \text{ and } h(a) = \vec{0}\right\}\right| = 1\right] \ge \frac{1}{8}$$

We will first prove Theorem 35 using the above lemmas and then go back and prove Lemma 37.

PROOF OF THEOREM 35: Suppose $USAT \in \mathbf{BPP}$, then according to Exercise 4 there is an **RP** algorithm for USAT. We can then construct the following **RP** algorithm for 3SAT. Given a formula φ, pick k at random between 0 and $n-1$, and then pick a random matrix $A \in \{0,1\}^n \times (k+2)$. Define $h : \{0,1\}^n \to \{0,1\}^{k+2}$ as $h(x) = Ax^T$. Define a formula φ' s.t. the number of satisfying assignments for φ' is equal to the number of assignments a s.t. a satisfies φ and $h(a) = \vec{0}$. Return YES iff USAT algorithm outputs YES given φ'.
We construct φ' in the following way. φ has variables $x_1, \dots, x_n$. We want $\varphi' \Leftrightarrow \varphi \wedge (Ax^T = \vec{0})$. The condition $Ax^T = \vec{0}$ amounts to:

$$\begin{array}{rcrcl} a_{11}x_1 & + \dots + & a_{1n}x_n & = & 0 \pmod 2 \\ \vdots & & \vdots & & \\ a_{k+2,1}x_1 & + \dots + & a_{k+2,n}x_n & = & 0 \pmod 2 \end{array}$$

We can enforce these $k+2$ equations by adding $(k+2)n$ new variables $y_{11}, \dots, y_{k+2,n}$ and adding clauses that for each $1 \le i \le k+2$ enforce:

$$\begin{aligned} & y_{i1} = a_{i1}x_1 \\ & y_{ij} = y_{i,j-1} + a_{ij}x_j \pmod 2 \\ & y_{in} \text{ is false} \end{aligned}$$

Thus, we added $O(kn)$ clauses with 1,2, or 3 variables each. Notice that for every assignment a to x s.t. $h(a) = \vec{0}$ there is exactly one assignment to y that satisfies all the new clauses, and thus the number of assignments that satisfies φ' is exactly the same as the number of assignments to x that satisfy φ and s.t. $h(x) = \vec{0}$.

We now analyze the correctness of the algorithm. Suppose φ has $t > 0$ satisfying assignments. Then with probability $\frac{1}{n}$ we chose a k s.t. $2^k \le t \le 2^{k+1}$. If k was chosen as such, then, according to Lemma 37, with probability $\ge \frac{1}{8}$ φ has a unique satisfying assignment a s.t. $h(a) = 0$. Thus, if $t > 0$ then with probability $\ge \frac{1}{8n}$ φ' is a YES instance of USAT. If $t = 0$ then clearly φ' has no satisfying assignment, and therefore it is a NO instance of USAT. Hence, when we apply the **RP** algorithm for USAT on φ' we will always output NO if φ is not satisfiable. If φ is satisfiable, we will output YES with probability $\geq \frac{1}{8n} \cdot \frac{1}{2} = \frac{1}{16n}$. As usual, we can enhance this probability by repeating the whole algorithm a number of times. $\Box$

Proof of Lemma 37: Recall that φ has t satisfying assignments, that $2^k \leq t \leq 2^{k+1}$, and that the hash functions are $h : \{0,1\}^n \rightarrow \{0,1\}^{k+2}$. Let a be some satisfying assignment for φ. Then

$$\begin{aligned}
&\mathbf{Pr}_{h\in H_{k+2}}[a \text{ is the unique sat. assignment of } \varphi \text{ s.t. } h(a)=\vec{0}] \quad = \\
&= \quad \mathbf{Pr}_h[h(a)=\vec{0} \wedge \forall b \neq a \;\; b \text{ sat. } \varphi \Rightarrow h(b)\neq\vec{0}] \quad = \\
&= \quad \mathbf{Pr}_h[\forall b \neq a \;\; b \text{ sat. } \varphi \Rightarrow h(b)\neq\vec{0} \mid h(a)=\vec{0}] \cdot \mathbf{Pr}_h[h(a)=\vec{0}]
\end{aligned}$$

Clearly, $\mathbf{Pr}_h[h(a)=\vec{0}] = \frac{1}{2^{k+2}}$. As for the first term,

$$\begin{aligned}
&\mathbf{Pr}_h[\forall b \neq a \;\; b \text{ sat. } \varphi \Rightarrow h(b)\neq\vec{0} \mid h(a)=\vec{0}] \quad = \\
&= \quad 1 - \mathbf{Pr}_h[\exists b \neq a \;\; b \text{ sat. } \varphi \wedge h(b)=\vec{0} \mid h(a)=\vec{0}] \quad \geq \\
&\geq \quad 1 - \sum_{b\neq a \text{ sat. } \varphi} \mathbf{Pr}_h[h(b)=\vec{0} | h(a)=\vec{0}] \quad =
\end{aligned}$$

using the pairwise independence and fact that $t \leq 2^{k+1}$ we get:

$$= \quad 1 - \sum_{b\neq a \text{ sat. } \varphi} \mathbf{Pr}_h[h(b)=\vec{0}] \quad \geq \quad 1 \; - \; 2^{k+1} \cdot \frac{1}{2^{k+2}} \quad = \quad \frac{1}{2}$$

Thus,

$$\mathbf{Pr}_h[a \text{ is the unique sat. assignment of } \varphi \text{ s.t. } h(a)=\vec{0}] \quad \geq \quad \frac{1}{2^{k+3}}$$

We can now finish the proof of the Lemma. Notice that the events "a is the unique satisfying assignment s.t. $h(a)=\vec{0}$", where a varies over all satisfying assignments, are all disjoint. Therefore,

$$\begin{aligned}
&\mathbf{Pr}_h[\varphi \text{ has a unique sat. assignment } a : h(a)=\vec{0}] \quad = \\
&= \quad \sum_{a \text{ sat. } \varphi} \mathbf{Pr}_h[a \text{ is the unique sat. assignment of } \varphi \text{ s.t. } h(a)=\vec{0}] \quad \geq \\
&\geq \quad 2^k \cdot \frac{1}{2^{k+3}} \quad = \quad \frac{1}{8}
\end{aligned}$$

$\Box$

11.2 Cryptography

We now start an introduction to Cryptography. We will discuss private-key security and pseudorandomness.

Figure 11.1 illustrates the scenario in the private-key security framework. There are two parties, conventionally called Alice and Bob, who wish to transfer messages between themselves. A third party, conventionally called Eve (for eavesdropper), tries to listen to the messages, inject new content into them, or even destroy them. Alice and Bob share a private key (a sequence of random bits) known only to them.

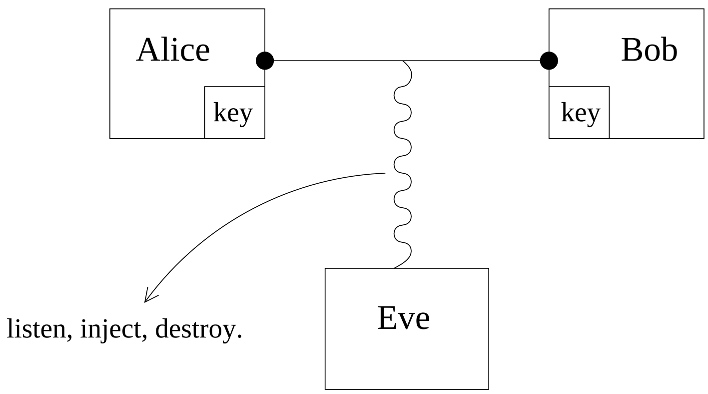

Figure 11.1: The setting for private-key security

We wish to come up with communication protocols having the following properties:

- Privacy (encryption) - Eve should have no information on the content of messages exchanged by Alice and Bob.
- Integrity (authentication) - Eve can not impersonate Alice/Bob and can not change the content of their messages.
- We will not deal with the possibility of Eve destroying the messages completely.

Constructing such protocols is related to constructing a pseudorandom generator. As illustrated in Figure 11.2, a pseudorandom generator takes a t bits long random seed and outputs an $m > t$ bits long sequence which "looks random".

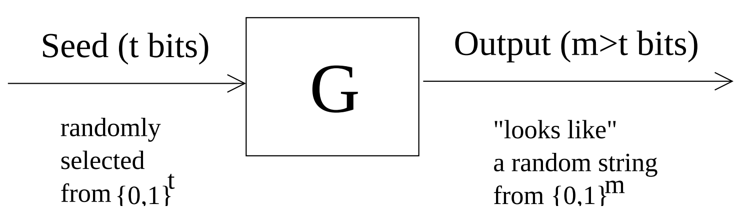

Figure 11.2: Pseudorandom generator

We now discuss what "looks random" means. We might require that the output sequence have some of the properties truly random sequences have. For instance:

- number of 0's and 1's differ by $O(\sqrt{m})$.
- the xor of a fixed subset of the bits is unbiased.
- there is a probability $\Theta(\frac{1}{m})$ that the output is a prime number.
- ...

The question is which and how many properties are enough to consider before concluding that the sequence looks random? As we will see next, we in fact consider all the possible properties which are computable under certain restrictions on the computation power.

Definition 17 *$G : \{0,1\}^t \to \{0,1\}^m$ is an (S,ϵ)-pseudorandom generator if for every property computable by a circuit C of size$\leq S$*

$$\left| \mathbf{Pr}_{x\in\{0,1\}^t}[C(G(x)) = 1] \; - \; \mathbf{Pr}_{z\in\{0,1\}^m}[C(z) = 1] \right| \; \leq \; \epsilon$$

Definition 18 *$\{G_t\}$ $G_t : \{0,1\}^t \to \{0,1\}^m$ with $m > t$ is a pseudorandom generator if G_t is computable in poly(t) time and there are functions $S(t)$,$\epsilon(t)$ s.t. S and $\frac{1}{\epsilon}$ are super-polynomial and G_t is an $(S(t),\epsilon(t))$-pseudorandom generator.*

Lemma 38
If $\mathbf{P} = \mathbf{NP}$ then there is no pseudorandom generator.

Proof: A pseudorandom generator must be some $G_t : \{0,1\}^t \to \{0,1\}^m$ with $m \geq t+1$ where G_t is computable in polynomial time. We can define the property "being a possible output of G_t". The property is in **NP** and therefore in **P**. This property is true with probability 1 on output of the generator, but with probability $\leq \frac{1}{2}$ on the uniform distribution of m bits. □

Notice that in fact, according to our definition of pseudorandom generator, even a weaker condition implies there is no pseudorandom generator, namely, if polynomial size circuits can solve all **NP** problems on average, i.e. for inverse polynomial fraction of the inputs, then there is no pseudorandom generator.

Chapter 12

One-way Functions and Pseudorandom Generators

March 12, 2001, Scribe: Jason Waddle

In this lecture we continue our discussion of pseudorandom generators. We introduce a definition of one-way functions and show how to use these to create pseudorandom generators.

12.1 Pseudorandom Generators and One-Way Functions

First, we give formal definitions of the two cryptographic primitives.

Pseudorandom Generators

A pseudorandom generator uses a small random seed to create a longer output sequence that appears random: for random inputs, the distribution of outputs is computationally difficult to distinguish from the uniform distribution.

Definition 19 (Pseudorandom Generator) *A function $G : \{0,1\}^l \to \{0,1\}^m$ with $l < m$ is (S,ϵ)-indistinguishable from uniform if for every circuit boolean circuit A of size at most S,*

$$\left|\mathbf{Pr}_{z\in\{0,1\}^m}[A(z)=1] - \mathbf{Pr}_{x\in\{0,1\}^l}[A(G(x))=1]\right| \leq \epsilon.$$

A function $G_l : \{0,1\}^l \to \{0,1\}^m$ is a pseudorandom generator if it is computable in poly(l) time and is $(S(l),\epsilon(l))$-indistinguishable from uniform, where $S(\cdot)$ and $1/\epsilon(\cdot)$ are superpolynomial.

In other words, if G is a pseudorandom generator, a size-bounded circuit can have only ϵ advantage in distinguishing the output of G from the uniform distribution on $\{0,1\}^m$.

One-Way Functions

A one-way function is easy to compute in one direction but difficult in the other.

Definition 20 (One-Way Function) *A function $f : \{0,1\}^n \to \{0,1\}^n$ is (S,ϵ)-one-way if for every circuit A of size a most S,*

$$\mathbf{Pr}_{x\in\{0,1\}^n}[f(A(f(x))) = f(x)] \leq \epsilon$$

A function $f_n : \{0,1\}^n \to \{0,1\}^n$ is a one-way function if it is computable in time poly(n) and f_n is $(S(n), \epsilon(n))$-one-way where $S(\cdot)$ and $1/\epsilon(\cdot)$ are superpolynomial.

This means that if f is a one-way function, no size-bounded circuit can use $f(x)$ to find x' such that $f(x') = f(x)$ for a large $(1-\epsilon)$ fraction of x in the domain.

12.2 The Connection between PRG and OWF

Now that the primitives are defined, we will show how they are related.

OWF from a PRG

The following result is fairly easy and intuitive, but its proof gives an example of the use of a security-preserving reduction.

Theorem 39
If there is a pseudorandom generator $G_l : \{0,1\}^l \to \{0,1\}^{2l}$, then there is a one-way function.

Proof:
Define the candidate one-way function $f : \{0,1\}^{2l} \to \{0,1\}^{2l}$ as $f(x,y) = G(x)$.

It is clear that f is computable in polynomial time since it involves only a call to the polynomial-time computable G_l.

To prove that f is one-way, we propose an adversary A that is capable of inverting f. From this we create an adversary A' for G_l, and use the reduction to determine how the security parameters for f and G_l relate.

Suppose A is a circuit of size S which inverts f on a fraction ϵ of inputs. That is,

$$\mathbf{Pr}_{x\in\{0,1\}^n}[f(A(f(x))) = f(x)] = \epsilon.$$

Define a circuit A' that attempts to determine if its input $z \in \{0,1\}^{2l}$ is from the distribution of G_l:

$(x,y) := A(z)$
if $G_l(x) = z$ **then accept**
  else reject

For a random input $z \in \{0,1\}^{2l}$, $A'(z)$ can accept only when z is in the image of G_l, that is, with probability at most $\frac{1}{2^l}$.

Consider, on the other hand, input $z = G_l(x)$ for a random x. With probability at least ϵ, $A(z)$ will find a pair (x', y') such that $f(x', y') = f(x, \cdot) = G_l(x)$. Then $G_l(x') = f(x', y') = G_l(x)$, so $A'(z)$ will accept.

A' then distinguishes G_l's output with advantage $\epsilon - \frac{1}{2^l}$:

$$\left|\mathbf{Pr}_{z\in\{0,1\}^{2l}}[A'(z) = 1] - \mathbf{Pr}_{x\in\{0,1\}^{l}}[A'(G_l(x)) = 1]\right| = \epsilon - \frac{1}{2^l}$$

To relate the adversaries, note that the circuit size of A' is essentially the circuit size of A plus the circuit size of G_l; recall that if G_l is computable in time t then it has a circuit of size $O(t^2)$.

The reduction show us that if f is not (S, ϵ)-one-way (i.e., by the existence of A), then G is not $(S+O(t^2), \epsilon-\frac{1}{2}$-indistinguishable (by our construction of A'). The contrapositive gives a positive statement: if G is (S, ϵ)-indistinguishable then f is $(S - O(t^2), \epsilon + \frac{1}{2^l})$-one-way. $\square$

PRG from a OWF

While it is straightforward to see that pseudorandom generators imply one-way functions, the converse is not as easy. The general result was shown in 1989 by Impagliazzo, Levin, and Luby; in 1990, Håstad gave a uniform reduction. These general results, however, are hard and do not have efficient reductions.

The version presented here is an earlier result by Blum, Micali, and Yao in 1982: it shows a special case where the one-way function is a permutation. It also gives an efficient reduction.

Definition 21 (One-Way Permutation) *A permutation $f : \{0,1\}^n \to \{0,1\}^n$ is (S, ϵ)-one-way if for every circuit A of size at most S,*

$$\mathbf{Pr}_{x\in\{0,1\}^n}[A(f(x)) = x] \leq \epsilon.$$

If f is also computable in poly(n) time, we say it is a one-way permutation.

Note that since f is a permutation, its output completely determines its input. Because f is one-way, however, there is no efficient way to recover the input with high probability.

It is also important to notice that while the totality of information on the input is disguised, some of the information may leak. For instance, a one-way permutation may preserve the parity of its input or simply leave a particular bit alone; this information on the input would to trivial to recover from the output.

Hard-Core Predicate

The hard-core predicate captures what information is disguised by a one-way permutation.

Definition 22 (Hard-Core Predicate) *A predicate $B : \{0,1\}^n \to \{0,1\}$ is (S,ϵ)-hard-core for a permutation $f : \{0,1\}^n \to \{0,1\}^n$ if for a circuit A of size at most S,*

$$\mathbf{Pr}_{x \in \{0,1\}^n}[A(f(x)) = B(x)] \leq \frac{1}{2} + \epsilon.$$

Exercise 5 Let the permutation $f : \{0,1\}^n \to \{0,1\}^n$ and the predicate $B : \{0,1\}^n \to \{0,1\}$ be computable in poly(n) time. Show that if B is hard-core for f_n then f_n is one-way.

To create a hard-core predicate, we need some way to extract a difficult-to-guess bit of information from a one-way permutation.

Lemma 40
If $f : \{0,1\}^n \to \{0,1\}^n$ is a (S,ϵ)-one-way permutation with superpolynomial $S, 1/\epsilon$, then there exist S' and ϵ' such that for every circuit A of size at most S',

$$\mathbf{Pr}_{x,r \in \{0,1\}^n}[A(f(x), r) = x \odot r] \leq \frac{1}{2} + \epsilon',$$

where the inner product $x \odot r = \sum_i x_i r_i \pmod 2$.

Proof: *Omitted.* □

Theorem 41 (Yao 1982, Goldreich and Levin 1989)
If there is a one-way permutation, then there is another one-way permutation with a hard-core predicate.

Proof:
Lemma 40 does not immediately reconcile with the definition of a hard-core predicate. We need a permutation that provides r:

If $f : \{0,1\}^n \to \{0,1\}^n$ is a one-way permutation, then $f' : \{0,1\}^{2n} \to \{0,1\}^{2n}$ where $f(x,r) = (f(x), r)$ is also a one-way permutation. Moreover, $B : \{0,1\}^{2n} \to \{0,1\}$ where $B(y,r) = y \odot r$ is a hard-core predicate for f'.
□

Pseudorandom Generator

The existence of a hard-core predicate allows us to create a security-preserving but length-extending primitive: a pseudorandom generator.

Theorem 42
If there is a one-way permutation, then there is a pseudorandom generator.

Proof:
Since there is a one-way permutation, Lemma 40 tells us that there is another one-way permutation, $f : \{0,1\}^n \to \{0,1\}^n$ with a corresponding hard-core predicate $B : \{0,1\}^n \to \{0,1\}$.

Define $G_n : \{0,1\}^n \to \{0,1\}^{n+1}$ as $G_n(x) = (f(x), B(x))$.

To see that G_n is computationally indistinguishable from uniform, suppose that there is some boolean distinguisher circuit D such that

$$\left|\mathbf{Pr}_{x\in\{0,1\}^n}[D(G_n(x)) = 1] - \mathbf{Pr}_{z\in\{0,1\}^{n+1}}[D(z) = 1]\right| \geq \epsilon.$$

Since D and $1 - D$ have basically the same complexity, we can assume that D distinguishes G_n's distribution with positive correlation. That is,

$$\mathbf{Pr}_{x\in\{0,1\}^n}[D(G_n(x)) = 1] - \mathbf{Pr}_{z\in\{0,1\}^{n+1}}[D(z) = 1] \geq \epsilon.$$

We will define an adversary, A, for f's hard-core predicate, B:

input: $y \in \{0,1\}^n$
  pick at random $b \in \{0,1\}$
  $v := D(y, b)$
  if $v = 1$ **then** output b
    else output $1 - b$

Intuitively, A picks b as a guess of the value of $B(x)$. The guess is verified with a call to $D(y, b)$. If the guess was good and D accepts, A outputs B; otherwise, if D rejects, A outputs b's complement.

We wish to show that $A(G_n(x))$ predicts $B(x)$ with high probability over the choices of x and b.

$$\begin{aligned}
\mathbf{Pr}[A(G_n(x)) = B(x)] = \quad & \mathbf{Pr}[A(f(x)) = B(x)|b = B(x)]\mathbf{Pr}[b = B(x)] \\
& + \mathbf{Pr}[A(f(x)) = B(x)|b \neq B(x)]\mathbf{Pr}[b \neq B(x)]
\end{aligned}$$

and since x and b are independent

$$\begin{aligned}
&= \frac{1}{2}\mathbf{Pr}[A(f(x)) = B(x)|b = B(x)] + \frac{1}{2}\mathbf{Pr}[A(f(x)) = B(x)|b \neq B(x)] \\
&= \frac{1}{2}\mathbf{Pr}[D(f(x), b) = 1|b = B(x)] + \frac{1}{2}\mathbf{Pr}[D(f(x), b) = 0|b \neq B(x)] \\
&= \frac{1}{2}\mathbf{Pr}[D(f(x), b) = 1|b = B(x)] + \frac{1}{2} - \frac{1}{2}\mathbf{Pr}[D(f(x), b) = 1|b \neq B(x)] \\
&= \frac{1}{2} + \mathbf{Pr}[D(f(x), b) = 1|b = B(x)] \\
&\quad - \frac{1}{2}\left(\mathbf{Pr}[D(f(x), b) = 1|b = B(x)] + \mathbf{Pr}[D(f(x), b) = 1|b \neq B(x)]\right) \\
&= \frac{1}{2} + \mathbf{Pr}[D(f(x), B(x)) = 1] - \mathbf{Pr}[D(f(x), b) = 1]
\end{aligned}$$

$G_n(x) = (f(x), B(x))$, and $(f(x), b)$ is uniformly distributed in $\{0,1\}^{n+1}$, so

$$\begin{aligned}
&= \frac{1}{2} + \mathbf{Pr}[D(G_n(x)) = 1] - \mathbf{Pr}_{z \in \{0,1\}^{n+1}}[D(z) = 1] \\
&\geq \frac{1}{2} + \epsilon
\end{aligned}$$

Thus A has a good (ϵ) advantage in predicting B, and its complexity is basically the same as D's. Thus, analogous to the proof of Theorem 39, if f is a one-way permutation, then G_n is a pseudorandom generator. $\square$

In the next lecture we will see how to extend the output of our PRG while preserving security.

Chapter 13

Pseudorandom Generators and Pseudorandom Functions

March 14, 2001, Scribe: Kunal Talwar

In this lecture, we start with a construction of a pseudorandom generator using a hard core predicate for a one-way function. We then define the concept of a pseudorandom function families and give a construction of one. Finally we show how these concepts could be used to build a secure private key cryptosystem.

13.1 A Pseudorandom generator

In the last lecture, we showed that if $f : \{0,1\}^n \to \{0,1\}^n$ is a one-way permutation and $B : \{0,1\}^n \to \{0,1\}$ is a hard core predicate for f, then $G : \{0,1\}^n \to \{0,1\}^{n+1}$ defined by

$$G(x) = f(x)B(x)$$

is a pseudorandom generator. We now extend that construction to get a pseudorandom generator $G^k : \{0,1\}^n \to \{0,1\}^{n+k}$ (see fig. 26.1), defined as

$$G^k(x) = B(x)B(f(x)) \dots B(f^{k-1}(x)).f^k(x)$$

Theorem 43
G_k defined as above is a pseudorandom generator.

Proof: We use proof by contradiction. Assume the contrary, i.e. suppose there is an algorithm D' such that

$$\left| \mathbf{Pr}_r[D'(r) = 1] - \mathbf{Pr}_x[D'(G^k(x)) = 1] \right| > \epsilon \tag{13.1}$$

then, as shown in the last lecture, there is a D of similar complexity such that

$$\mathbf{Pr}_r[D(r) = 1] - \mathbf{Pr}_x[D(G^k(x)) = 1] > \epsilon \tag{13.2}$$

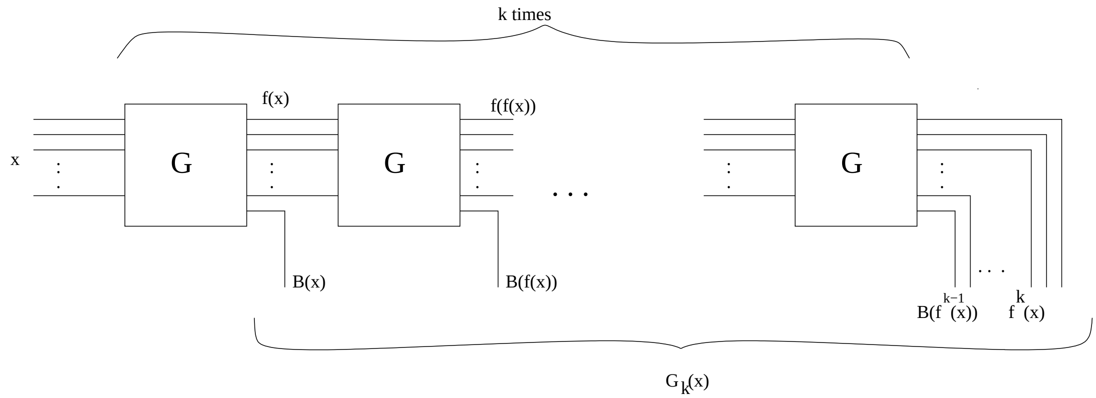

Figure 13.1: Pseudorandom generator

Now we use a technique called the hybrid argument, originally due to Goldwasser, Micali and first used in the context of pseudorandom generators by Blum and Micali. We define a series of distributions $H_0, H_1 \dots, H_k$ such that H_i and H_{i+1} are close and the first and the last are the two distribtions distinguished by D above.

Pick $x \in \{0,1\}^n, (r_1, \dots, r_n) \in \{0,1\}^k$

$$
\begin{aligned}
H_0 &= B(x)B(f(x))\dots B(f^{k-1}(x)).f^k(x) && \approx \text{output of} G^k \\
H_1 &= r_1 B(f(x))\dots B(f^{k-1}(x)).f^k(x) \\
H_2 &= r_1 r_2 \dots B(f^{k-1}(x)).f^k(x) \\
&\vdots \\
H_k &= r_1 r_2 \dots r_k.f^k(x) && \approx \text{uniform distribution}
\end{aligned}
$$

Using (13.2) and the definition of H_i's above,

$$\mathbf{Pr}[D(H_0) = 1] - \mathbf{Pr}[D(H_k) = 1] > \epsilon \tag{13.3}$$

If we define $p_i = \mathbf{Pr}[D(H_i) = 1]$, then (13.3) gives

$$
\begin{aligned}
\epsilon &= p_0 - p_k && (13.4)\\
&= (p_0 - p_1) + (p_1 - p_2) + \dots + (p_{k-1} - p_k) && (13.5)
\end{aligned}
$$

Thus, there exists an i such that

$$\mathbf{Pr}[D(H_{i-1}) = 1] - \mathbf{Pr}[D(H_i) = 1] > \frac{\epsilon}{k} \tag{13.6}$$

Now recollect that

$$
\begin{aligned}
H_{i-1} &= r_1 r_2 \dots r_{i-1} B(f^{i-1}(x))B(f^i(x))\dots B(f^{k-1}x)).f^k(x) && (13.7)\\
H_i &= r_1 r_2 \dots r_{i-1} r_i B(f^i(x))\dots B(f^{k-1}x)).f^k(x) && (13.8)
\end{aligned}
$$

Using D, we now construct an algorithm A that distinguishes $G(x)$ from uniform.

input: $y.b//y = f(z)$for a random $z, b \in \begin{cases} \text{a random bit} \\ B(z) \end{cases}$

pick at random $r_1, \ldots, r_{i-1}$.
output $D(r_1 \ldots r_{i-1} b B(y) B(f(y)) \ldots B(f^{k-i-1}(y)) f^{k-i}(y))$

To show that A distinguishes $f(z)B(z)$ from uniform, we show that

- If A recieves $f(z)B(z)$ as input, then A feeds H_{i-1} into D.
- If A recieves a random input, then A feeds H_i into D.

Note that input to D in the first case is

$$r_1 \ldots r_{i-1} B(z) B(f(z)) B(f^2(z)) \ldots B(f^{k-i}(z)) f^{k-i+1}(z)$$

Since $f^{i-1}(x)$ has the same distribution as z when x and z are random and f is a permutation, the first part of the claim holds. The input to D in the second case is

$$r_1 \ldots r_{i-1} b B(y) B(f(y)) \ldots B(f^{k-i-1}(y)) f^{k-i}(y)$$

Again since $f^i(x)$ has the same distribution as y when x and y are random and f is a permutation, and since b is random, the claim holds. Thus if we have a D that ϵ-distinguishes $G^k(x)$ from uniform, then we have an A that $\frac{\epsilon}{k}$-distinguishes $f(z)B(z)$ from uniform. However since B was a hard core predicate, there is no such A. Thus the theorem holds. $\square$

13.2 Pseudorandom function families

The construction described in the last section was a sequential composition of generators. We can also have a parallel composition as in Figure 13.2(a).

We observe an interesting property of this pseudorandom generators. Let G be computable in time $O(n^c)$. Set $k \geq c \log n$. Then each bit of the output is computable in time $k \cdot O(n^c) = O(n^c \log n)$. The length of the output of the generator is $2^k n = O(n^{c+1})$. Thus we have a long sequence of pseudorandom bits such that any one of them can be evaluated in time sublinear in the length of the string. Note that this cannot be the case for a sequential composition of pseudorandom generators.

Now consider the case with $k = n$. This construction does not satisfy the traditional definition of a pseudorandom generator since the output size is exponential in the input size and hence it cannot run in polynomial time. We now define the concept of a pseudorandom function family. Let $\mathcal{F} = \{f_s\}_{s \in \{0,1\}^k}$ be a family of functions such that $f_s : \{0,1\}^n \rightarrow \{0,1\}^n$ for every s. Let A be any algorithm that queries an oracle on some set of inputs (see figure 13.3). The oracle can be one of the following. It either computes the function $f_s(x)$ for some s chosen uniformly at random from $\mathcal{F}$, or computes a function $F(x)$ chosen uniformly at random from the set of all functions from $\{0,1\}^n \rightarrow \{0,1\}^n$. If no small sized circuit A can differentiate these two cases, then $\mathcal{F}$ is called a pseudorandom function family. More formally,

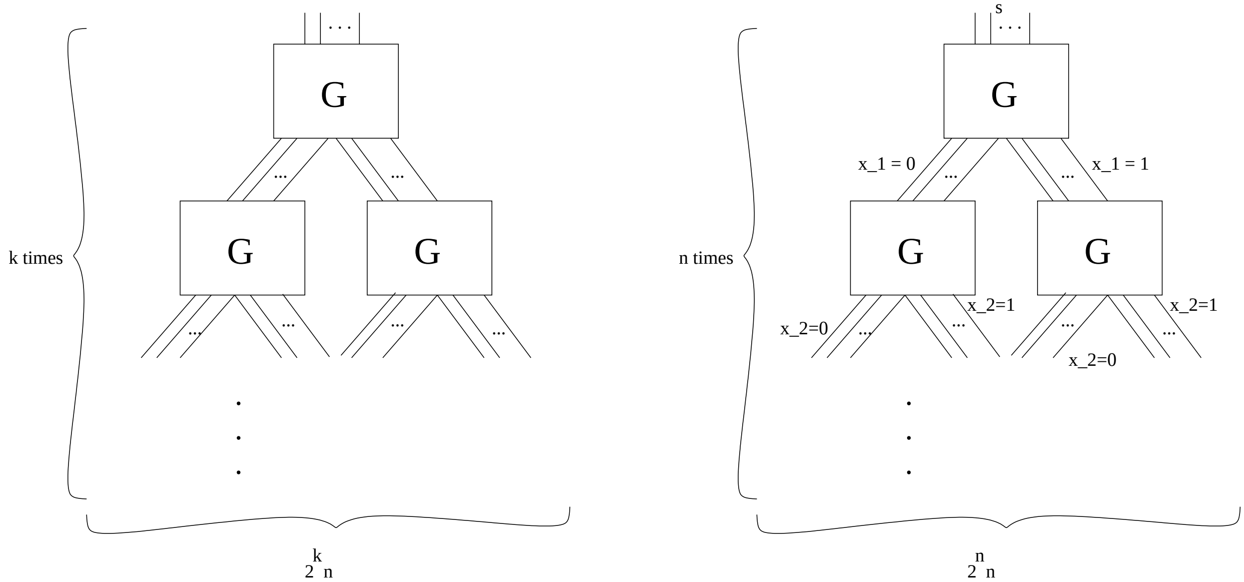

Figure 13.2: (a) Parallel composition of pseudorandom generators. (b) Pseudorandom function family

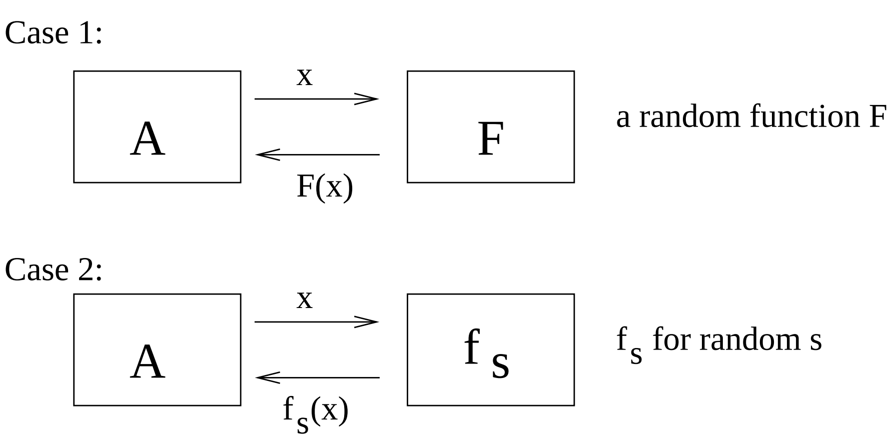

Figure 13.3: Oracle model

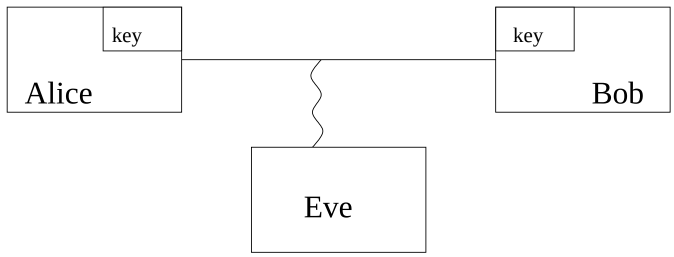

Figure 13.4: Communication model

Definition 23 *$\{f_s\}_{s\in\{0,1\}^k}, f_s : \{0,1\}^n \to \{0,1\}^n$ is called a family of (S,ϵ) pseudorandom functions if for every* oracle circuit *A of size S,*

$$\left|\mathbf{Pr}_{F:\{0,1\}^n\to\{0,1\}^n}[A^F = 1] - \mathbf{Pr}_{s\in\{0,1\}^k}[A^{f_s} = 1]\right| \leq \epsilon$$

Asymptotically,

Definition 24 *Let $\{f_s\}_{s\in\{0,1\}^k}, f_s : \{0,1\}^n \to \{0,1\}^n$, where $k = O(n^c)$ for some c. Further let $f_s(x)$ be computable in time polynomial in n, and let $\{f_s\}$ be an $(S(n),\epsilon(n))$ pseudorandom function family, for super-polynomial functions S and $\frac{1}{\epsilon}$. Then $\{f_s\}_{s\in\{0,1\}^k}$ is a family of pseudorandom functions.*

If $G : \{0,1\}^n \to \{0,1\}^{2n}$ is a pseudorandom generator, then the tree construction gives a pseudorandom function family in the following way. The seed s is the input to the first pseudorandom generator. The input x specifies a path in the tree (See figure 13.2(b)). More formally, define

$$G_0(s) = \text{first } \ n \ \text{ bitsof } \ G(s) \ \text{ and}$$

$$G_1(s) = \text{bits } \ n+1, n+2, \ldots, 2n \ \text{ of } \ G(s)$$

then, $f_s : \{0,1\}^n \to \{0,1\}^n$ for any $s \in \{0,1\}^n$, is defined as

$$f_s(x) = G_{x_n}(\ldots G_{x_2}(G_{x_1}(s))\ldots)$$

This construction is due to Goldreich, Goldwasser and Micali.

13.3 Cryptography

We now apply the concepts of pseudorandomness to private and secure communication. Consider the scenario in figure 13.4. Alice and Bob want to communicate using an insecure channel. An eavesdropper Eve can listen, possibly corrupt and send malicious messages. We want

Privacy : Eve must get no information about the messages being exchanged.

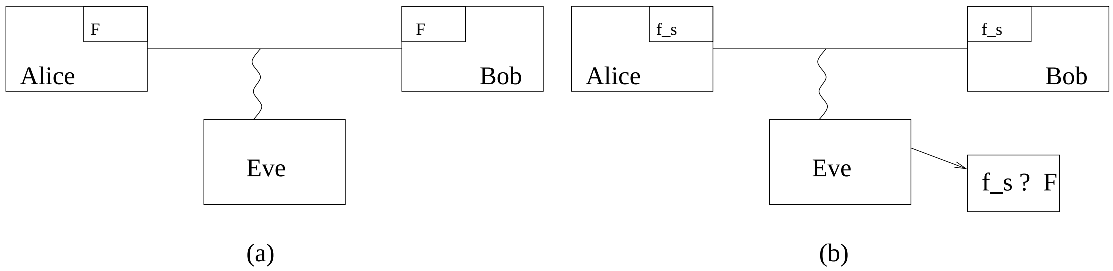

Figure 13.5: (a) Ideal world model (b) Real world

Integrity : Eve should not be able to modify messages or to impersonate Alice/Bob.

We first consider a simple threat model where Eve can only read the channel, but not modify/write anything, and give a protocol that guarantees privacy. We then consider a stronger threat model and give a protocol that guarantees both privacy an integrity.

Idealized world

We first look at an idealized scenario where Alice and Bob share a completely random function $F : \{0,1\}^n \rightarrow \{0,1\}^n$ (see figure 13.5(a)). Note that both Alice and Bob share this *private key*, which is communicated using some other mechanism which is assumed to be secure. The protocol for communication is the following

Alice	Bob
message $m \in \{0,1\}^n$	
picks an $r \in \{0,1\}^n$ randomly	
sends $r.(F(r) \oplus m)$	
	recieves $r.z$
	computes $m = F(r) \oplus z$

From the point of view of Eve, what she sees is a random string: r is uniform by choice. Since the function F was chosen uniformly at random, $F(r)$ is uniformly distributed for any arbitrary r, and hence $F(r) \oplus m$ is also uniformly distributed. Thus any two messages with distinct r appear completely random to Eve, and give no information. When an r gets repeated, he can get the xor of the two messages. However, if Alice and Bob exchange polynomially many messages, the probabilty of that happening is exponentially small.

Real world

In a real scenario, the random function F is replaced by a function f_s from a pseudorandom function family, and r and s are pseudorandom strings (figure 13.5(b)). We claim that Eve cannot distinguish what it sees from uniform, and hence get no information about the message. This must be the case because if Eve could distinguish f_s from F, we can simulate the scenario to construct an algorithm A that distinguishes f_s from F.

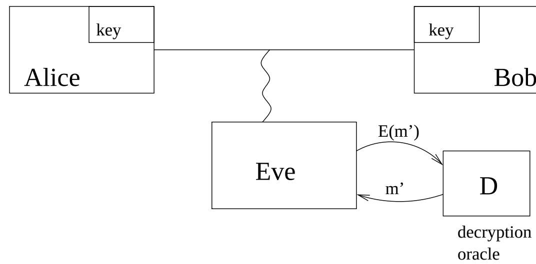

Figure 13.6: Stronger threat model

Stronger threat model

We now consider a stronger adversary who has access to a decryption procedure. The decryption oracle D returns the decryption of any message that has not been communicated between Alice and Bob (see figure 13.6). Note that this model captures adaptive chosen ciphertext attacks. In this model, the previous protocol fails immediately, since Eve on reading $r.z$ from the channel, can ask for decryption of $r.z'$ where z' differs from z only in the last bit, and get back m' that is same as m with the last bit flipped.

A simple fix is to send $r.(F(r) \oplus m).F(F(r) \oplus m)$. Now the probability that a message constructed by Eve is properly formatted is exponentially small. Thus with little error, we can simulate the new protocol with one where D always answers "Not well formatted", and this model is equivalent to the one without D. Also note that since a message constructed by Eve is invalid with very high probability, Eve cannot write valid message on the channel, and hence integrity is also guaranteed.

Thus this construction is secure against adaptive chosen ciphertext attack. Similary, we can define a protocol that is secure against even stronger threat models like chosen ciphertext and chosen plaintext attacks.

Chapter 14

Goldreich-Levin

March 19, 2001, Scribe: Mark Pilloff

Today we will prove the Goldreich-Levin theorem:

THEOREM 44
If $f : \{0,1\}^n \rightarrow \{0,1\}^n$ is a one-way permutation then $B(x,r) = \sum_i x_i r_i \pmod 2$ is a hardcore predicate for $f'(x,r) = f(x), r$.

In other words, given only $f(x)$ and r, it is hard to compute $\sum x_i r_i$. Before proving the above theorem, we establish some preliminary results.

14.1 Linear Boolean Functions on $\{0,1\}^n$

Consider a $2^n \times 2^n$ matrix M where each row and column is associated with an n-bit string: $a \in \{0,1\}^n$ and $x \in \{0,1\}^n$ respectively. Let the matrix element at position (a,x) be given by

$$M_{ax} = a \cdot x. \tag{14.1}$$

Then each row of this matrix corresponds to a linear function $f_a : \{0,1\}^n \rightarrow \{0,1\}$ where $f_a(x) = ax$. In fact, all 2^n possible linear boolean functions on $\{0,1\}^n$ are represented in the rows of this matrix.

LEMMA 45
For all $a \neq b, |\{x : ax \neq bx\}| = \frac{1}{2}2^n$.

PROOF:

$$\begin{aligned} |\{x : ax \neq bx\}| &= |\{x : ax \oplus bx = 1\}| && (14.2) \\ &= |\{x : (a \oplus b)x = 1\}| && (14.3) \\ &= \frac{1}{2}2^n && (14.4) \end{aligned}$$

where the last equality follows from symmetry and the fact that $a \oplus b \neq 0$. □

14.2 Error Correcting Codes (ECC)

The preceding lemma implies that any two rows of M differ in half of their values. This makes the rows of M well-suited as codewords for an error correcting code.

DEFINITION 25 $\mathcal{C} \subseteq \{0,1\}^m$ *is an error correcting code with minimum distance d if for any $y, z \in \mathcal{C}$, y and z differ in at least d positions.*

Here we have $\mathcal{C} \subseteq \{0,1\}^{2^n}$ with $d = 2^{n-1}$. This code is known as a Reed-Muller code or a Hadamard code.

The utility of such a code is that a sender (Alice) wishes to communicate with a receiver (Bob) over a possibly unreliable channel. To send the value $a \in \{0,1\}^n$, Alice sends to Bob the function $f_a(x) = ax$ (more precisely, she sends the value of f_a at all 2^n values of x). In other words, Alice sends row a of M which is 2^n bits long. Because the channel is unreliable, Bob receives the function $g(x)$ which may not be the same as $f_a(x)$. However, as we shall demonstrate shortly, provided that $g(x)$ and $f_a(x)$ agree on a fraction $\frac{3}{4} + \epsilon$ of the values in the domain, Bob can always recover $f_a(x)$ and therefore a.

LEMMA 46
Suppose that in the above scenario f_a and g disagree on at most a fraction $\frac{1}{4} - \epsilon$ of the values in the domain. Then for all $b \neq a$, f_b and g disagree on at least $\frac{1}{4}$ of the values in the domain. Hence, a is uniquely determined and can be recovered.

PROOF: Suppose by way of contradiction that there exists $b \neq a$ such that f_b and g disagree on less than $\frac{1}{4}$ of the values in the domain. Then f_a and f_b disagree on less than $\frac{1}{4} - \epsilon + \frac{1}{4} < \frac{1}{2}$ of the values in the domain. But this is impossible for $a \neq b$ by Lemma 45. □

14.2.1 Efficient Decoding

This leaves unanswered the question of how Bob can actually recover a from the function g. Naively, this can be done in time quadratic in 2^n by trying all functions f_a on all inputs x until the answer is found. The following theorem, however, indicates that a much more efficient algorithm exists.

THEOREM 47
Let $g : \{0,1\}^n \to \{0,1\}$ be such that $g(x)$ agrees with $f_a(x)$ in $\frac{3}{4} + \epsilon$ of the x values. Then with oracle access to g, we can compute a in time $O(n \log n/\epsilon^2)$.

PROOF: Let e_i be row i of the identity matrix. That is, $e_i = (00\ldots010\ldots00)$ where the 1 occurs in position i. Then for any x

$$\begin{aligned} a_i &= ae_i && (14.5)\\ &= ae_i \oplus ax \oplus ax && (14.6)\\ &= a(e_i \oplus x) \oplus ax && (14.7) \end{aligned}$$

Now consider the following operation: choose a random $x \in \{0,1\}^n$ and compute $g(e_i+x) \oplus g(x)$. With probability $\frac{3}{4}+\epsilon$, $g(e_i+x) = a(e_i+x)$ and with probability $\frac{3}{4}+\epsilon$, $g(x) = a(x)$. It then follows that both equalities hold with probability at least $\frac{1}{2}+2\epsilon$ and we conclude

$$\mathbf{Pr}_{x\in\{0,1\}^n}[g(e_i+x) \oplus g(x) = a_i] \geq \frac{1}{2}+2\epsilon. \tag{14.8}$$

This suggests the following algorithm:

Algorithm $\frac{3}{4}+\epsilon$**:**
 for $i := 1$ to n **do**
  choose $k = O(\log n/\epsilon^2)$ points $x_1, \ldots, x_k \in \{0,1\}^n$
  compute:
   $g(x_1+e_1) \oplus g(x_1)$
   $g(x_2+e_2) \oplus g(x_2)$
   $\vdots$
   $g(x_k+e_k) \oplus g(x_k)$
  assign to a_i the value occurring in the majority of these computations

To analyze this program, note that for a particular value of i, we expect to get the right answer in a fraction $\frac{1}{2}+2\epsilon$ of the k trials. Then by a Chernoff bound, the probability of estimating a_i incorrectly is $\exp -\Omega(\epsilon^2 k)$. We can then choose $k = O(\log n/\epsilon^2)$ such that this probability is bounded by $\frac{1}{n^2}$ and hence we conclude

$$\mathbf{Pr}[\text{any of the } a_i \text{ is incorrectly estimated}] \leq \frac{1}{n}. \tag{14.9}$$

We complete the proof by noting that the running time of this program is $O(nk) = O(n \log n/\epsilon^2)$. $\square$

14.2.2 Dealing With Noisy Channels

All of the above assumes that the received codeword g differs from the sent codeword f_a on less than one quarter of the values in the domain. If g differs from f_a on a faction greater than or equal to $\frac{1}{4}$ of the values in the domain then it is in general impossible to uniquely correct g to obtain f_a. In particular, there may be more than one value b for which the distance between f_b and g is a minimum. In such a situation, the best we can do is to find a list of codewords which disagree with g in no more than a specified number of locations. The size of such a list is bounded by the following theorem which we state without proof:

THEOREM 48
For every $g : \{0,1\}^n \to \{0,1\}$ there are at most $\frac{1}{4\epsilon^2}$ values $a \in \{0,1\}^n$ such that $g(x)$ and $f_a(x)$ agree in at least fraction $\frac{1}{2}+\epsilon$ of their values.

We will shortly provide an algorithm to prove the following:

Theorem 49

Let $g : \{0,1\}^n \to \{0,1\}$ and ϵ be given. Then in $O(n \log n/\epsilon^4)$ time, we can construct a list of size $O(1/\epsilon^2)$ containing elements of $\{0,1\}^n$ such that the list contains every $a \in \{0,1\}^n$ such that $f_a(x)$ has agreement $\geq \frac{1}{2} + \epsilon$ with $g(x)$.

In view of theorem 48, we note that the list constructed according to theorem 49 is at most a constant factor larger than optimal.

Proof: Fix a such that $g(x)$ and $f_a(x)$ have agreement at least $\frac{1}{2}+\epsilon$. Our goal is to reduce this problem to the $\frac{3}{4}+\epsilon$ case. First choose $x_1 \ldots x_k \in \{0,1\}^n$ where $k = O(1/\epsilon^2)$. For the moment, let us suppose that we have "magically" obtained the values $ax_1, \ldots, ax_k$. Then define $\hat{g}(z)$ as the majority value of:

$$(ax_i) \oplus g(z + x_i) \qquad i = 1, 2, \ldots, k \tag{14.10}$$

Since for each i we obtain az with probability at least $\frac{1}{2}+\epsilon$, by choosing k sufficiently large we can ensure that

$$\mathbf{Pr}_{z,x_1,\ldots,x_k}\left[\hat{g}(z) = az\right] \geq \frac{31}{32}. \tag{14.11}$$

from which it follows that

$$\mathbf{Pr}_{x_1,\ldots,x_k}\left[\mathbf{Pr}_z\left[\hat{g}(z) = az\right] \geq 7/8\right] \geq \frac{3}{4}. \tag{14.12}$$

Consider the following algorithm:

Algorithm First-Attempt:
  choose $x_1, \ldots, x_k \in \{0,1\}^n$ where $k = O(1/\epsilon^2)$
  for all $b_1, \ldots, b_k \in \{0,1\}$
    define $\hat{g}_{b_1 \ldots b_k}(z)$ as majority of: $b_i \oplus g(z + x_i)$
    apply Algorithm $\frac{3}{4}+\epsilon$ to uniquely decode $\hat{g}_{b_1 \ldots b_k}$
    add result to list

The idea behind this program is that we don't in fact know the values of the ax_i so we guess all possibilities by considering all choices for the b_i. For each a such that f_a and g agree on more than half of their domain, we will eventually choose $b_i = ax_i$ for all i and then, with high probability, recover a via Algorithm $\frac{3}{4}+\epsilon$. The obvious problem with this algorithm is that its running time is exponential in $k = O(1/\epsilon^2)$ and the resulting list may also be exponentially larger than the $O(1/\epsilon^2)$ bound provided by Theorem 48.

To overcome these problems, consider the following similar algorithm:

Algorithm $\frac{1}{2}+\epsilon$:
  choose $x_1, \ldots, x_t \in \{0,1\}^n$ where $t = O\left(\log(1/\epsilon)\right)$
  for all $b_1, \ldots, b_t \in \{0,1\}$
    define $\hat{g}_{b_1 \ldots b_t}(z)$ as majority over all nonempty $S \subseteq \{1, \ldots, t\}$ of: $(\oplus_{i \in S} b_i) \oplus g\left(z + \sum_{i \in S} x_i\right)$
    apply Algorithm $\frac{3}{4}+\epsilon$ to uniquely decode $\hat{g}_{b_1 \ldots b_k}$
    add result to list

Let us now see why this algorithm works. First we define, for any nonempty $S \subseteq \{1, \ldots, t\}$, $x_S \equiv \sum_{i \in S} x_i$. Then since $x_1, \ldots, x_t \in \{0,1\}^n$ are random, it follows that for any $S \neq T$, x_S and x_T are independent and uniformly distributed. Now consider any a such that $f_a(x)$ and $g(x)$ agree on $\frac{1}{2} + \epsilon$ of the values in their domain. Then for the choice of $\{b_i\}$ where $b_i = ax_i$ for all i, we have that

$$\bigoplus_{i \in S} b_i = ax_S, \tag{14.13}$$

and, with probability $\frac{1}{2} + \epsilon$,

$$\begin{aligned} g\left(z + \sum_{i \in S} x_i\right) &= g(z + x_S) && (14.14) \\ &= a(z + x_s) = az + ax_S, && (14.15) \end{aligned}$$

so combining the above results yields

$$\bigoplus_{i \in S} b_i + g\left(z + \sum_{i \in S} x_i\right) = az \tag{14.16}$$

with probability $\frac{1}{2} + \epsilon$.

Note the following simple lemma whose proof we omit:

Lemma 50

Let $R_1, \ldots, R_k$ be a set of pairwise independent $0 - 1$ random variables, each of which is 1 with probability at least $\frac{1}{2} + \epsilon$. Then $\mathbf{Pr}[\sum_i R_i \geq k/2] \geq 1 - O(1/\epsilon^2 k)$.

Lemma 50 allows us to upper-bound the probability that the majority operation used to compute $\hat{g}$ gives the wrong answer. Combining this with our earlier observation that the $\{x_S\}$ are pairwise independent, we see that choosing $t = 2 \log 1/\epsilon + O(1)$ suffices to ensure that $\hat{g}_{b_1 \ldots b_t}(z) = az$ with probability $1 - c \geq \frac{3}{4} + \epsilon$ for any constant $c > 0$. Thus we can use Algorithm $\frac{3}{4} + \epsilon$ to obtain a with high probability. Choosing t as above ensures that the list generated is of length at most $2^t = O(1/\epsilon^2)$ and the running time is then $O(n \log n/\epsilon^4)$ (due to the $O(1/\epsilon^2)$ iterations of Algorithm $\frac{3}{4} + \epsilon$). This completes the proof of Theorem 49. $\square$

14.3 Hardcore Predicates

We are now in a position to prove the Goldreich-Levin theorem stated at the beginning of these notes:

Theorem 51

If $f : \{0,1\}^n \to \{0,1\}^n$ is a one-way permutation then $B(x, r) = \sum_i x_i r_i \pmod 2$ is a hardcore predicate for $f'(x, r) = f(x), r$.

PROOF: Suppose A is such that

$$\mathbf{Pr}_{x,r}[A(f(x),r) = xr] > \frac{1}{2} + \epsilon. \tag{14.17}$$

from which it follows that

$$\mathbf{Pr}_x\left[\mathbf{Pr}_r[A(f(x),r) = xr] \geq \frac{1}{2} + \frac{\epsilon}{2}\right] \geq \frac{\epsilon}{2}. \tag{14.18}$$

For a fixed x, define $g(r) \equiv A(f(x),r)$. With probability at least $\epsilon/2$ there is a linear function $f_x(r) = xr$ which agrees with $g(r)$ on at least $\frac{1}{2} + \epsilon$ of the values in the domain. Hence we can use the preceding algorithm on g with parameter $\epsilon/2$ to construct a list of size $O(1/\epsilon^2)$. By applying f to each item in the list, we can find a preimage x' of $f(x)$ (ie, $f(x') = f(x)$). Hence we are able to invert f for a fraction $\epsilon/2$ of its domain, contradicting the assumption that f is one-way. Thus $B(x,r)$ must be hardcore. $\square$

Chapter 15

Levin's Theory of Average-Case Complexity

March 21, 2001, Scribe: Scott Aaronson

This lecture introduces Levin's theory of average-case complexity. The theory is difficult to use and is still useless, in that it doesn't yet allow us to prove anything interesting about important problems. But eventually complexity theory ought to address how hard problems are to solve on average, and Levin's theory is significant as a step in this direction. Levin presented the theory in a one-page paper in FOCS'84 and in a two-pages SICOMP paper [Lev86].

We first define the model—itself a nontrivial issue—and then we prove the existence of a problem that is complete for the average-case class **NP**.

15.1 Distributional Problems

A *distributional problem* is a pair $\langle L, \mu \rangle$, where L is a decision problem and μ is a distribution over the set $\{0,1\}^*$ of possible inputs. Importantly, μ is *not* a collection of separate distributions, one for each input size. This is because reductions do not in general preserve input sizes.

We want μ to be polynomial-time computable. What does this mean? For all $x \in \{0,1\}^*$, let

$$\mu(x) = \sum_{y \leq x} \Pr[y]. \tag{15.1}$$

where '$\leq$' denotes lexicographic ordering. Then μ must be computable in $\operatorname{poly}(|x|)$ time. Clearly this notion is at least as strong as the requirement that $\Pr[x]$ be computable in polynomial time, for

$$\Pr[x] = \mu\prime(x) = \mu(x) - \mu(x-1), \tag{15.2}$$

$x-1$ being the lexicographic predecessor of x. Indeed one can show that if $\mathbf{P} \neq \mathbf{NP}$, then there exist distributions that are efficiently computable in the second sense but not polynomial-time computable in our sense.

We can define the 'uniform distribution' to be

$$\Pr[x] = \frac{1}{|x|(|x|+1)} 2^{-|x|}; \tag{15.3}$$

that is, first choose an input size at random under some polynomially-decreasing distribution, then choose an input of that size uniformly at random. It is easy to see that the uniform distribution is polynomial-time computable.

15.2 DistNP

We define the complexity class

$$\mathbf{DistNP} = \{\langle L, \mu \rangle : L \in NP, \mu \text{ polynomial-time computable}\}. \tag{15.4}$$

There are two justifications for looking only at polynomial-time computable distributions.

1. One can show that there exists a distribution μ such that every problem is as hard on average under μ as it is in the worst case. Therefore, unless we place some computational restriction on μ, the average-case theory is identical to the worst-case one.

2. Someone, somewhere, had to generate the instances we're trying to solve. If we place computational restrictions on ourselves, then it seems reasonable also to place restrictions on whoever generated the instances.

It should be clear that we need a whole *class* of distributions to do reductions; that is, we can't just parameterize a complexity class by a single distribution. This is because a problem can have more than one natural distribution; it's not always obvious what to take as the 'uniform distribution.'

15.3 Polynomial-Time Samplability

Say μ is *polynomial-time samplable* if there exists a probabilistic algorithm A, taking no input, that outputs x with probability $\mu\prime(x)$ and runs in $\text{poly}(|x|)$ time. Any polynomial-time computable distribution is also polynomial-time samplable, provided that for all x,

$$\mu\prime(x) \geq 2^{-\text{poly}(|x|)} \text{ or } \mu\prime(x) = 0. \tag{15.5}$$

To see this: have A first choose a real number r uniformly at random from $[0,1]$, to $\text{poly}(|x|)$ bits of precision, then use binary search to find the first x such that $\mu(x) \geq r$. However,

μ efficiently samplable does *not* imply μ efficiently computable, under weaker assumptions than the existence of pseudorandom generators. So in addition to **DistNP**, we can look at the class

$$\langle \mathbf{NP}, \mathbf{P}\text{-samplable} \rangle = \{ \langle L, \mu \rangle : L \in \mathbf{NP}, \mu \text{ polynomial-time samplable} \} . \tag{15.6}$$

15.4 Reductions

We say that $\langle L_1, \mu_1 \rangle \leq \langle L_2, \mu_2 \rangle$ if there exists a polynomial-time computable function f such that:

1. $x \in L_1$ iff $f(x) \in L_2$.
2. For all $y \in \{0,1\}^*$,

$$\sum_{x : f(x) = y} \mu\prime_1(x) \leq \text{poly}(|y|)\, \mu\prime_2(y) . \tag{15.7}$$

The second condition is called *domination*.

15.5 Polynomial-Time on Average

Given a problem $\langle L, \mu \rangle$ and an algorithm A that runs in time $t(x)$ on input x, what does it mean to say that A solves $\langle L, \mu \rangle$ in polynomial time on average? We will consider three bad definitions before settling on the best one.

Try 1 EX $[t(x)]$ should be small. But this expectation can be infinite, even for algorithms that run in worst-case polynomial time.

Try 2 For each input size n, EX $[t(x) \mid |x| = n]$ should be $O(n^c)$ for some constant c. But having defined distributional problems and reducibility without separating problems by input size, it seems inelegant to separate them now.

Try 3 EX $\left[\frac{t(x)}{|x|^c}\right]$ should be $O(1)$ for some constant c. But this definition is not *robust*, meaning that (1) reductions aren't transitive and (2) the definition is sensitive to trivial changes such as replacing a matrix representation of a graph by an adjacency list. To see this, let μ be the uniform distribution, and let

$$t(x) = 2^n \text{ if } x = \overrightarrow{0},\ t(x) = n^2 \text{ otherwise.} \tag{15.8}$$

The average running time is about n^2. But now consider squaring the running times:

$$t(x) = 2^{2n} \text{ if } x = \overrightarrow{0},\ t(x) = n^4 \text{ otherwise.} \tag{15.9}$$

Now the average running time is about 2^n.

Try 4 $\mathrm{EX}\left[\frac{t(x)^{1/c}}{|x|}\right]$ should be $O(1)$ for some constant c. This definition seems to work; it is achieved in particular by any algorithm that runs in worst-case polynomial time. It is the one we'll use[1].

Suppose that

$$\mathrm{EX}\left[\frac{t(x)^{1/c}}{|x|}\right] = O(1). \tag{15.10}$$

Then what is

$$\Pr\left[t(x) \geq k\,|x|^{c}\right] \quad ? \tag{15.11}$$

Clearly it's the same as

$$\Pr\left[t(x)^{1/c} \geq k^{1/c}\,|x|\right]. \tag{15.12}$$

By Markov's inequality, this probability is $O\left(k^{-1/c}\right)$. In other words, the algorithm runs in time at most kn^{c} for a subset of inputs having probability $1-k^{-1/c}$. Thus we see that our definition gives a quantitative tradeoff for how much time we need to solve an increasing fraction of inputs.

15.6 Some Results

A difficult result of Levin states that if $\langle L_1, \mu_1\rangle \leq \langle L_2, \mu_2\rangle$ and $\langle L_2, \mu_2\rangle$ is polynomial-time on average (under the right definition), then $\langle L_1, \mu_1\rangle$ is also polynomial-time on average.

Another result, due to Impagliazzo and Levin, states that if $\langle L, \mu\rangle$ is **DistNP**-complete, then $\langle L, \mu\rangle$ is also complete for the class $\langle \mathbf{NP}, \mathbf{P}\text{-samplable}\rangle$. This result is useful because, for natural problems, it's often easier to prove **DistNP**-completeness than to prove $\langle \mathbf{NP}, \mathbf{P}\text{-samplable}\rangle$-completeness.

15.7 Existence of Complete Problems

We now show that there exists a problem (albeit an artificial one) complete for **DistNP**. Let the inputs have the form $\left\langle M, x, 1^{t}, 1^{l}\right\rangle$, where M is an encoding of a Turing machine and 1^{t} is a sequence of t ones.

* Decide whether there exists a string y such that $|y| \leq l$ and $M(x,y)$ accepts in at most t steps.

[1] Indeed we might even go a step farther, and say that A is average-case polynomial if there exists an input set S, having probability $1-o(n^{c})$, such that A runs in polynomial time on S. But we probably can't pull off reductions with this last definition.

That (*) is **NP**-complete follows directly from the definition. Recall the criterion for $L \in \mathbf{NP}$: there exists a machine M running in $t = \text{poly}(|x|)$ steps such that $x \in L$ iff there exists a y with $y = \text{poly}(|x|)$ such that $M(x, y)$ accepts. Thus, to reduce L to (*) we need only map x onto $R(x) = \langle M, x, 1^t, 1^l \rangle$.

But in the average-case setting things are harder. Let $\langle L, \mu \rangle \in \mathbf{DistNP}$. Define a uniform distribution over the $\langle M, x, 1^t, 1^l \rangle$ as follows:

$$\Pr\left[\left\langle M, x, 1^t, 1^l \right\rangle\right] = \frac{1}{|M|\,(|M|+1)\,2^{|M|}} \cdot \frac{1}{|x|\,(|x|+1)\,2^{|x|}} \cdot \frac{1}{(t+l)\,(t+l+1)}. \tag{15.13}$$

The trouble is that, because of the domination condition, we can't map x onto $R(x)$ if $\mu\prime(x) > \text{poly}(|x|)\,2^{-|x|}$. So what we'll do is *compress* x to a shorter string if $\mu\prime(x)$ is large. Intuitively, by mapping high-probability strings onto shorter lengths, we make their high probability less conspicuous. The following lemma shows how to do this.

Lemma 52
Suppose μ is a polynomial-time computable distribution over x. Then there exists a polynomial-time algorithm C such that

1. *C is injective: $C(x) \neq C(y)$ iff $x \neq y$.*
2. *$|C(x)| \leq 1 + \min\left\{|x|, \log \frac{1}{\mu\prime(x)}\right\}$.*

Proof: Use Huffman encoding. If $\mu\prime(x) \leq 2^{-|x|}$ then simply let $C(x) = 0x$, that is, 0 concatenated with x. If, on the other hand, $\mu\prime(x) > 2^{-|x|}$, then let $C(x) = 1z$. Here z is the longest common prefix of $\mu(x)$ and $\mu(x-1)$ when both are written out in binary. Since μ is computable in polynomial time, so is z. C is injective because only two binary strings s_1 and s_2 can have the longest common prefix z; a third string s_3 sharing z as a prefix must have a longer prefix with either s_1 or s_2. Finally, since $\mu\prime(x) \leq 2^{-|z|}$, $|C(x)| \leq 1 + \log \frac{1}{\mu\prime(x)}$.
□

Now the reduction is to map x onto $R_2(x) = \left\langle \overline{M}, C(x), 1^{\overline{t}}, 1^{l+|x|} \right\rangle$. Here $\overline{M}$ is a machine that on input z, x, y checks that $C(x) = z$ (i.e., that x is a valid decoding of z) and that $M(x, y)$ accepts. The running time of $\overline{M}$ is $\overline{t}$. Clearly $x \in L$ iff $\overline{M}$ accepts. To show that domination holds, let $\mu_2\prime(x) = \Pr[R_2(x)]$. Then, since the map is one-to-one, we need only show that $\mu\prime(x) \leq \text{poly}(|x|)\,\mu\prime_2(x)$. Since $\overline{t} = O(\text{poly}(t))$,

$$\begin{aligned}
\mu\prime_2(x) &= \frac{1}{O\left(|\overline{M}|^2\right) 2^{|\overline{M}|}} \cdot \frac{1}{O\left(|C(x)|^2\right) 2^{|C(x)|}} \cdot \frac{1}{O(t+l+|x|)^2} \\
&\geq \text{poly}(|x|) \max\left(2^{-|x|}, \mu\prime(x)\right) \\
&\geq \text{poly}(|x|)\,\mu\prime(x)
\end{aligned}$$

and we're done.

Note that, since we mapped longer inputs to shorter ones, we could not have done this reduction input-length-wise.

Chapter 16

Average-Case Complexity of the Permanent

March 23, 2001, Scribe: John Leen

Today we will connect average-case complexity with worst-case complexity by introducing a #**P**-complete problem, PERMANENT, and showing that if PERMANENT is polynomial in the average case then it is polynomial in the worst case.

16.1 The PERMANENT Problem

The *permanent* is a number assigned to a matrix $M \in R$, for a ring R, according to the formula

$$\text{perm}(M) = \sum_{\pi \in S_n} \prod_{i=1}^{n} M_{i,\pi(i)}$$

where S_n is the group of permutations on n items. The definition of the permanent closely resembles that of the determinant; the only difference is that the determinant has a coefficient $(-1)^{\text{sgn}(\pi)}$ in front of the product. However, this resemblance is deceptive: the determinant can be calculated in polynomial time, whereas PERMANENT is #**P**-complete.

CLAIM 53
Let $R = \mathbb{Z}$ and let M be a 0/1 matrix, which we think of as representing a bipartite graph $G = (U, V, E)$ with $|U| = |V| = n$ as follows: number both edge sets U and V, separately, by $1, 2, \ldots n$, and let $M_{i,j}$ be 1 if edge $(i, j) \in E$, and 0 otherwise. Then $\text{perm}(M)$ is the number of perfect matchings in G.

PROOF: Each $\pi \in S_n$ is a potential perfect matching, in that it can be thought of as a bijection between U and V. This corresponds to an actual perfect matching iff all edges $(i, \pi(i))$ are in E, i.e. if $M_{i,\pi(i)} = 1$ for all i, which is equivalent to saying that $\prod_{i=1}^{n} M_{i,\pi(i)} = 1$. Thus we get a summand of 1 in the formula for $\text{perm}(M)$ for each perfect matching of G. $\square$

Suppose we allow the entries of M to be nonnegative integers. Then M represents a bipartite graph where the edges have multiplicity given by the entries of M. It still makes sense to interpret $\text{perm}(M)$ as the number of permanent matchings, because the product of the multiplicities of the edges of a matching is the number of ways of realizing that matching.

Note that in $\mathbb{Z}_2$, $\text{perm}(M) = \det(M)$ and thus is polynomial time. But for sufficiently large moduli (or no modulus at all) it's #**P**-complete.

THEOREM 54
Suppose for fixed n and $\mathcal{F}$, $|\mathcal{F}| > n^{O(1)}$, we can compute $\text{perm}(M)$ *on a fraction $1 - \frac{1}{2n}$ of the inputs M in polynomial time. Then there is a (worst-case) polynomial time probabilistic algorithm for computing* $\text{perm}(M)$.

Rather than proving this theorem directly, we will just observe that $\text{perm}(M)$ is a degree n polynomial in indeterminates $M_{11}, M_{12}, \ldots, M_{1n}, \ldots, M_{nn}$, and prove the following much more general result.

THEOREM 55
Suppose we have oracle access to a function $f : \mathcal{F}^n \to \mathcal{F}$ and f agreees with some degree d polynomial $p : \mathcal{F}^n \to \mathcal{F}$ on a fraction $1 - \frac{1}{2d}$ of its inputs where $|\mathcal{F}| \geq poly(n, d)$. Then we can compute $f(x)$ with high probability for all $x \in \mathcal{F}^n$ in $poly(n, d)$ time, assuming field operations can be performed in $poly(n, d)$ time.

PROOF: Suppose that we input $x \in \mathcal{F}^n$ and pick $y \in \mathcal{F}^n$ uniformly at random. Consider the line $l(t) = x + ty$. Each point of l is uniformly distributed except $x = l(0)$. Now,

$$\begin{aligned}
\Pr[p(l(i)) = f(l(i)) \text{ for } i = 1, 2, \ldots, d+1] &= 1 - Pr[\exists i \in 1, \ldots, d+1 : p(l(i)) \neq f(l(i))] \\
&\geq 1 - \frac{d+1}{2(d+1)} = \frac{1}{2}
\end{aligned}$$

Now $q(t) = p(l(t))$ is a degree d polynomial in t, and we know $d + 1$ of its values, so we can interpolate the others. Thus we have the following algorithm:

- input $x \in \mathcal{F}^n$
- pick random $y \in \mathcal{F}^n$
- consider $l(t) = x + ty$
- compute $f(l(1)), \ldots, f(l(d+1))$
- compute $q(t)$ of degree d such that $q(i) = f(l(i))$ for $i = 1, \ldots, d+1$
- output $q(0)$

This runs in $\text{poly}(n, d)$ time, and with probability $\geq \frac{1}{2}$, it outputs $f(x)$. □

We can strengthen our result to hold even if $p(x)$ only agrees with $f(x)$ on $\frac{3}{4} + \epsilon$ of its inputs. This stronger proof depends on the following theorem, due to Berlekamp and Welch.

THEOREM 56
Let $q : \mathcal{F} \to \mathcal{F}$ be a polynomial of degree d. Say we are given n pairs $(a_1, b_1), \ldots, (a_n, b_n)$ such that for fewer than $\frac{n-d}{2}$ pairs (a_i, b_i), $b_i \neq p(a_i)$. Then q can be reconstructed with $poly(n, d)$ field operations.

Intuitively, you should think of q as a message which we encode as the values $q(a_1), \ldots, q(a_n)$ as an encoding which uniquely specifices q if $n > d$. Say we receive $b_1, \ldots, b_n$. If we assume that fewer than $\frac{n-d}{2}$ of the transmitted values $q(a_i)$ have been been corrupted, then we can recover our original message q by attempting to extrapolate it from different sets of $d+1$ received values; a majority of them will give the correct q.

Now, as promised, we apply Berlekamp-Welch to strengthen our main theorem:

Theorem 57
Suppose we have oracle access to a function $f : \mathcal{F}^n \to \mathcal{F}$ and f agreees with some degree d polynomial $p : \mathcal{F}^n \to \mathcal{F}$ on a fraction $\frac{3}{4} + \epsilon$ of its inputs where $|\mathcal{F}| \geq poly(n, d)$. Then we can compute $f(x)$ with high probability for all $x \in \mathcal{F}^n$ in $poly(n, d)$ time, assuming field operations can be performed in $poly(n, d)$ time.

Proof: Consider the following algorithm:

- input $x \in \mathcal{F}^n$
- pick random $y \in \mathcal{F}^n$
- consider $l(t) = x + ty$
- compute $f(l(1)), \ldots, f(l(d+1))$
- use Berlekamp-Welch to find q such that $q(t) = f(l(t))$ on more than $|\mathcal{F}| - \frac{|\mathcal{F}|-d}{2}$ values of t
- output $q(0)$

If $q(t) = p(l(t))$ and $f(l(t))$ agree on $> |\mathcal{F}| - 1 - \frac{|\mathcal{F}|-d-1}{2}$ values of t, Berlekamp-Welch will find $q(t) \equiv p(l(t))$ and the algorithm succeeds (and in polynomial time). So, assuming only $\frac{1}{4} - \epsilon$ of our points disagree, we need to show that only half of our runs fail.

$$\begin{aligned}
\mathbf{E}\left[\#\text{ values of } t \in \mathcal{F} - \{0\} \text{ with } p(l(t)) \neq f(l(t))\right] &= \sum_{t \neq 0} \mathbf{E}\left[\begin{cases} 0 & \text{if } p(l(t)) = f(l(t)) \\ 1 & \textit{otherwise} \end{cases}\right] \\
&\leq (|\mathcal{F}| - 1)\left(\frac{1}{4} - \epsilon\right)
\end{aligned}$$

So Markov tells us that

$$\Pr\left[\#t : p(l(t)) \neq f(l(t)) > \frac{n-d}{2}\right] \leq \frac{n(\frac{1}{4} - \epsilon)}{\frac{n-d}{2}} \leq \frac{1}{2} \quad \text{if } n \geq \tfrac{d}{2\epsilon}$$

Thus, our algorithm succeeds more than half the time, so it can be repeated to yield any degree of accuracy we wish. □

There is an even stronger result in which $\frac{3}{4} + \epsilon$ is replaced with $\frac{1}{2} + \epsilon$, but we won't prove it here.

Chapter 17

Interactive Proofs

April 2, 2001, Scribe: Ranjit Jhala

In this lecture we will look at the notion of *Interactive Proofs* and the induced class **IP**. Intuitively, this class is obtained by adding the ingredients Interaction and Randomness to the class **NP**. As we shall see, adding either one does not make much difference but the two put together make **IP** a very powerful class.

17.1 Interactive Proofs

We begin be recalling the definition of **NP**. Given a language L, we say that: $L \in \mathbf{NP}$ iff there is an algorithm $V(\cdot,\cdot)$ running in polynomial time $T(\cdot)$ and a polynomial $p(\cdot)$ such that

$$x \in L \Leftrightarrow \exists y, |y| \leq p(|x|) \text{ and } V(x,y) \text{ accepts}$$

We can rewrite the above as two separate conditions, using elementary ideas from logic.

$$\begin{aligned} x \in L &\Rightarrow \exists y, |y| \leq p(|x|) \text{ and } V(x,y) \text{ accepts} \qquad \text{(Completeness)} \\ x \notin L &\Rightarrow \forall y, |y| \leq p(|x|) \text{ and } V(x,y) \text{ rejects} \qquad \text{(Soundness)} \end{aligned}$$

We may generalise this to the figure below 17.1 where the process of recognizing if $x \in L$ is carried out by a *Verifier* V that must run in polynomial time, and a *Prover* P, who has unbounded time and space. The task of the prover is to convince the verifier that $x \in L$ and the verifier is a skeptic who's task is to be convinced if and only if the string actually belongs in the language.

Now the class **NP** is the set of languages L such that $L \in \mathbf{NP}$ iff there is a Prover P and a polynomial time verifier V such that:

$$\begin{aligned} x \in L &\Rightarrow P \text{ has a strategy to convince V} \qquad \text{(Completeness)} \\ x \notin L &\Rightarrow P \text{ has no strategy to convince } V \qquad \text{(Soundness)} \end{aligned}$$

By *strategy* we mean in this case the certificate or proof that is polynomially small, (corresponding to y in the figure 17.1) that P supplies to V, while later we will generalise

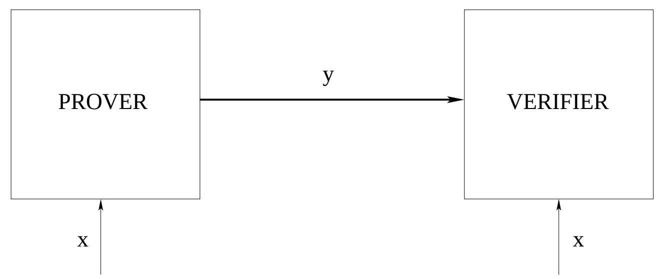

Figure 17.1: **NP** with a Prover-Verifier mechanism

it to interaction *i.e.* where a sequence of messages is exchanged between the two and a strategy for P is a function from the sequence of messages seen to the next message that P must send to V.

Thus, we generalise the above definition to get the class **IP**. The class **IP** differs in the following two ways:

1. There is randomness, the verifier may be a randomized machine.
2. There is interaction: unlike the above where a single static proof is sent across, there are rounds of communication, where the verifier may "ask" further questions of the prover based on the messages that have been sent to it.

First, we shall see that both of the above are required.

17.1.1 NP + Interaction = NP

Let the class $\overline{\mathbf{NP}}$ be the class corresponding to **NP** + interaction. That is we allow the verifier to ask questions and thus the strategy is a function from sequences of messages (the past communication) to messages (the next message that the verifier must send). We say that the class $\overline{\mathbf{NP}}$ is the set of languages L such that $L \in \overline{\mathbf{NP}}$ iff there is a Prover P and a polynomial time verifier V such that:

$$x \in L \Rightarrow P \text{ has a strategy to convince } V$$
$$x \notin L \Rightarrow P \text{ has no strategy to convince } V$$

It is easy to check that $\mathbf{NP} = \overline{\mathbf{NP}}$. We only need to show $\overline{\mathbf{NP}} \subseteq \mathbf{NP}$, as the other inclusion is obvious. Say $L \in \overline{\mathbf{NP}}$, let P, V be the prover and verifier for L. Define P', V' the one-round prover and verifier as follows: P' sends to V' the entire transcript of the interaction between P, V as the certificate. Suppose that interaction was the set of messages $y_1, \ldots, y_n$ as in the figure 17.2 below. V' is essentially like V only it has been given all the answers to its questions at once. Thus it takes the first message of P, y_1, its response (behaving as V) is exactly y_2, but y_3 is the reply to that, and so on, it "asks" the questions and is supplied all the answers on the same transcript, and it accepts iff after this exchange V accepted. Thus the language accepted by P', V' is the same as L and so $L \in \mathbf{NP}$. Note that the entire set of messages is polynomially long as there are polynomially many messages and each is polynomially long.

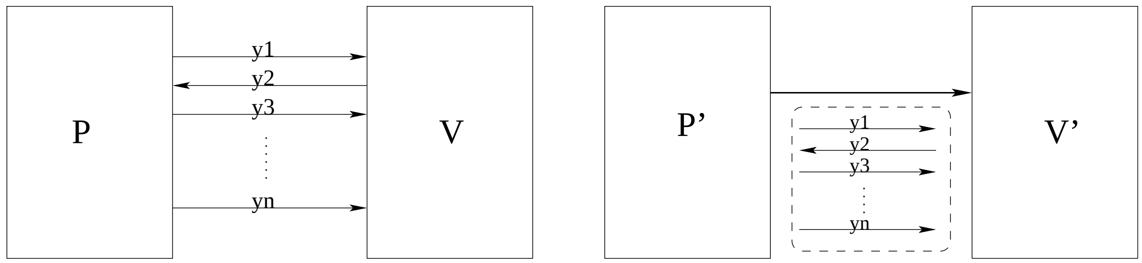

Figure 17.2: Simulating several rounds of interaction in one round

Note that for the above to work, the prover must at the very beginning know what responses V may make, which is possible only if V is deterministic. If V was randomized then to cover all possible questions in a polynomially long interaction P' would have to send an exponentially large transcript.

17.1.2 NP + Randomness

By adding randomness to the class **NP** we get the class **MA**.

DEFINITION 26 (**MA**) *$L \in$ **MA** iff there exists a probabilistic polynomial time machine V such that:*

$$x \in L \Rightarrow \exists y.\mathbf{Pr}[V(x, y) \text{ accepts }] \geq \frac{2}{3}$$
$$x \in L \Rightarrow \forall y.\mathbf{Pr}[V(x, y) \text{ accepts }] \leq \frac{1}{3}$$

As in similar classes, the success probabilities may be boosted arbitrarily high. It is conjectured that **MA** = **NP**. What is known is that if **coNP** $\subseteq$ **MA** then the polynomial heirarchy collapses.

EXERCISE 6 A non-deterministic circuit C has two inputs $x = x_1, \ldots, l_m$ and $y = y_1, \ldots, y_n$. C accepts the string x iff $\exists y.C(x, y) = 1$. Show that **MA** has non-deterministic circuits of polynomial size. (*Hint:* Proof is similar to that of **BPP** has poly sized circuits)

We are now in a position to define formally the class **IP**.

17.2 IP

DEFINITION 27 (**IP**) *A language L is in $\mathbf{IP}(r(\cdot))$ iff there exists a probabilistic polynomial time verifier V such that:*

$$x \in L \Rightarrow \exists P.\mathbf{Pr}[V \text{ interacting with } P \text{ accepts }] \geq \frac{2}{3}$$
$$x \in L \Rightarrow \forall P.\mathbf{Pr}[V \text{ interacting with } P \text{ accepts }] \leq \frac{1}{3}$$

Where V also uses at most $r(|x|)$ rounds of interaction.

Remark 1 By round we really mean a single "message" and not a question-answer sequence. That is, by two rounds we mean that the Verifier asks a question of the Prover and the Prover replies, which is exactly two messages.

A related class is **AM** which is defined as follows:

Definition 28 (**AM**) *A language L is in $\mathbf{AM}(r(\cdot))$ iff $L \in \mathbf{IP}(r(\cdot))$ and at each round the verifier sends a random message, that is a message that is completely random and independent of all previous communication.*

The above is also known as *Public Coin Proof Systems*.

There are the following surprising theorems about the classes **AM** and **IP**.

Theorem 58
$\mathbf{IP}(r(n)) \subseteq \mathbf{AM}(r(n)+2)$

Theorem 59
$\forall r(n) \geq 2$, $\mathbf{AM}(2r(n)) \subseteq \mathbf{AM}(r(n))$.

This yields the following corollaries:

Corollary 60
$\mathbf{AM}(O(1)) \subseteq \mathbf{AM}(2)$

That is, all constant round **AM** proofs can be done in just two rounds ! And also:

Corollary 61
IP *(O(1))*=**AM** *(O(1))*=**AM** *(2)*

Finally we have the famous theorem of Shamir:

Theorem 62
$\mathbf{IP}(poly(n)) = \mathbf{PSPACE}$

We know that $\mathbf{AM}(2)$ is good enough for systems with a constant number of rounds, but it would be surprising indeed if polynomially many rounds could be simulated in a constant number of round. Due to the above result, when people say **AM** they mean $\mathbf{AM}(2)$, but when people say **IP** they mean $\mathbf{IP}(\text{poly})$.

The above classes **IP** and **AM** were discovered independently. The first was discovered by Shafi Goldwasser, Silvio Micali and Charles Rackoff. They proposed the class **IP** in order to facilitate the definition of the class Zero Knowledge Proofs which we shall see shortly, the paper was written in 1983 but appeared first in FOCS 85. The class **AM** was discovered by Lazlo Babai (the journal version of the paper is by Babai and Moran), in order to characterize the complexity of a group-theoretic problem. The paper also appeared in FOCS 85, and contained theorem 59.

There is also the following theorem that gives an upper bound on **IP** with constant rounds.

Theorem 63
If $\mathbf{coNP} \subseteq \mathbf{IP}(O(1))$ *then the polynomial heirarchy collapses.*

Thus for a long time it was believed that **IP** could not capture **coNP** as it was thought that **IP** with a constant number of rounds was roughly as good as **IP** with polynomially many rounds. However, it was shown that in fact **coNP** was contained in **IP**(poly(n)), the proof of which we shall shortly see and a few weeks after that result, Shamir proved that in fact **IP** was the same as **PSPACE**.

17.3 An Example: Graph Non-Isomorphism

To get a feel of the power of interaction and randomness let us look at an example, Graph Non-Isomorphism. The problem is the following:

- Input: Graphs $G_1 = (V_1, E_1),\ G_2 = (V_2, E_2)$
- Output: Decide whether there exists *NO* $\pi : V_1 \mapsto V_2$ such that $(u, v) \in E_1$ iff $(\pi(u), \pi(v)) \in E_2$.

This problem is not known to be in **NP**. The difficulty lies in constructing a succint proof that no permutation of the vertices is good, given that there are exponentially many permutations, let us see how this is in **AM** (2).

Suppose that the skeptic Arthur (the verifier) has the two graphs G_1, G_2 and wishes to know whether or not all the permutations are bad. His only resources are randomness and his wizard Merlin who is all knowing (the prover).

He does the following:

1. Picks a random $b \in \{1, 2\}$.
2. Creates G to be a random permutation of G_b.
3. Sends G to Merlin and demands to know of Merlin which graph he sent him a permutation of.
4. If Merlin was correct Arthur believes that the graphs are non-isomorphic (accepts), otherwise Merlin was incorrect, Arthur believes the graphs are isomorphic (rejects).

Notice that if the graphs are really non-isomorphic, then Merlin will always be able to tell which graph was sent to him as only one of the graphs will be isomorphic to the random permutation that Merlin is given, and with his infinite resources Merlin will figure out which one and thus will reply correctly and so Arthur will accept with probability 1.

If the graphs are isomorphic then Merlin cannot tell as the permutation is a permutation of both $G_1,\ G_2$. Merlin's output depends only on G_1, G_2, G, say it is i. Now $i = b$ with probability $\frac{1}{2}$ as b is picked independantly at the start. Thus in the case the graphs are isomorphic, Arthur accepts only with probability $\frac{1}{2}$. By repeating this k times we can shrink this to 2^{-k}.

Thus we have a protocol, a prover and a verifier for Graph Non-Isomorphism and so it is in **AM** (2).

Note: The verifier of the above has learnt nothing at all about the Graphs that he did not know before. He does not have a proof or certificate with which he may convince a third party that the Graphs are non-isomorphic. The entire power was in the interaction between it and the prover, and the one bit that the prover gave it gives it significant confidence but no *knowledge*. This was therefore a *Zero Knowledge* Proof.

REMARK 2 GRAPH NON-ISOMORPHISM $\in$ **IP**(2) = **AM**(2). Thus, GRAPH ISOMORPHISM is not **NP**-complete unless the polynomial heirarchy collapses. Thus is because, if GRAPH ISOMORPHISM is **NP**-complete then GRAPH NON-ISOMORPHISM is **coNP**-complete, and from the above result, **coNP** $\subseteq$ **AM**(2) and from the theorem 63, the polynomial heirarchy would collapse.

In the next couple of lectures we will see how interactive proofs can be given for problems in **coNP**.

THEOREM 64
coNP $\subseteq$ **IP**($poly(n)$)

We shall see that the problem of checking if a 3-CNF formula is not satisfiable, which is a **coNP**-complete problem, is in **IP** and thus conclude that **coNP** is contained in **IP**. This proof may be extended to a proof that the problem of deciding the language of QUANTIFIED BOOLEAN FORMULAE, which is **PSPACE**-complete is in **IP**, and thus shows that **PSPACE** = **IP**.

Chapter 18

IP=PSPACE

April 4, 2001, Scribe: Dror Weitz

In this and the next lecture we will see that **IP** = **PSPACE**. In this lecture we give an interactive proof system for UNSAT, and therefore show that **coNP** $\subseteq$ **IP**. In fact, as we will see, the same proof system with minor modifications can be used for #SAT. Since #SAT is #**P**-complete, and since **PH** $\subseteq$ $\mathbf{P}^{\#\mathbf{P}}$, then **PH** $\subseteq$ **IP**. In the next lecture we will generalize this protocol in order to solve QSAT, and thus show that **PSPACE** $\subseteq$ **IP**.

18.1 UNSAT $\subseteq$ IP

Let ϕ be a formula with m clauses over the variables $x_1, \ldots, x_n$. Let $N > 2^n \cdot 3^m$ be a prime number. We translate ϕ to a polynomial p over the field $\pmod N$ in the following way. Literals are translated as follows: $x_i \to x_i$, $\overline{x_i} \to (1 - x_i)$. A clause is translated to the sum of the (at most 3) expressions corresponding to the literals of the clause. Finally, p is taken as the product of the m expressions corresponding to the m clauses. Notice that each literal is of degree 1, and therefore p is of degree at most m. Furthermore, for a 0-1 assignment, p evaluates to 0 if this assignment does not satisfy ϕ, and to a non-zero number if this assignment does satisfy ϕ where this number can be at most 3^m. Hence, ϕ is unsatisfiable iff

$$\sum_{x_1 \in \{0,1\}} \sum_{x_2 \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} p(x_1, \ldots, x_n) \equiv 0 \pmod N$$

Also, the translation to p can be done in polynomial time, and if we take $N = 2^{O(n+m)}$ then elements in the field can be represented by $O(n+m)$ bits, and thus one evaluation of p in any point in $\{0, \ldots, N-1\}^m$ can be computed in polynomial time.

We now describe a protocol that enables the prover to convince the verifier that ϕ is unsatisfiable (see also Figure 18.1 for a sketch of the protocol). At the first round, the prover sends N, a proof that it is prime, and an m-degree univariate polynomial $q_1(x) = \sum_{x_2, \ldots, x_n \in \{0,1\}} p(x, x_2, \ldots, x_n)$. At this point, the verifier verifies the primality proof as well as checking that $q_1(0) + q_1(1) = 0$. If any of the checks fail then the verifier rejects. Otherwise, it continues by choosing a random $r_1 \in \{0, \ldots, N-1\}$ and sending it to the prover. The prover responds by sending an m-degree univariate polynomial

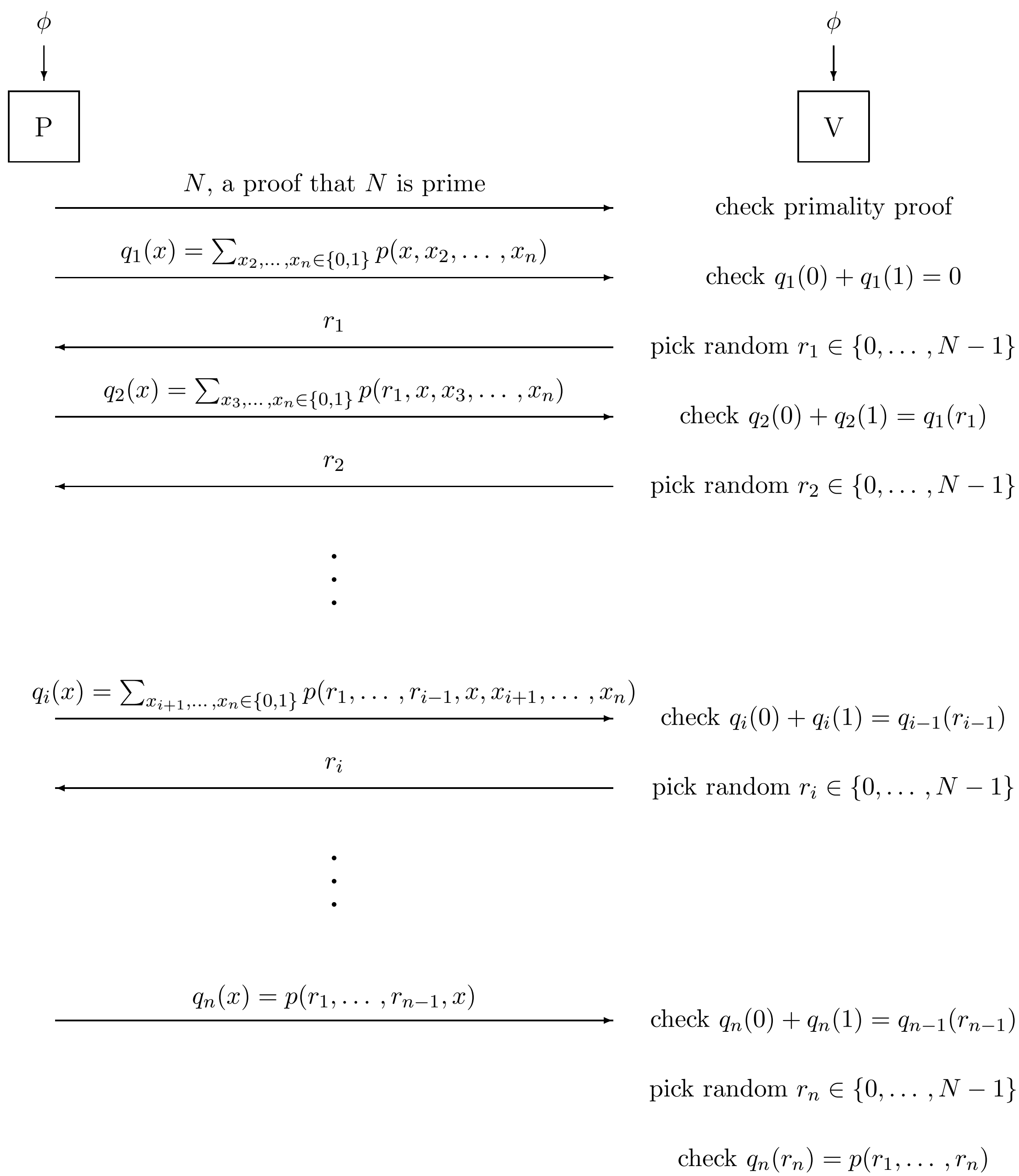

Figure 18.1: The protocol for proving unsatisfiability

$q_2(x) = \sum_{x_3,\dots,x_n \in \{0,1\}} p(r_1, x, x_3, \dots, x_n)$. Now, the verifier checks if $q_2(0)+q_2(1) = q_1(r_1)$. If not, it rejects, and otherwise it continues by choosing a random $r_2 \in \{0, \dots, N-1\}$ and sending it to the prover. The protocol continues in this way where the ith message that the prover sends is a polynomial $q_i(x) = \sum_{x_{i+1},\dots,x_n \in \{0,1\}} p(r_1, \dots, r_{i-1}, x, x_{i+1}, \dots, x_n)$. The verifier responds by checking that $q_i(0) + q_i(1) = q_{i-1}(r_{i-1})$, rejecting if it is not, and otherwise, choosing a random $r_i \in \{0, \dots, N-1\}$ and sending it to the prover. At the last round, the prover sends $q_n(x) = p(r_1, \dots, r_{n-1}, x)$. The verifier checks that $q_n(0) + q_n(1) = q_{n-1}(r_{n-1})$, chooses a random $r_n \in \{0, \dots, N-1\}$, and checks that $q_n(r_n) = p(r_1, \dots, r_n)$. The verifier rejects if either of the checks fail, and accepts otherwise. The verifier can indeed perform the last check since as mentioned before, it can translate ϕ to p and evaluate it at the point $(r_1, \dots, r_n)$ in polynomial time. Also, the messages are of polynomial length because elements of the field can be written with $O(m+n)$ bits, and since each polynomial has m coefficients.

We now show the correctness of this proof system. If ϕ is unsatisfiable then the prover can make the verifier accept w.p. 1 by just following the protocol since $q_1(0) + q_1(1) = \sum_{x_1,\dots,x_n \in \{0,1\}} p(x_1, \dots, x_n) \equiv 0 \pmod{N}$. It is clear that the rest of the checks will succeed if the prover send the q_i's accordingly. We have to show that if ϕ is satisfiable then no matter what polynomials the prover sends, the verifier will reject with high probability. We write $p_i(x)$ for the polynomial that a prover who follows the protocol sends in round i. Since ϕ is satisfiable then $\sum_{x_1,\dots,x_n \in \{0,1\}} p(x_1, \dots, x_n) \not\equiv 0 \pmod{N}$. Therefore, $p_1(0) + p_1(1) \neq 0$, and hence, if the prover sends $q_1 = p_1$ then the verifier will reject. If the prover sends $q_1 \neq p_1$ then since both are degree-m polynomials then they agree on at most m places. Thus, there is probability $\geq 1 - \frac{m}{N}$ that $p_1(r_1) \neq q_1(r_1)$. Suppose that indeed $p_1(r_1) \neq q_1(r_1)$. If the prover then sends $q_2 = p_2$ then the verifier will reject because $q_2(0)+q_2(1) = p_1(r_1) \neq q_1(r_1)$. Thus, the prover must send $q_2 \neq p_2$. Again, in that case, q_2 and p_2 will differ on a fraction $\geq 1 - \frac{m}{N}$ of the elements of $\{0, \dots, N-1\}$, so $p_2(r_2) \neq q_2(r_2)$ with probability $\geq 1 - \frac{m}{N}$. We continue the argument in a similar way. If $q_i \neq p_i$ then w.p. $\geq 1 - \frac{m}{N}$, r_i is such that $p_i(r_i) \neq q_i(r_i)$. If so, then the prover must send $q_{i+1} \neq p_{i+1}$ in order for the verifier not to reject. At the end, if the verifier has not rejected before the last check, then w.p. $\geq 1 - (n-1)\frac{m}{N}$, $q_n \neq p_n$. If so, then w.p. $\geq 1 - \frac{m}{N}$ the verifier will reject since, again, $q_n(x)$ and $p(r_1, \dots, r_{n-1}, x)$ differ on at least that fraction of the points. Thus, the total probability that the verifier will accept is at most $\frac{nm}{N}$.

18.2 A Proof System for #SAT

We now show that with a few minor modifications, the previous proof system can be used to prove that a formula has a given number of satisfying assignments. Suppose ϕ has k satisfying assignments. We wish to give a protocol s.t. if the prover gives k as an answer, he is able to continue with the protocol s.t. the verifier will accept w.p. 1, but if it gives another number as an answer, then no matter what messages the prover sends afterwards, the verifier will reject with high probability.

We first change the way we translate ϕ to the polynomial p. The change is only in the way we translate clauses. Instead of translating a clause to the sum of the literals, we translate the clause $(z_1 \vee z_2 \vee z_3)$ to $1-(1-z_1)(1-z_2)(1-z_3)$. Notice that if a 0-1 assignment to the variables satisfies the clause, then the value of this expression will be 1, and if the assignment does not satisfy the clause, this value will be 0. Hence, 0-1 assignments that satisfy ϕ will be evaluated to 1 by p, while as before, the 0-1 assignments that do not satisfy ϕ will be evaluated to 0. Notice that now the degree of p is $3m$ instead of m, but now the sum over all evaluations of 0-1 assignments is equal to the number of satisfying assignments of ϕ. For the same reason, it is now enough to take $N > 2^n$.

In the current protocol, the prover starts by sending k, and then continues as in the previous protocol (with the updated definition of p). After receiving the first message, the verifier checks if $q_1(0) + q_1(1) = k$ instead of checking $q_1(0) + q_1(1) = 0$, and follows the rest of the previous protocol. If k is indeed the correct number of satisfying assignments, then the prover can continue by sending $q_i = p_i$ at each round, and the verifier will accept with probability 1. If k is not the correct number of satisfying assignments, then according to the same analysis as in the previous section, the verifier will accept with probability at most $\frac{3mn}{N}$.

Chapter 19

IP=PSPACE

April 11, 2001, Scribe: Beini Zhou

In this lecture, we will prove that **PSPACE** $\subseteq$ **IP**. We will essentially use the same algebraic ideas from the last lecture where we proved $\sharp$**P** $\subseteq$ **IP**.

19.1 PSPACE-Complete Language: TQBF

For a 3-CNF boolean formula $\phi(x_1, x_2, \dots, x_n)$, we may think of its satisfiability problem as determining the truth value of the statement

$$\exists x_1 \exists x_2, \dots, \exists x_n \quad \phi(x_1, x_2, \dots, x_n).$$

We can generalize this idea to allow universal quantifiers in additional to existential quantifiers to get formulas denoted as quantified boolean formulas. For example, $\forall x_1 \exists x_2 \quad (x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2})$ is a quantified boolean formula. In fact, it's easy to see that it's a true quantified boolean formula. Now consider the language of all true quantified boolean formulas:

$$\text{TQBF} = \{\Phi : \quad \Phi \text{ is a true quantified boolean formula}\}.$$

It is known that TQBF is **PSPACE**-complete. Thus, if we have an interactive-proof protocol for recognizing TQBF, then we have a protocol for recognizing any language in **PSPACE**. In the rest of the lecture, we will provide a protocol for recognizing TQBF.

19.2 Arithmetization of TQBF

We will consider the following quantified boolean formula:

$$\Phi = \forall x_1 \exists x_2 \forall x_3 \dots \forall x_n \quad \phi(x_1, x_2, \dots, x_n),$$

where $\phi(x_1, x_2, \dots, x_n)$ is a $3-CNF$ boolean formula. Without loss of generality, we can assume that all the quantified boolean formulas given to us have this form. The main idea in the interactive protocol for recognizing TQBF is the same as proving $\sharp$**P** $\subseteq$ **IP** from the last lecture: we will first arithmetize the quantified boolean formula and then the prover will convince the verifier that the arithmetized expression evaluates to 1. In the following, all the random elements are drawn from a field $\{0, 1, \dots, p-1\}$, where p is sufficiently large.

19.2.1 Naive Solution

So how do we arithmetize the quantified formula? we will first show a naive solution and point out its problems. To begin with, we know from the last lecture how to arithmetize $\phi(x_1, x_2, \ldots, x_n)$ to obtain the polynomial $F(x_1, x_2, \ldots, x_n)$. Recall that F has degree $3m$ in each variable (m is the number of clauses in ϕ) and agrees with ϕ for all $0-1$ assignments to the n variables. Now we read the quantifiers from right to left and consider the expression $\forall x_n \phi(x_1, x_2, \ldots, x_n)$. This expression has $n-1$ free variables, and for each substitution of values to these variables the expression itself either evaluates to true or false. We're looking for a polynomial that has the same behavior. Using F we can now write a new polynomial $P_{\forall x_n} F(x_1, x_2, \ldots, x_n)$, which is equal to

$$F(x_1, x_2, \ldots, x_{n-1}, 0) \cdot F(x_1, x_2, \ldots, x_{n-1}, 1).$$

Next consider the previous quantifier $\exists x_{n-1}$. We want to find a polynomial representation for $\exists x_{n-1} \forall x_n \phi(x_1, x_2, \ldots, x_n)$, given the polynomial representation $G(x_1, x_2, \ldots, x_{n-1})$ for $\forall x_n \phi(x_1, x_2, \ldots, x_n)$. The following polynomial, which we denote by $P_{\exists x_{n-1}} G(x_1, x_2, \ldots, x_{n-1})$, satisfies the condition:

$$1 - (1 - G(x_1, x_2, \ldots, x_{n-2}, 0)) \cdot (1 - G(x_1, x_2, \ldots, x_{n-2}, 1)).$$

In the notations introduced above, the polynomial for $\exists x_{n-1} \forall x_n \phi(x_1, x_2, \ldots, x_n)$ is

$$P_{\exists x_{n-1}} P_{\forall x_n} F(x_1, x_2, \ldots, x_n).$$

In general, we transform the quantified boolean formula Φ into an arithmetic expression using operators $P_{\forall u}$ and $P_{\exists v}$ as follows:

- Turn the 3-CNF formula $\phi(x_1, x_2, \ldots, x_n)$ into the polynomial $F(x_1, x_2, \ldots, x_n)$ as done in the last lecture.
- Replace the quantifier $\exists x_i$ by the operator $P_{\exists x_i}$.
- Replace the quantifier $\forall x_i$ by the operator $P_{\forall x_i}$.

The final expression after the transformation always evaluates to 0 or 1. It evaluates to 1 if and only if the quantified boolean formula Φ is true.

We have thus arrived at a way of representing quantified boolean formulas using arithmetic operations. Now consider the arithmetic expression after the transformation

$$P_{\forall x_1} P_{\exists x_2} \ldots P_{\forall x_n} F(x_1, x_2, \ldots, x_n)$$

and it evaluates to either 0 or 1. Let us apply the idea from the last lecture to derive a protocol and see what happens. In the first step, the prover eliminates the first operator $P_{\forall x_1}$ and sends the verifier the polynomial $G_1(x_1)$ corresponding to the rest of the arithmetic expression:

$$G_1(x_1) = P_{\exists x_2} P_{\forall x_3} \ldots P_{\forall x_n} F(x_1, x_2, \ldots, x_n).$$

Upon receiving the polynomial $\hat{G_1}(x_1)$ (which the prover claims to be $G_1(x_1)$), the verifier checks that $\hat{G_1}(0) \cdot \hat{G_1}(1) = 1$. The verifier then picks a random value r_1, computes $\beta_1 = \hat{G_1}(r_1)$ and sends r_1 to the prover. The prover then needs to convince the verifier that

$$\beta_1 = P_{\exists x_2} P_{\forall x_3} \dots P_{\forall x_n} F(r_1, x_2, \dots, x_n).$$

The prover then proceeds by eliminating the operator one after another, and in the end the verifier checks that $\beta_n = F(r_1, r_2, \dots, r_n)$.

However, since each operator potentially doubles the degree of each variable in the expression, the degree of the polynomial $G_1(x_1)$ in the end can very well be exponential. This means that the prover will have to send the verifier exponentially many coefficients, but the polynomially bounded verifier will not be able to read them all.

19.2.2 Revised Solution

We will ensure that the degree of any variable in any intermediate stage of the transformation never goes above two. The arithmetization given above clearly does not satisfy this property. Yet there is a simple way to fix this. At any stage of the transformation, we have a polynomial $J(x_1, x_2, \dots, x_n)$ where some variables' degree might exceed two. Normally, we can't expect to transform J into a new polynomial $J'(x_1, x_2, \dots, x_n)$ where the degree of any variable is at most two while still evaluating to the same value at all points. Yet, notice here that we only need J' to agree with J on all $0-1$ assignments. What J' does at any other point is of no interests to us. The key observation then is that for $x = 0$ or $x = 1$, $x^k = x$ for all positive integers k. The desired polynomial J' can thus be obtained from J by erasing all exponents, that is, replacing them with 1. For example, if $J(x_1, x_2, x_3) = x_1^3 x_2^4 + 5x_1 x_2^3 + x_2^6 x_3^2$, the transformed polynomial J' is $6x_1x_2 + x_2x_3$. In general, we define a new operator, Rx_i, which when applied to a polynomial reduces the exponents of x_i to 1 at all occurrences. More formally, we have

$$\begin{aligned} Rx_i J(x_1, \dots, x_n) = x_i \cdot J(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n) + \\ (1 - x_i) \cdot J(x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n). \end{aligned}$$

In this notation, we have

$$J'(x_1, \dots, x_n) = Rx_1 Rx_2 \dots Rx_n J(x_1, \dots, x_n).$$

In the revised arithmetization process, the only change we make is to apply the reduce operators whenever the degree of any variable can potentially be above 1 in the polynomial obtained from the previous step. In particular, we apply the reduce operators after the 3-CNF formula ϕ is arithmetized into the polynomial $F(x_1, \dots, x_n)$. Also, since the other two operators, $P_{\forall u}$ and $P_{\forall v}$, might double the degree of some variables in the transformation, we apply the reduce operators after each application of $P_{\forall u}$ or $P_{\exists v}$. This revision will ensure that the degree of any variable at any intermediate stage never goes above 2. The degree of some variable might reach 2 after a $P_{\forall u}$ or $P_{\exists v}$ operator, but will be reduced back to 1 after the reduce operators are applied.

19.3 The Interactive Protocol

Using the revised arithmetization, we arithmetize the quantified boolean formula of the form

$$\Phi = \forall x_1 \exists x_2 \forall x_3 \ldots \forall x_n \phi(x_1, \ldots, x_n)$$

into

$$E = P_{\forall x_1} Rx_1 P_{\exists x_2} Rx_1 Rx_2 P_{\forall x_3} Rx_1 Rx_2 Rx_3 P_{\exists x_4} \ldots P_{\forall x_n} Rx_1 \ldots Rx_n F(x_1, \ldots, x_n).$$

Now the idea of the interactive protocol from the last lecture can be put to use without any trouble. The prover and the verifier communicate to eliminate one operator at each stage. In the end, no operators are left and the verifier checks the required condition.

Initially, the prover eliminates the first operator $P_{\forall x_1}$ and sends the verifier the polynomial $G_1(x_1)$ corresponding to the rest of the arithmetic expression:

$$G_1(x_1) = Rx_1 P_{\exists x_2} \ldots Rx_n F(x_1, \ldots, x_n).$$

Notice that this is a polynomial of degree 1 instead of exponential degree in the original naive arithmetization. Upon receiving the polynomial $\hat{G_1}(x_1)$(which the prover claims to be $G_1(x_1)$), the verifier checks that $P_{\forall x_1} \hat{G_1}(x_1) = \hat{G_1}(0) \cdot \hat{G_1}(1) = 1$. The verifier then picks a random value r_1, computes $\beta_1 = \hat{G_1}(r_1)$ and sends r_1 to the prover. (In this stage, the verifier actually doesn't have to send r_1 to the prover, but for consistency with later rounds we let the verifier send r_1 to the prover. The soundness of the protocol remains.) The prover then needs to convince the verifier that

$$\beta_1 = Rx_1 P_{\exists x_2} \ldots Rx_n F(r_1, x_2, \ldots, x_n).$$

The prover then eliminates the second operator Rx_1 and sends the verifier the polynomial $G_2(x_1)$ corresponding to the rest of the arithmetic expression:

$$G_2(x_1) = P_{\exists x_2} \ldots Rx_n F(x_1, \ldots, x_n).$$

Notice that this is a polynomial of degree at most 2. Upon receiving the polynomial $\hat{G_2}(x_1)$(which the prover claims to be $G_2(x_1)$), the verifier checks that $(Rx_1 \hat{G_2}(x_1))[r_1] = \hat{G_1}(r_1)$, where $[r_1]$ means that the polynomial in the parenthesis with free variable x_1 is evaluated at point r_1. The verifier then picks a random value r_2 for x_1, computes $\beta_2 = \hat{G_1}(r_2)$ and sends r_2 to the prover.

In general, when the verifier needs to be convinced that

$$\beta = (Rx_i P(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}, x_i))[\alpha_i],$$

he asks for the polynomial $G(x_i) = P(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}, x_i)$, and gets $\hat{G}(x_i)$. ($\alpha_j (1 \leq j \leq i)$ is the most recent random value the verifier assigned to variable x_j.) He verifies that $(Rx_i \hat{G}(x_i))[\alpha_i] = \beta$ and computes $\beta' = \hat{G}(\alpha'_i)$ by choosing a new random value α'_i for x_i. Now the verification is reduced to whether

$$\beta' = P(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}, \alpha'_i).$$

The case of the $P_{\forall u}$ or $P_{\exists v}$ operator is the same as in the last lecture where we proved $\sharp\mathbf{P} \subseteq \mathbf{IP}$.

The prover proceeds by eliminating the operators one after another, and in the end the verifier checks that $q(\alpha_n) = F(\alpha_1, \ldots, \alpha_n)$, where q is the polynomial the prover sends to the verifier in the last round and α_i is the most recent random value the verifier assigned to variable x_i.

19.4 Analysis

Theorem 65
(a) If $\Phi \in$ TQBF, then there exists a prover such that $Pr[Verifier\ accepts] = 1$.
(b) If $\Phi \notin$ TQBF, then for all provers $Pr[Verifier\ accepts] \leq 1/2$.

Proof: Part (a) is obvious. For part (b), the analysis is the same as in the case of $\sharp\mathbf{P} \subseteq \mathbf{IP}$ from the last lecture. The error can be introduced only when the verifier happens to pick a root of some low degree polynomial at some round. While dealing with the last n reduce operators, the polynomials may have degree at most $3m$. For the remaining $n(n-1)/2$ reduce operators, the degree is at most 2. For every other operator (n operators of the form $P_{\forall u}$ or $P_{\exists v}$) the degree of the polynomial is 1. Thus, the sum of the degrees of all the polynomials encountered in the interaction is at most $3mn + n^2$. Thus, if we choose the size of our field p large enough, the probability of error is at most $(3mn + n^2)/p \leq 1/2$. $\Box$

Chapter 20

PCP and Hardness of Approximation

April 9, 2001, Scribe: François Labelle

In this lecture we see the connection between **PCP** and hardness of approximation. In particular, we show that the **PCP** Theorem implies that MAX-3SAT cannot be approximated within $(1-\epsilon)$ for some $\epsilon > 0$, unless $\mathbf{P} = \mathbf{NP}$. Also, we show that Max Clique cannot be approximated within any constant factor.

20.1 Probabilistically Checkable Proofs

Recall the following definition and theorem:

Definition 29 *Given functions $r(n)$, $q(n)$, we say that $L \in \mathbf{PCP}(r(n), q(n))$ if there is a polynomial time probabilistic verifier V, which is given x, oracle access to a proof π, uses $r(|x|)$ random bits, reads $q(|x|)$ bits of π and such that*

$$x \in L \Rightarrow \exists \pi \cdot \mathbf{Pr}[V^{\pi}(x) = 1] = 1$$
$$x \notin L \Rightarrow \forall \pi \cdot \mathbf{Pr}[V^{\pi}(x) = 1] \leq \tfrac{1}{2}.$$

Theorem 66 (**PCP** Theorem)
$\mathbf{NP} = \mathbf{PCP}(O(\log n), O(1))$.

20.2 PCP and MAX-3SAT

20.2.1 Approximability

Definition 30 (MAX-3SAT) *Given a 3CNF formula φ (i.e. with at most 3 variables per clause), find an assignment that satisfies the largest number of clauses.*

Note that MAX-3SAT generalizes 3SAT, and so cannot be solved in polynomial time unless $\mathbf{P} = \mathbf{NP}$. However, it is easy to approximate it within a factor of 2:

Algorithm. Output the best solution between all-true and all-false.

Analysis. Every clause is satisfied by at least one of the two solutions, therefore one of the solutions satisfies at least half of the clauses.

It is possible to do better. The Karloff-Zwick algorithm runs in polynomial time and satisfies $\geq \frac{7}{8}$ times the optimal number of clauses.

Some problems admit polynomial time algorithms with approximation ratio $\geq (1-\epsilon)\ \forall\epsilon$. The **PCP** Theorem implies this is not the case for MAX-3SAT, as we see next.

20.2.2 Inapproximability

Theorem 67

The **PCP** *Theorem implies that there is an $\epsilon > 0$ such that $(1-\epsilon)$-approximation of MAX-3SAT is* **NP***-hard.*

Proof: Fix any **NP**-complete problem L. By the **PCP** Theorem, $L \in \mathbf{PCP}(O(\log n), O(1))$. Let V be the verifier for L.

Given an instance x of L, our plan is to construct a 3CNF formula φ_x on m variables such that (for some $\epsilon > 0$ to be determined)

$$\begin{array}{lll} x \in L & \Rightarrow & \varphi_x \text{ is satisfiable} \\ x \notin L & \Rightarrow & \text{no more than } (1-\epsilon)m \text{ clauses of } \varphi_x \text{ are satisfiable.} \end{array} \tag{20.1}$$

Without loss of generality, assume that V makes non-adaptive queries (by reading all the bits that could possibly be needed at once). This assumption is valid because the number of queries was a constant and remains a constant. Let q be the number of queries.

Enumerate all random strings R for V. The length of each string is $r(|x|) = O(\log |x|)$, so the number of such strings is polynomial in $|x|$. For each R_i, V chooses q positions $i_1, \ldots, i_q$ and a Boolean function $f_R : \{0,1\}^q \to \{0,1\}$ and accepts iff $f_R(\pi(i_1), \ldots, \pi(i_q))$.

We want a Boolean formula to simulate this. Introduce Boolean variables $x_1, \ldots, x_l$, where l is the length of the proof. Now we need a correspondence between the number of satisfiable clauses and the probability that the verifier accepts.

For every R we add clauses that represent $f_R(x_{i_1}, \ldots, x_{i_q})$. This can be done with 2^q SAT clauses. We need to convert clauses of length q to length 3 with additional variables. e.g. $x_2 \vee x_{10} \vee x_{11} \vee x_{12}$ becomes $(x_2 \vee x_{10} \vee y_R) \wedge (\overline{y}_R \vee x_{11} \vee x_{12})$. This requires at most $q2^q$ 3SAT clauses. In general there is a series of auxiliary variables for each R.

If $z \in L$, then there is a proof π such that $V^\pi(z)$ accepts for every R. Set $x_i = \pi(i)$ and auxiliary variables in the right way, all clauses are satisfied.

If $z \notin L$, then for every assignment to $x_1, \ldots, x_l$ (and to y_R's), the corresponding proof $\pi(i) = x_i$ makes the verifier reject for half of the $R \in \{0,1\}^{r(|z|)}$. For each such R, one of the clauses representing f_R fails. Therefore a fraction $\epsilon = \frac{1}{2}\frac{1}{q2^q}$ of clauses fails. □

It turns out if this holds for every **NP**-complete problem, then the **PCP** Theorem must be true.

Theorem 68

If there is a reduction as in (20.1) for some problem L in **NP***, then $L \in \mathbf{PCP}(O(\log n), O(1))$, i.e. the* **PCP** *Theorem holds for that problem.*

Proof: We describe how to construct a verifier for L. V on input z expects π to be a satisfying assignment for φ_z. V picks $O(\frac{1}{\epsilon})$ clauses of φ_z at random, and checks that π satisfies all of them. Randomness is $O(\frac{1}{\epsilon}\log m) = O(\log|z|)$. $O(\frac{1}{\epsilon}) = O(1)$ bits are read in the proof.

$$
\begin{aligned}
z \in L &\Rightarrow \varphi_z \text{ is satisfiable} \\
&\Rightarrow \exists\pi \text{ such that } V_\pi(z) \text{ always accept.}
\end{aligned}
$$

$$
\begin{aligned}
z \notin L &\Rightarrow \forall\pi \text{ a fraction } \tfrac{1}{\epsilon} \text{ of clauses are unsatisfied} \\
&\Rightarrow \forall\pi\ V^\pi(z) \text{ will reject with probability } \geq \tfrac{1}{2} \quad \text{(details omitted).}
\end{aligned}
$$

□

20.2.3 Tighter result

What is the best known value of ϵ in Theorem 67 ? We state without proof the following result:

Theorem 69 (Håstad)
There is a **PCP** *verifier for* **NP** *that uses $O(\log n)$ bits of randomness which, based on R, chooses 3 positions i_1, i_2, i_3 and a bit b and accepts iff $\pi(i_1) \oplus \pi(i_2) \oplus \pi(i_3) = b$. The verifier satisfies*

$$z \in L \Rightarrow \exists\pi \cdot \mathbf{Pr}[V^\pi(x) = 1] \geq 1 - \epsilon \tag{20.2}$$

$$z \notin L \Rightarrow \forall\pi \cdot \mathbf{Pr}[V^\pi(x) = 1] \leq \tfrac{1}{2} + \epsilon \tag{20.3}$$

Note the slightly non-standard bounds for the probabilities above. If we had "$= 1$" in Equation (20.2), we could find a proof π by solving a system of linear equations (which would imply $\mathbf{P} = \mathbf{NP}$). For every R, one can encode $x_{i_1} \oplus x_{i_2} \oplus x_{i_3} = b$ with 4 clauses in 3CNF.

If $z \in L$, then a fraction $\geq (1-\epsilon)$ of clauses are satisfied.

If $z \notin L$, then for a $(\frac{1}{2} - \epsilon)$ fraction of R, $\frac{1}{4}$ of clauses are contradicted.

This is not producing a formula directly, but is enough to prove the hardness of approximation ratio:

$$\frac{1 - \frac{1}{4}(\frac{1}{2} - \epsilon)}{1 - \epsilon} = \frac{7}{8} + \epsilon'$$

20.3 Max Clique

20.3.1 Approximability

Definition 31 (Max Clique) *Given an undirected $G = (V, E)$, find the largest $C \subseteq V$ such that $\forall u, v \in C$, $(u, v) \in E$. When convenient, we use the complementary graph equivalent "Max Independent Set" which is the largest $I \subseteq V$ such that $\forall u, v \in I$, $(u, v) \notin E$.*

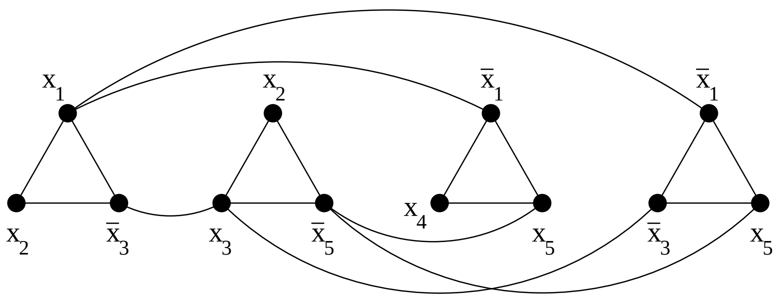

Figure 20.1: Graph construction corresponding to the 3CNF formula $\varphi = (x_1 \vee x_2 \vee \overline{x}_3) \wedge (x_2 \vee x_3 \vee \overline{x}_5) \wedge (\overline{x}_1 \vee x_4 \vee x_5) \wedge (\overline{x}_1 \vee \overline{x}_3 \vee x_5)$.

Finding a largest clique is **NP**-hard, so we ask about approximations. There is a simple $\frac{\log n}{n}$-approximation:

- arbitrarily partition the graph into subsets of size $\log n$
- solve exactly each subset
- report the largest clique found

Suppose there is a clique of size αn, then one of the subset contains a clique of size $\alpha \log n$, hence the approximation. Best known is a $\frac{(\log n)^2}{n}$-approximation algorithm.

20.3.2 Inapproximability

We apply the **PCP** Theorem to Max Independent Set. Given a 3CNF formula $\varphi = c_1 \wedge c_2 \wedge \ldots \wedge c_m$ over variables $x_1, \ldots, x_n$ arising from Theorem 67, we build a graph using the same construction as in the **NP**-completeness proof of Independent Set in Lecture 3 (see Figure 20.1).

The graph has $3m$ vertices. Note that G has an independent set of size m iff φ is satisfiable, and G has an independent set with k vertices iff φ has an assignment that satisfies k clauses. So the hardness results for MAX-3SAT translate directly to Max Independent Set. Below we show how this can be further improved.

Given $G = (V, E)$, consider $G' = (V', E')$ where $V' = V \times V$ and $((u_1, u_2), (v_1, v_2)) \in E'$ iff $(u_1, v_1) \in E$ OR $(u_2, v_2) \in E$. It can be shown that the size of the maximum independent set in G' is equal to (size of max ind. set in $G)^2$. As a sidenote, this result is not true for cliques (a clique in G' could be possibly larger than expected), unless we use a different definition of G'.

By a constant number of applications of the graph product operation, we get:

graph	G	G'	G''
size of ind. set if φ is satisfiable	m	m^2	m^4
size of ind. set if only $(1-\epsilon)m$ clauses of φ are satisfiable	$(1-\epsilon)m$	$(1-\epsilon)^2 m^2$	$(1-\epsilon)^4 m^4$

The ratio decreases without bound, therefore there can be no constant approximation algorithm for Max Clique. In fact:

Theorem 70 (Håstad)
If $\mathbf{NP} \neq \mathbf{ZPP}$, *then* $\forall \epsilon > 0$, *Max Independent Set cannot be approximated within* $\frac{1}{n^{1-\epsilon}}$ *(*n *is the number of vertices).*

Chapter 21

NP = PCP [log n, polylog n]

April 17,2001, Scribe: Lawrence Ip and Beini Zhou

In this lecture, we will prove that $\mathbf{NP} = \mathbf{PCP}(O(\log n), poly \log n)$. Since one direction is easy to see, we will prove below that $\mathbf{NP} \subseteq \mathbf{PCP}(O(\log n), poly \log n)$.

21.1 Overview

It suffices to prove that the **NP**-complete problem 3SAT $\in \mathbf{PCP}(O(\log n), poly \log n)$. There are three major components in proving this result:

- The verifier expects the proof π to consist of a low-degree polynomial $P : \mathcal{F}^l \to \mathcal{F}$ and P is represented by its value at each of the $|\mathcal{F}|^l$ points. The verification procedure picks a random constraint on the polynomial P among $poly(n)$ constraints. If the 3SAT-formula ϕ is satisfiable, then there exists a low-degree polynomial P that satisfies all the constraints; on the other hand, if ϕ is not satisfiable, then less than half of the constraints are satisfied.

- Above we require that the proof consists of a low-degree polynomial. However, there is no guarantee that the prover will oblige by conforming to this restriction. Thus, we need an efficient way to enforce this low-degree restriction on the proof, and this is given by low-degree tests. Ideally, we would like to have a test that, given access to an oracle $f : \mathcal{F}^l \to \mathcal{F}$, outputs yes with probability 1 if f is a degree $\leq d$ polynomial and outputs no with high probability otherwise. This, however, is not possible, since we can have an f which disagrees with a degree d polynomial at one point and agrees with the polynomial everywhere else, and thus passes any test that only queries f at poly(m, d) places with high probability. We thus relax the soundness condition and only require the low-degree test to reject with high probability if f disagrees with any degree d polynomial on more than δ-fraction of the points with an appropriately chosen δ.

- The gap between the completeness and soundness of the low-degree test above still leaves us with a problematic situation: assume the prover provides a function f :

$\mathcal{F}^l \to \mathcal{F}$ that agrees with a degree d polynomial p at more than $1-\delta$-fraction of the points, and it passes the low-degree test. Then, we need to check a random constraint on p, but how can we get access to the polynomial p with oracle access to f? The polynomial reconstruction algorithm computes $p(x)$ for any given $x \in \mathcal{F}^l$ with high probability given oracle access to f.

Armed with the above three major components, we can now describe our **PCP** verifier. The verifier is given a 3SAT instance ϕ. The prover then supplies an oracle for a function $f : \mathcal{F}^l \to \mathcal{F}$. The verification procedure proceeds as follows:

1. First run the low-degree tests on f. Reject if the test rejects.
2. Pick a random constraint to see if it satisfies on f. Use the polynomial reconstruction algorithm to do the checking. Reject if the constraint does not pass.
3. Accept otherwise.

In the verification procedure, we make sure that we use $O(\log n)$ random bits and query *poly* $\log n$ bits from the oracle. In the following, we will provide details on each of the three components.

21.2 Arithmetization of 3SAT

We begin by describing how to arithmetize 3SAT. An instance ϕ of 3SAT consists of n variables and t clauses where each clause is of the form $\{x_{i_1} = b_1 \vee x_{i_2} = b_2 \vee x_{i_3} = b_3\}$ where each $b_j \in \{0,1\}$. In this way, we can view ϕ as an indicator function $\phi : \{1,2,\dots,n\}^3 \times \{0,1\}^3 \to \{0,1\}$ where $\phi(i_1,i_2,i_3,b_1,b_2,b_3) = 1$ if there is a clause of the form $\{x_{i_1} = b_1 \vee x_{i_2} = b_2 \vee x_{i_3} = b_3\}$ in the instance ϕ.

To arithmetize ϕ, we pick h and m, where $h = \log n$ and $m = \log n / \log\log n$ so that $h^m = n$. Now, set $H = \{1,2,\dots,h\}$, denote h_i the ith element in H and identify $\{1,2,\dots,n\}$ with H^m in some canonical way (e.g. representation in base h). Extending $b_j (\in \{0,1\})$ to H, the instance ϕ can be viewed as an m'-variate function $\phi : H^{m'} \to \{0,1\}$ where $m' = 3m+3$. (Whenever $b_j \notin \{0,1\}$, ϕ is evaluated to 0.) Similarly, an assignment $A : \{1,2,\dots,n\} \to \{0,1\}$ can be viewed as a function $A : H^m \to \{0,1\}$.

In this language, 3SAT can be restated as follows: ϕ is satisfiable if and only if there exists an assignment $A : H^m \to \{0,1\}$ such that for all $i_1, i_2, i_3 \in H^m$ and for all $b_1, b_2, b_3 \in H$,

$$\phi(i_1,i_2,i_3,b_1,b_2,b_3)(A(i_1)-b_1)(A(i_2)-b_2)(A(i_3)-b_3) = 0 \tag{21.1}$$

Now we introduce the notion of polynomial codes: we fix a field $\mathcal{F}$of size $\log^3 n$ that contains H and extend the function $\phi : H^{m'} \to \{0,1\}$ to a low-degree polynomial $\hat{\phi} : \mathcal{F}^{m'} \to \mathcal{F}$ such that $\phi(x) = \hat{\phi}(x)$ for all $x \in H^{m'}$. The following Lemma says that such a low-degree polynomial exists:

Lemma 71
For every function $f : H^m \to \mathcal{F}$ there is a polynomial $\hat{f} : F^m \to \mathcal{F}$ such that $f(x) = \hat{f}(x)$ for all $x \in H^m$. Every variable in $\hat{f}$ has degree at most h, and $\hat{f}$ is a degree hm polynomial. $\hat{f}$ is called a low-degree extension of f. (The extension might not be unique.)

Similarly, we extend the assignment function $A : H^m \to \{0, 1\}$ to a low-degree polynomial $\hat{A} : \mathcal{F}^m \to \mathcal{F}$ such that $A(x) = \hat{A}(x)$ for all $x \in H^m$. Now ϕ is satisfiable if and only if there exists an assignment polynomial $\hat{A} : F^m \to \mathcal{F}$ such that for all $i_1, i_2, i_3 \in H^m$ and for all $b_1, b_2, b_3 \in H$,

$$\hat{\phi}(i_1, i_2, i_3, b_1, b_2, b_3)(\hat{A}(i_1) - b_1)(\hat{A}(i_2) - b_2)(\hat{A}(i_3) - b_3) = 0 \tag{21.2}$$

Denote the polynomial in the above equation to be $\hat{P}_1$. Since $\hat{\phi}$ and $\hat{A}$ have degree at most h in each variable, $\hat{P}_1$ is an m'-variate polynomial of degree at most $2h$ in each variable and thus has (total) degree at most $2hm'$.

21.2.1 First Try

Now let us assume the proof π contains not only the assignment polynomial P_0 claimed to be $\hat{A}$, but also a polynomial P_1 that is supposed to be $\hat{P}_1$. The verifier needs to check the following constraints:

- (C0). $P_1(x) = \hat{P}_1(x)$ for all $x \in \mathcal{F}^{m'}$
  (The verifier can efficiently check constraints C0 since he can efficiently compute $\hat{\phi}(x)$ and thus $\hat{P}_1(x)$ once given the assignment polynomial P_0.)
- (C00). $P_1(x) = 0$ for all $x \in H^{m'}$

Constraints C0 satisfy the soundness requirement of **PCP**, that is, if $P_1 \neq \hat{P}_1$, then $\mathbf{Pr}_x[P_1(x) = \hat{P}_1(x)] \leq 1/2$. This follows from the following Schwartz-Zippel Lemma:

Lemma 72 (Schwartz-Zippel)
Two distinct polynomials $\mathcal{F}^m \to \mathcal{F}$ of (total) degree d disagree on at least $1 - \frac{d}{|\mathcal{F}|}$ fraction of points in $\mathcal{F}^m$.

However, constraints C00 do not satisfy the soundness requirement of **PCP**, because it's possible for P_1 to be zero on all but one point of $H^{m'}$ and thus to pass the test with high probability if we evaluate P_1 at a random point in $H^{m'}$.

21.2.2 Second Try

We will construct a sequence of polynomials which are zero on a larger and larger fraction of points and eventually the last polynomial would be zero on all points and thus can be efficiently tested. Specifically we will define a sequence of low-degree polynomials $P_2, P_3, \ldots, P_{m'+1}$ such that $P_1 = 0$ over $H^{m'}$ if and only if $P_2 = 0$ over $\mathcal{F} \times H^{m'-1}$, and in general for $1 \leq i \leq m' + 1$, $P_i = 0$ over $\mathcal{F}^{i-1} \times H^{m'-i+1}$ if and only if $P_{i+1} = 0$ over $\mathcal{F}^i \times H^{m'-i}$. Hence, $P_{m'+1}$ will be identically zero on $F^{m'}$ if and only if P_1 is identically zero on $H^{m'}$. Each of these constraints will satisfy the soundness requirement of **PCP**.

To start with, let us first see how to construct P_2 from P_1. Given $P_1 : \mathcal{F}^{m'} \to \mathcal{F}$, define $P_2 : \mathcal{F}^{m'} \to \mathcal{F}$ to be:

$$P_2(y_1, x_2, x_3, \ldots, x_{m'}) = \sum_{j=1}^{|H|} P_1(h_j, x_2, x_3, \ldots, x_{m'}) y_1^j. \tag{21.3}$$

Clearly, if P_1 is identically zero on $H^{m'}$, then P_2 is identically zero on $\mathcal{F} \times H^{m'-1}$. Conversely, if $P_1(z_1, \ldots, z_{m'}) \neq 0$ for some $z \in H^{m'}$, then $P_2(y_1, z_2, \ldots, z_{m'})$ is some non-zero univariate polynomial of degree at most $|H|$ in y_1 and so is non-zero on at least $|\mathcal{F}| - |H|$ points. Thus, the converse also holds.

In constructing the sequence of polynomials, we apply the above transformation, once in each variable. In general, in transforming P_{i-1} to P_i, we change variable x_{i-1} to y_{i-1} as described above for the case $i = 2$, namely, for $2 \leq i \leq m' + 1$, define

$$P_i(y_1, \ldots, y_{i-1}, x_i, \ldots, x_{m'}) = \sum_{j=1}^{|H|} P_{i-1}(y_1, \ldots, y_{i-2}, h_j, x_i, \ldots, x_{m'}) y_{i-1}^j. \tag{21.4}$$

Note that $P_1, P_2, \ldots, P_{m'+1}$ all have degree at most $2hm'$. By the same reasoning as above, we can see that $P_i = 0$ over $\mathcal{F}^{i-1} \times H^{m'-i+1}$ if and only if $P_{i+1} = 0$ over $\mathcal{F}^i \times H^{m'-i}$.

Now let us assume that the proof π contains a sequence of polynomials $P_0, P_1, \ldots, P_{m'+1}$ where P_0 is an m-variate assignment polynomial of degree at most mh and $P_1, \ldots, P_{m'+1}$ are m'-variate polynomials of degree at most $2hm'$. The verifier needs to check the following constraints: for all $x \in \mathcal{F}^{m'}$,

- (C1). $P_1(x) = \hat{P}_1(x)$
  ($\hat{P}_1$ is as defined in equation 2 above)
- (Ci). For all $2 \leq i \leq m' + 1$
  $P_i(x_1, \ldots, x_{m'}) = \sum_{j=1}^{|H|} P_{i-1}(x_1, \ldots, x_{i-2}, h_j, x_i, \ldots, x_{m'}) x_{i-1}^j$
  (condition from equation 4 above at point x)
- (C(m'+2)). $P_{m'+1}(x) = 0$.

There are basically $|\mathcal{F}|^{m'}$ constraints $C(x)$, one for every point $x \in \mathcal{F}^{m'}$. The constraint for point x is defined in terms of the above $m' + 2$ constraints as follows:

$$C(x) = \bigwedge_{j=1}^{m'+2} (Cj)(x). \tag{21.5}$$

In the verification procedure, the verifier picks a random constraint $z \in \mathcal{F}^{m'}$ and accepts if $C(z)$ passes. Clearly, if ϕ is satisfiable, there exists a proof consisting of a sequence of polynomials that satisfies every constraint. On the other hand, if ϕ is not satisfiable, for all proofs consisting of a sequence of polynomials of the above specified form, less than half of the constraints satisfy. In other words, the above $|\mathcal{F}|^{m'}$ constraints satisfy the soundness requirement of **PCP**. This follows from the following Lemma:

Lemma 73
If there exist polynomials $P_0, P_1, \ldots, P_{m'+1}$ of degree specified above such that

$$\forall 1 \leq j \leq m' + 1 \quad \mathbf{Pr}_{x \in \mathcal{F}^{m'}}[\textit{Constraint Cj above satisfies}] \geq \frac{2hm' + h}{|\mathcal{F}|} \tag{21.6}$$

$$\bigwedge \quad \mathbf{Pr}_{x \in \mathcal{F}^{m'}}[\textit{Constraint C(m'+2) above satisfies}] \geq \frac{2hm'}{|\mathcal{F}|}, \tag{21.7}$$

then ϕ is satisfiable.

Proof: The contrapositive of the statement follows from the Schwartz-Zippel Lemma (Lemma 2 above) since all the polynomials involved in the constraints Cj ($1 \leq j \leq m' + 1$) have degree at most $2hm' + h$. $\square$

It follows that if ϕ is not satisfiable, no proof consisting of a sequence of polynomials of the above specified form is accepted with probability greater than $\frac{2hm'+h}{|\mathcal{F}|}$, and this probability is less than a given ϵ for sufficiently large n.

21.2.3 Bundling Polynomials into a Single Polynomial

In the above, we have a sequence of $O(\log n)$ polynomials involved in the proof and constraints. It turns out that in order to reduce the randomness complexity of the verifier down to $O(\log n)$, we need to bundle the sequence of polynomials $P_0, P_1, \ldots, P_{m'+1}$ into a constant number of polynomials. Below we show how to bundle the $m' + 2$ polynomials into a single polynomial P.

We want to define a low-degree polynomial $P : \mathcal{F}^{m'+1} \to \mathcal{F}$ such that for all $x \in \mathcal{F}^m$, $P(i, x) = P_i(x)$. The existence of such a polynomial is guaranteed by the following Lemma:

Lemma 74
Given polynomials $q_0, q_1, \ldots, q_m : \mathcal{F}^l \to \mathcal{F}$ with $|\mathcal{F}| > m$ and each of (total) degree at most d, there exists a degree $d + m$ polynomial $Q : \mathcal{F}^{l+1} \to \mathcal{F}$ such that for $i = 0, 1, \ldots, m$ and for all $x \in \mathcal{F}^m$, $Q(i, x) = q_i(x)$.

Proof: For each $i \in \{0, 1, \ldots, m\}$, there is a unique univariate polynomial δ_i of degree m such that

$$\delta_i(a) = \begin{cases} 1 & if\ a = i \\ 0 & if\ 0 \leq a \leq m\ but\ a \neq i. \end{cases}$$

Now define polynomial Q as $Q(a, x) = \sum_{i=0}^{m} \delta_i(a) q_i(x)$. Then, we have $Q(i, x) = q_i(x)$ for all $x \in \mathcal{F}^m$. $\square$

Now instead of containing a sequence of polynomials, the proof consists of one single polynomial P of degree $2hm' + m' + 1$. The verifier picks a random constraint $z \in \mathcal{F}^{m'}$ and checks that $C(z)$ passes as described in section 2, except that he queries at $P(i, x)$ whenever he needs the value $P_i(x)$. Thus we can now characterize the constraint C as follows: C has the form $(x_1, x_2, \ldots, x_{w(n)}, C)$, where each $x_i \in \mathcal{F}^{m'+1}$ and depends on z, C is a $w(n)$-variate function $\mathcal{F}^{w(n)} \to \{0, 1\}$ and $w(n)$ is *poly* $\log n$. With this characterization and by Lemma 3, we have the following key Lemma:

Lemma 75
If the given 3SAT instance ϕ is satisfiable, then there exists a degree $2hm' + m' + 1$ polynomial $p : \mathcal{F}^{m'+1} \to \mathcal{F}$ such that for all constraints $z \in \mathcal{F}^{m'}$, $C(p(x_1^{(z)}), p(x_2^{(z)}), \ldots, p(x_{w(n)}^{(z)})) = 1$; if ϕ is not satisfiable, for every degree $2hm' + m' + 1$ polynomial $p : \mathcal{F}^{m'+1} \to \mathcal{F}$, $C(p(x_1^{(z)}), p(x_2^{(z)}), \ldots, p(x_{w(n)}^{(z)})) = 0$ for at least ϵ-fraction of the constraints $z \in \mathcal{F}^{m'}$.

21.3 Low-degree Testing

So far, we have simply assumed that the proof given contains a low-degree polynomial. The soundness result shown above holds only when this happens. But in our setup, the proof has no such restrictions so the verifier must check that the proof consists of a low-degree polynomial.

However, there is no way for the verifier to check that a function is a low-degree polynomial with just polynomially many queries because a function could agree with the polynomial in all but a few points. However, we can check if the function is "close" to a low-degree polynomial. By close we mean the following:

DEFINITION 32 *Function $f : \mathcal{F}^l \to \mathcal{F}$ is said to be δ-close to a polynomial $p : \mathcal{F}^l \to \mathcal{F}$ if $\mathbf{Pr}_x[f(x) \neq g(x)] \leq \delta$ when x is drawn uniformly at random from $\mathcal{F}^l$.*

A low-degree test is given $\delta > 0$, $d \in \mathbb{Z}^+$, and oracle access to $f : \mathcal{F}^l \to \mathcal{F}$ and its task is to verify that f is δ-close to a degree $\leq d$ polynomial. The test works as follows:

1. Fix $t_0, t_1, \ldots, t_{d+1} \in \mathcal{F}$.
2. Pick at random $x, y \in \mathcal{F}^l$ and find the unique degree d polynomial q such that $q(t_i) = f(x + t_i y)$ for $i = 1, \ldots, d + 1$.
3. Verify that $q(t_0) = f(x + t_0 y)$.

The test works by choosing a random line in $\mathcal{F}^l$. The multivariate polynomial restricted to this line becomes a univariate degree d polynomial. We then evaluate at $d+1$ random points on the line, interpolate to find the polynomial, and then test the interpolated polynomial at another random point.

We have the following theorem regarding the above test:

THEOREM 76 (RUBINFELD-SUDAN)
There exists a constant $\delta_0 > 0$ such that for every $\delta < \delta_0$, if the above low-degree test accepts with probability at least $1 - \delta$, then f is 2δ-close to some degree d polynomial.

21.4 Polynomial Reconstruction Algorithm

We are still left with a problematic situation: what to do if the proof contains a function that is δ-close to a degree $2hm' + m' + 1$ polynomial $P : \mathcal{F}^{m'+1} \to \mathcal{F}$ and it passes the low-degree tests? In this case, we would like to pick a random constraint and test on the polynomial P. However, we don't have oracle access to P but only to f that is δ-close to P. Recall that we ran into the same situation in lecture 16 on the average-case complexity of the permanent and there we gave a polynomial reconstruction algorithm to evaluate P at any given point with high probability with oracle access to f. For completeness we restate the algorithm here. Given $x \in \mathcal{F}^l$ and oracle for f which is δ-close to a polynomial p, compute $p(x)$ as follows:

1. Pick $y \in \mathcal{F}^l$ at random.
2. Query $f(x + y), f(x + 2y), \ldots, f(x + (d + 1)y)$ and let $b_1, \ldots, b_{d+1}$ be the responses.

3. Find, by interpolation, the unique degree d univariate polynomial h such that $h(i) = b_i$ for $1 \le i \le d+1$.

4. Output $h(0)$ as the value of $p(x)$.

The error probability of the above reconstruction is given in the following theorem:

THEOREM 77
For any given $x \in \mathcal{F}^l$, the above algorithm computes $p(x)$ with probability at least $1 - d\delta$ with $d+1$ queries.

21.5 Summary

Armed with Lemma 5, Theorem 6 and 7, we can now describe our final **PCP** verifier. The verifier is given a 3SAT instance ϕ. The proof then contains a function $f : \mathcal{F}^{m'+1} \to \mathcal{F}$. The verification procedure proceeds as follows:

1. Run the low-degree test on f with $d = 2m'h + m' + 1$ and an appropriate δ (to be specified below). Reject if the low-degree test rejects.

2. Pick a random constraint $z \in \mathcal{F}^{m'}$ and derive $x_1, x_2, \ldots, x_{w(n)} \in \mathcal{F}^{m'+1}$ based on z. ($w(n)$ is *poly* $\log n$.)

3. Use the polynomial reconstruction algorithm to reconstruct $p(x_1^{(z)}), p(x_2^{(z)}), \ldots, p(x_{w(n)}^{(z)})$. Let the reconstructed values to be $b_1, b_2, \ldots, b_{w(n)}$.

   - In order to save the random bits, the verifier uses the same random $y \in \mathcal{F}^{m'+1}$ in reconstructing each $p(x_i^{(z)})$.

4. Reject if $C(b_1, b_2, \ldots, b_{w(n)}) = 0$.

We set $\delta = O(\frac{1}{d \cdot w(n)})$. First we make sure that the above procedure rejects with probability at least 1/2 when the 3SAT instance ϕ is not satisfiable: If the function f is not δ-close to a degree d polynomial p, then by Theorem 6 the first step rejects with probability at least 1/2. Now we only need to consider the case when f is δ-close to some degree d polynomial p. In this case, we have

$$\begin{aligned}\text{Probability of error} \le\ & \mathbf{Pr}[\text{a random constraint passes the check in step 4}] \\ & + \mathbf{Pr}[\text{there exists an } i \text{ such that } b_i \ne p(x_i^{(z)}) \text{ in step 3}]\end{aligned}$$

For sufficiently large n, the first term on the right-hand side, by Lemma 5, can be made less than 1/6. The second term, by Theorem 7, is upper bounded by $d\delta w(n)$ and can be made less than 1/6 by our choice of δ. Thus, the total error probability can be made less than 1/2.

Now we need to make sure that the query complexity of the verifier is *poly* $\log n$ and the randomness complexity of the verifier is $O(\log n)$.

- Query complexity: the verifier makes queries to the proof f in step 1 and 3. The low-degree test in step 1 makes $\mathit{poly}\log n$ queries; the polynomial reconstruction algorithm in step 3 makes $(d+1)\cdot w(n) = \mathit{poly}\log n$ queries. Thus, the test makes $\mathit{poly}\log n$ queries in total.

- Randomness complexity: the verifier flips coins in step 1, 2 and 3. In the low-degree tests in step 1, in order to reduce the error probability down to 1/2, we need to repeat the test described in Section 3 $O(1/\delta)$ times, which is $\mathit{poly}\log n$ times by our choice of δ. Since each time the test uses $2\mathcal{F}^{m'+1} = O(\log n)$ random bits, the tests would use $\mathit{poly}\log n \cdot O(\log n)$ random bits in total. Yet by using pair-wise independence between the $\mathit{poly}\log n$ trials, the verifier can reduce the number of random bits down to $O(\log\log n) + O(\log n)$, which is $O(\log n)$. In step 2, picking a random constraint takes $m'\log\mathcal{F} = O(\log n)$ bits. In step 3, the verifier picks one random $y \in \mathcal{F}^{m'+1}$ for reconstructing all the values and thus also takes $O(\log n)$ bits. Hence, the test uses $O(\log n)$ random bits in total.

This completes the proof of the **PCP** characterization $\mathbf{NP} = \mathbf{PCP}[O(\log n), \mathit{poly}\log n]$.

Chapter 22

Parallelization

April 16, 2001, Scribe: Mark Pilloff

In a previous lecture we showed that $\mathbf{NP} = \mathbf{PCP}[O(\log n), poly(\log n)]$. Today we will begin a proof that $\mathbf{NP}$ is contained in a variant of $\mathbf{PCP}[O(\log n), poly(\log n)]$ where we make only a *constant* number of queries into the proof, although each query will read $poly(\log n)$ bits of the proof. At first this may not appear to be a useful improvement, but this restricted variant is the foundation of tighter results on the relationship between $\mathbf{NP}$ and $\mathbf{PCP}$. For today we will content ourselves with stating the main results and laying the foundation for the proof. The analysis and justification will follow in the next lecture.

22.1 Main Lemma

Lemma 78

Given a 3CNF formula ϕ, with m clauses on n variables, fix a field $\mathcal{F}$ of size $|\mathcal{F}| = O(\log^3 n)$. Define $m \equiv \log n / \log\log n$, $m' \equiv 3m + 3$, $h \equiv \log n$, and $d \equiv hm' + m' + 2$. Then, there is a sequence of $poly(n)$ tests of the form

$$(\underline{x_1}, \ldots, \underline{x_c}, T)$$

where $\underline{x_i} \in \mathcal{F}^{m'+1}$, $T : \mathcal{F}^c \to \{0,1\}$, and $c = O(poly(\log n))$, such that:

- *If ϕ is satisfiable, then there exists a polynomial $p : \mathcal{F}^{m'+1} \to \mathcal{F}$ of degree d such that $T(p(\underline{x_1}), \ldots, p(\underline{x_c})) = 1$ for every test.*

- *If ϕ is not satisfiable, then for all polynomials $p : \mathcal{F}^{m'+1} \to \mathcal{F}$ of degree d, for at least $1/2$ of the tests, $T(p(\underline{x_1}), \ldots, p(\underline{x_c})) = 0$.*

22.2 Another Low-Degree Test

Our goal is to test whether a function for which we have oracle access, $f : \mathcal{F}^{m'+1} \to \mathcal{F}$, is a low-degree polynomial. In addition to the oracle for f, we assume we are given a parameter d (the degree for which we are testing) as well as oracle access to a function $L : (\mathcal{F}^{m'+1} \times \mathcal{F}^{m'+1}) \to \mathcal{F}^{d+1}$. The claim to be tested is that:

1. f is a degree d polynomial

2. For all $\underline{x}, \underline{y} \in \mathcal{F}^{m'+1}$, $L(\underline{x}, \underline{y})$ is the description of the degree d univariate polynomial q such that $q(t) = f(\underline{x} + t\underline{y})$ for every $t \in \mathcal{F}$.

Our test is as follows:

pick at random $\underline{x}, \underline{y} \in \mathcal{F}^{m'+1}$ and $t \in \mathcal{F}$
$q \leftarrow L(\underline{x}, \underline{y})$
reject if $f(\underline{x} + t\underline{y}) \neq q(t)$
accept

We claim that there exists a constant δ_0 such that if the preceding test accepts (f, L) with probability $1 - \delta$ where $\delta < \delta_0$, then f has agreement at least $1 - 2\delta$ with some degree d polynomial.

22.3 Curves in $\mathcal{F}^m$

Definition 33 *A curve in $\mathcal{F}^m$ is a function $C : \mathcal{F} \to \mathcal{F}^m$. We say C is a curve of degree d if there are m polynomials of degree d, $\{c_1, c_2, \ldots, c_m\}$, such that $C(t) = (c_1(t), c_2(t), \ldots, c_m(t))$.*

For example, $C(t) = \underline{x} + t\underline{y}$ is a curve of degree 1. We note the following easily verified facts about curves.

1. For any $d+1$ points $\underline{x_0}, \ldots, \underline{x_d} \in \mathcal{F}^m$, and for any $t_0, \ldots, t_d \in \mathcal{F}$, there is a curve C of degree d such that $C(t_i) = \underline{x_i}$ for all i. This follows from component-wise polynomial interpolation.

2. For any d points $\underline{x_1}, \ldots, \underline{x_d} \in \mathcal{F}^m$, fix $t_0, \ldots, t_d \in \mathcal{F}$. Pick at random $\underline{x_0} \in \mathcal{F}^m$ and construct C of degree d such that $C(t_i) = \underline{x_i}$ for $i = 0, 1, \ldots, d$. Then $C(t)$ is uniformly distributed in $\mathcal{F}^m$ for every $t \notin \{t_1, \ldots, t_d\}$.

   Why is this true? That coordinate i of $C(t)$ is uniformly distributed in $\mathcal{F}$ follows from the facts that there is a bijection between coordinate i of $C(t_0) = \underline{x_0}$ and coordinate i of $C(t)$ for any $t \notin \{t_1, \ldots, t_d\}$. (The latter point in turn follows from the fact that a polynomial of degree d is uniquely specified by $d+1$ points.) Finally, since the coordinates of $\underline{x_0}$ are chosen independently, it follows that $C(t)$ is uniformly distributed over $\mathcal{F}^m$.

3. If $p : \mathcal{F}^m \to \mathcal{F}$ is a degree d polynomial and $C : \mathcal{F} \to \mathcal{F}^m$ is a degree c curve, then $p(C(t))$ is a degree cd univariate polynomial. This follows from basic facts about polynomial composition.

22.4 Components of the Proof

The prover must provide the following 3 items.

1. A description of the function $f : \mathcal{F}^{m'+1} \to \mathcal{F}$. The claim of the prover is that $f = p$, the degree d polynomial whose existence is asserted by lemma 78 for satisfiable instances ϕ.

2. A description of the function $L : (\mathcal{F}^{m'+1} \times \mathcal{F}^{m'+1}) \to \mathcal{F}^{d+1}$. The claim of the prover is that for all $\underline{x}, \underline{y} \in \mathcal{F}^{m'+1}$, $L(x, y)$ describes the degree d univariate polynomial q satisfying $q(t) = f(\underline{x} + t\underline{y})$.

3. A description of the function $\mathcal{C} : (\mathcal{F}^{m'+1})^{c+1} \to \mathcal{F}^{cd+1}$. For all $\underline{x_0}, \dots, \underline{x_c} \in \mathcal{F}^{m'+1}$, let C be the unique degree c curve $C : \mathcal{F} \to \mathcal{F}^{m'+1}$ such that $C(t_i) = \underline{x_i}$ for all $i \in \{0, \dots, c\}$. Then the claim of the prover is that $\mathcal{C}(\underline{x_0}, \dots, \underline{x_c})$ is the degree cd polynomial equal to $f(C(t))$.

22.5 Verification

The verifier performs the following test whose correctness will be demonstrated in the next lecture.

input: a proof as described above
output: if proof is valid, **accept** with probability 1; otherwise **accept** with probability $\leq \frac{1}{2}$.

pick at random $\underline{x}, \underline{y} \in \mathcal{F}^{m'+1}$ and $t \in \mathcal{F}$
$q \leftarrow L(\underline{x}, \underline{y})$
reject if $f(\underline{x} + t\underline{y}) \neq q(t)$
pick at random a test $(\underline{x_1}, \dots, \underline{x_c}, T)$
pick at random $\underline{x_0} \in \mathcal{F}^{m'+1}$
$q \leftarrow \mathcal{C}(\underline{x_0}, \dots, \underline{x_c})$
compute C such that $C(t_i) = \underline{x_i}$ for all i
pick random $t \in \mathcal{F}$
reject if $f(C(t)) \neq q(t)$
reject if $T(q(t_1), \dots, q(t_c)) \neq 1$
accept

Finally we note that to choose a random element of $\mathcal{F}$ requires $O(\log \log n)$ random bits and hence choosing an element of $\mathcal{F}^{m'+1}$ requires $O(m' \log \log n) = O(\log n)$ random bits. Likewise there are only polynomially many tests to choose from so we need only $O(\log n)$ random bits to make such a choice. Thus the total number of random bits required is $O(\log n)$. Furthermore it is evident that the verification algorithm makes only a constant number of queries to each of $f, L, \mathcal{C}$ and hence only a constant number of queries total are required, although we again note that the total number of bit queries is still $poly(\log n)$.

Chapter 23

Parallelization

April 18,2001, Jittat Fakcharoenphol

In this lecture, we continue the construction of a PCP-verifier that uses $O(\log n)$ random bits and looks at a constant number of places in the proof. We also give a description of the verifier that reads only a constant number of bits but uses polynomial number of random bits.

23.1 Verifier that accesses to a constant number of places

Last time, we introduced the use of curves to encode the evaluations of the polynomial. We review the lemma from the last lecture.

Lemma 79
Given 3SAT formula ϕ, one can construct a sequence of constraints of the form $(x_1, \ldots, x_c, T)$ where
$x_1, \ldots, x_c \in \mathcal{F}$,
$T : \mathcal{F}^c \to \{0, 1\}$,
$|\mathcal{F}| = O(\log^3 n)$,
$m = \log n / \log \log n$,
$m' = 3m + 3$,
$h = \log n$, *and*
$d = 2hm' + m' + 2$,
such that

- *if ϕ is satisfiable, there is a polynomial $p : \mathcal{F}^{m'+1} \to \mathcal{F}$ of degree d such that for every constraint $(x_1, \ldots, x_c, T)$, $T(p(x_1), \ldots, p(x_c)) = 1$.*
- *if ϕ is not satisfiable, for every polynomial $p : \mathcal{F}^{m'+1} \to \mathcal{F}$ of degree d such that for at least half of the constraints $(x_1, \ldots, x_c, T)$, $T(p(x_1), \ldots, p(x_c)) = 0$.*

23.1.1 The proof

The proof consists of 3 parts, namely, f, L, and $Curve$. The polynomial f is the proof, the table L is for checking that if f is close to some polynomial p, and the table $Curve$ is for evaluating p in many points by only looking at a constant number of places. Each of the entries of the proof with the claim from the prover is described below.

- $f : \mathcal{F}^{m'+1} \to \mathcal{F}$.

  Claim: $f \equiv p$ where p is the degree d polynomial that satisfies all constraints.

- $L : (\mathcal{F}^{m'+1} \times \mathcal{F}^{m'+1}) \to \mathcal{F}^{d+1}$.

  Claim: for all $\bar{x}, \bar{y} \in \mathcal{F}^{m'+1}$, $L(\bar{x}, \bar{y})$ describes a degree-d univariate polynomial q such that $q(t) = f(\bar{x} + t\bar{y})$.

- $Curve : (\mathcal{F}^{m'+1})^{c+1} \to \mathcal{F}^{cd+1}$.

  For all $\bar{x}_0, \ldots, \bar{x}_c \in \mathcal{F}^{m'+1}$, let $C : \mathcal{F} \to \mathcal{F}^{m'+1}$ be a curve passing through $x_0, \ldots, x_c$ at $t_0, \ldots, t_c$.

  Claim: $Curve(x_0, \ldots, x_c)$ describes a degree cd univariate polynomial q such that $q(t) = f(C(t))$.

23.1.2 The verifying procedure

The verifier proceeds as follows.

Verifying procedure:
  repeat $O(1)$ times $\{$*for testing if* $f \approx p\}$
    pick random $\bar{x}, \bar{y} \in \mathcal{F}^{m'+1}$ and $t \in \mathcal{F}$.
    let $q = L(\bar{x}, \bar{y})$, reject if $q(t) \neq f(\bar{x} + t\bar{y})$.
  repeat $O(1)$ times $\{$*for retrieving the evaluations of* $p\}$
    pick a random constraint $(x_1, \ldots, x_c, T)$.
    pick a random $x_0 \in \mathcal{F}^{m'+1}$.
    let $q = Curve(x_0, x_1, \ldots, x_c)$.
    let C be a degree-d curve passing through $x_0, \ldots, x_c$.
    pick a random $t \in \mathcal{F} \setminus \{t_1, \ldots, t_c\}$.
    reject if $q(t) \neq f(C(t))$.
    reject if $T(q(t_1), \ldots, q(t_c)) = 0$.
  accept.

Essentially, the step that checks if the formula is satisfiable is the last one that uses the constraint $(x_0, \ldots, x_c, T)$. The other parts of the procedure are for verifying the consistency of the proof and for retrieving the evaluations of the polynomial at c places. Clearly, if the polynomial q in the procedure represents the correct polynomial as described in the first part of the proof, using lemma 79, the test in the last step will correctly verify the proof.

We assume that the proof f passes the low-degree test, described in the last lecture, in the first step; therefore, it is close to some polynomial p. Now, we will prove the correctness of this verifier by proving that the second part of the procedure that outputs the evaluations of the polynomial is correct. The following procedure describes the same procedure used by the verifier.

> pick random $x_0 \in \mathcal{F}^{m'+1}$.
> let $q = Curve(x_0, \ldots, x_c)$.
> let C be a curver of degree c passing through $x_0, \ldots, x_c$.
> pick random $t \in \mathcal{F} \setminus \{t_1, \ldots, t_c\}$.
> reject if $q(t) \neq f(C(t))$.
> output $q(t_1), \ldots, q(t_c)$.

The following lemma states that the procedure outputs the correct evaluations with large-enough probability.

Lemma 80
If $f : \mathcal{F}^{m'+1} \to \mathcal{F}$ has agreement $(1-\delta)$ with a degree-d polynomial $p : \mathcal{F}^{m'+1} \to \mathcal{F}$, then for every $x_1, \ldots, x_c \in \mathcal{F}^{m'+1}$ and for every function $Curve : (\mathcal{F}^{m'+1})^{c+1} \to \mathcal{F}^{cd+1}$, the polynomial evaluation procedure above, conditioned on not rejecting, outputs $p(x_1), \ldots, p(x_c)$ with probability at least $1 - 2\sqrt{\delta} - \frac{cd}{|\mathcal{F}|-c}$.

Proof: We will bound the probability that the procedure accepts and gives the wrong answer. Let $x_1, \ldots, x_c \in \mathcal{F}$ and $t \notin \{t_1, \ldots, t_c\}$ be fixed. Consider C a degree-d curve $C(t_i) = x_i$ for $i = 0, \ldots, c$. We claim without proof that $C(t)$ is uniformly distributed in $\mathcal{F}^{m'+1}$ for $t \notin \{t_1, \ldots, t_c\}$. Therefore,

$$\mathbf{E}_{x_0}[\text{fraction of } t \notin \{t_1, \ldots, t_c\} \text{ such that } f(C(t)) \neq p(C(t))] \leq \delta.$$

Using Markov's Inequality, we have

$$\mathbf{Pr}_{x_0}[(\text{fraction of } t \notin \{t_1, \ldots, t_c\} \text{ such that } f(C(t)) \neq p(C(t))) \geq \sqrt{\delta}] \leq \sqrt{\delta},$$

and hence

$$\mathbf{Pr}_{x_0}[\ f(C(t)) = p(C(t)) \text{ for at least } 1-\sqrt{\delta} \text{ fraction of } t \notin \{t_1, \ldots, t_c\}] \geq 1 - \sqrt{\delta}.$$

Now, consider the case that C is such that $f(C(t)) = p(C(t))$ for $1-\sqrt{\delta}$ fraction of $t \in \mathcal{F} \setminus \{t_1, \ldots, f_c\}$. We note that the polynomial q of degree cd is uniquely specified by $Curve(x_0, \ldots, x_c)$. Because q is of degree cd, either (1) $q(t) = p(C(t))$ or (2) $q(t)$ and $p(C(t))$ agrees on at most cd points. In case (1), the output of the procedure is correct.

Hence, suppose that we are in case (2). We have that $q(\cdot)$ and $p(C(\cdot))$ agree on less than $\frac{cd}{|\mathcal{F}|-c}$ fraction of $\mathcal{F} \setminus \{t_1, \ldots, t_c\}$. Also, $p(\cdot)$ and $f(\cdot)$ agree on $\geq 1-\sqrt{\delta}$ fraction of $\mathcal{F} \setminus \{t_1, \ldots, t_c\}$, i.e., they disagree on less than $\sqrt{\delta}$ fraction. Therefore, $q(\cdot)$ and $f(\cdot)$ agree on less than $\sqrt{\delta} + \frac{cd}{|\mathcal{F}|-c}$ of $\mathcal{F} \setminus \{t_1, \ldots, t_c\}$.

The procedure accepts in case (2) with probability at most $\sqrt{\delta} + \frac{cd}{|\mathcal{F}|-c}$. Taking into account the probability at most $\sqrt{\delta}$ for the case that $f(C(t))$ and $p(C(t))$ has too few agreements, the probability that the procedure accepts and outputs the wrong answer is at most $\sqrt{\delta} + \sqrt{\delta} + \frac{cd}{|\mathcal{F}|-c}$. The lemma follows. □

Note that the lemma above holds for any function $Curve$.

The verifying procedure in this section reads the proof 2 times in the first part and another 2 times in the second part. Therefore, the verifier reads $O(1)$ many places.

From here to $\mathbf{PCP}[O(\log n), O(1)]$... We have to encode the proof again, and verify that the encoding satisfies the same property.

23.2 Verifier that reads a constant number of bits

In this section we will give the first part of the proof that $\mathbf{NP} \subseteq \mathbf{PCP}[poly(n), O(1)]$. We show that we can construct the proof of exponential size and give a verifier that reads only a constant number of bits. The proof of correctness will be in the next lecture.

Our favorite problem, 3SAT, is not suitable for this reduction. We will use another $\mathbf{NP}$-complete problem.

DEFINITION 34 *Given a degree-2 polynomial* $p_1(x_1, \ldots, x_n), \ldots, p_m(x_1, \ldots, x_n)$ *over* Z_2, *the problem* COMMON ROOT *is the problem of deciding if there are* $a_1, \ldots, a_n$ *such that*

$$p_i(a_1, \ldots, a_n) = 0,$$

for $i = 1, \ldots, m$.

The following lemma shows that COMMON ROOT is $\mathbf{NP}$-complete.

LEMMA 81
COMMON ROOT *is* $\mathbf{NP}$*-complete.*

PROOF: Given a 3CNF formula ϕ over $x_1, \ldots, x_n$ with m clauses, we can construct $2m$ polynomials p_i of degree 2 over $x_1, \ldots, x_n, y_1, \ldots, y_m$ such that ϕ is satisfiable iff these polynomials have a common root.

An example for such construction is as follows. For a clause $(x_1 \wedge x_2 \wedge \bar{x_3})$, two polynomials will be created: $y + (1 - x_1)(1 - x_2) - 1$ and $(x_3)(1 - y)$. $\Box$

Suppose there is a that satisfies $p_1, \ldots, p_m$, i.e., $p_j(a_1, \ldots, a_n) = 0$ for all $j = 1, \ldots, m$. We will construct the proof, which is an encoding of a, of exponential size. The proof consists of 2 parts:

- $A : \{0,1\}^n \to \{0,1\}$.

  claim: $A(x_1, \ldots, x_n) = \sum_i a_i x_i \bmod 2$.
- $B : \{0,1\}^{n \cdot n} \to \{0,1\}$.

  claim: $B(z_{11}, \ldots, z_{nn}) = \sum_{i,j} a_i a_j z_{ij} \bmod 2$.

The first part of the proof is all possible linear functions in a, and the second one is all possible degree-2 polynomials in a. This encoding is somewhat an opposite way of encoding polynomials. In polynomial codes, the code words are usually the evaluations of a polynomial on every points in the field. In this encoding, we fix a point an evaluate every polynomial at that point, instead. We note that B is not a linear code of a, i.e., B is of degree-2. However, B itself is a linear function.

The encoding described above is very useful. For example, if the encoding is correct, we can evaluate any degree-2 polynomial $p(y_1, \ldots, y_n) = c_0 + \sum_i c_i y_i + \sum_{i,j} c_{ij} y_i y_j$ by reading at 2 bits of the proof. Therefore, to check that the assignment a satisfies any polynomial p, one needs to verify that $c_0 + A(c_1, \ldots, c_n) + B(c_{11}, \ldots, c_{nn}) = 0$.

Moreover, checking the consistency of these two tables can be done with a constant number of accesses. If A and B are close to linear functions, i.e., $A(x_1, \ldots, x_n) = \sum a_i x_i$ for a large fraction of $x_1, \ldots, x_n$, and $B(z_{11}, \ldots, z_{nn}) = \sum b_{ij} z_{ij}$ for a large fraction of $z_{11}, \ldots, z_{nn}$, then we can check that $b_{ij} = a_i a_j$ for every i, j with $O(1)$ queries. Also, if A and B are not close to linear, we can check that with $O(1)$ queries.

We will present the verifier for this proof in the next lecture.

Chapter 24

NP in PCP[poly(n),1]

April 20, 2001, Scribe: Chris Harrelson

Today we will show that $\mathbf{NP} \subseteq \mathbf{PCP}[O(n^2), 112]$, which will finish the proof of the PCP theorem: $\mathbf{NP} = \mathbf{PCP}[p(n), O(1)]$.

24.1 Picking an NP-complete problem for the reduction

Instead of using 3SAT, we will use a different NP-complete problem for which it is easier to formulate a **PCP** protocol. Here is the problem, called SimultaneousRoots:

> Given m degree-2 polynomials over n variables $x_1, \ldots, x_n$ in $\mathbb{Z}_2$, decide if there are $a_1, \ldots, a_n \in \{0,1\}$ such that
>
> $$\begin{aligned} p_1(a_1, \ldots, a_n) &= 0 \\ &\vdots \\ p_m(a_1, \ldots, a_n) &= 0 \end{aligned}$$
>
> where all operations are mod 2. In other words, find a set of a_i's that are a zero of all of the polynomials.

Claim: this problem is NP-complete.

24.2 The prover-verifier interaction

The proof provided by the prover has two parts, A and B. Part A has length 2^n and part B has length 2^{n^2}. Hence, one can think of A as a function $A : \{0,1\}^n \to \{0,1\}$ which is specified by listing its value for all of the 2^n different inputs. Similarly, one can think of B as a function $B : \{0,1\}^{n^2} \to \{0,1\}$.

The prover claims to the verifier that

$$A(x_1, \ldots, x_n) = \sum_i x_i a_i$$
$$B(x_{11}, \ldots, x_{nn}) = \sum_{i,j} x_{ij} a_i a_j$$

where $a_1, \ldots, a_n$ are a solution to the instance of SimultaneousRoots in question. This is equivalent to saying that

$$A = \{\vec{a} \cdot \vec{x}^T \ \forall \text{ vectors } \vec{x}\}$$
$$B = \{\vec{a} X \vec{a}^T \ \forall \text{ matrices } X\}$$

Here is the algorithm that the verifier will run to verify that this is in fact the case with good probability:

Repeat 7 times:

1. Pick random $\vec{r}, \vec{s} \in \{0,1\}^n$.
   Reject if $A(\vec{r}) + A(\vec{s}) \neq A(\vec{r} + \vec{s})$.
2. Pick random $\vec{r}, \vec{s} \in \{0,1\}^{n^2}4$.
   Reject if $B(\vec{r}) + B(\vec{s}) \neq B(\vec{r} + \vec{s})$.
3. Pick random $\vec{r}, \vec{s} \in \{0,1\}^n$.
   Reject if $\text{SelfCorrect}(A, r) \cdot \text{SelfCorrect}(A, s) \neq \text{SelfCorrect}(B, \ldots, r_i s_j, \ldots)$.
4. Pick random $\vec{r} \in \{0,1\}^n$. Define $p(x_1, \ldots, x_n)$ to be $\sum_i r_i p_i(x_1, \ldots, x_n)$.
   Let $p(x_1, \ldots, x_n)$ be written for convenience as $c_0 + \sum_i c_i x_i + \sum_{i,j} c_{ij} x_i x_j$.
5. Reject if $c_0 + \text{SelfCorrect}(A, c_1, \ldots, c_n) + \text{SelfCorrect}(B, c_{11}, \ldots, c_{nn}) \neq 0$.

Here we use an auxiliary procedure $\text{SelfCorrect}(A, r_1, \ldots, r_n)$ which picks random $y_1, \ldots, y_n$ and returns $A(\vec{y} + \vec{r}) - A(\vec{y})$.

Each iteration of the loop makes 16 bit queries to the proof: 3 for step 1, 3 for step 2, 6 for step 3, and 4 for step 5. Since there are 7 iterations, there will be a total of $16 \cdot 7 = 112$ bit queries, as required.

24.3 Proof that the verifier is not fooled

It remains to be proven that if there exists an $A : \{0,1\}^n \to \{0,1\}$ and a $B : \{0,1\}^{n^2} \to \{0,1\}$ such that the verifier accepts with probability > 0.5 on input $p_1, \ldots, p_m$, then $p_1, \ldots, \mathbf{P}_m$ is a YES instance of the SimultaneousRoots problem.

Assume for the sake of contradiction that there exists such an A and B, and that this is not a YES instance of the SimultaneousRoots problem.

If the verifier accepts after running the whole procedure with probability ≥ 0.5, then it accepts each round with probability at least $0.5^{1/7} > 0.9$. Hence we will assume that it accepts on one iteration with at least this probability in the following lemmas and claims.

CLAIM 82
There is an $\vec{a} \in \{0,1\}^n$ *such that* A *has agreement of* > 0.9 *with the function* $\tilde{A}(x_1, \dots, x_n) = \sum_i a_i x_i$.

LEMMA 83
(Linearity test): Let $f : \{0,1\}^n \to \{0,1\}$. *If* $\mathbf{Pr}_{\vec{x},\vec{y}}[f(\vec{x}) + f(\vec{y}) = f(\vec{x} + \vec{y})] = \alpha$, *then there is an* $\vec{a}$ *such that* f *has agreement at least* α *with the function* $\tilde{f}(x) = \sum_i a_i x_i$.

This lemma was proved in a weaker version by Blum, Luby and Rubinfeld, which was in turn strengthened to this version by Bellare, Håstad, Kiwi and Sudan. Claim 82 follows easily from the lemma and the assumption that step 1 accepts with probability > 0.9.

CLAIM 84
There is a $\vec{b} \in \{0,1\}^{n^2}$ *such that* B *has agreement* > 0.9 *with the function* $\tilde{B}(x_{11}, \dots, x_{nn}) = \sum_{i,j} b_{ij} x_{ij}$.

This follows from the same lemma and step 2's acceptance with probability > 0.9.

CLAIM 85
For every $\vec{r} \in \{0,1\}^n$, *SelfCorrect*$(A, r) = \tilde{A}(\vec{r})$ *with probability at least 0.8, and the same is true for* B.

LEMMA 86
(Self-correction of linear functions): If $f : \{0,1\}^n \to \{0,1\}$ *as agreement at least* $1 - \delta$ *with a linear function* $\tilde{f}$, *then fore every* $\vec{x} \in \{0,1\}^n$ *the following experiment (i.e. SelfCorrect) gives* $\tilde{f}(\vec{x})$ *with probability at least* $1 - 2\delta$*: pick a random* $\vec{y} in \{0,1\}^n$, *and return* $f(\vec{x} + \vec{y}) - f(\vec{y})$.

PROOF: (of Lemma) The first call to f in $f(\vec{x} + \vec{y}) - f(\vec{y})$ is correct with probability at least $1 - \delta$, and so is the second. Hence both are correct with probability at least $1 - 2\delta$. □

Setting $\delta = 0.9$ yields the claim.

CLAIM 87
If step 3 does not reject, then $b_{ij} = a_i a_j$ *for every* i, j.

PROOF: (of Claim 87) Suppose not. Define $M_1 = (b_{ij})$ and $M_2 = (a_i \cdot a_j)$. Then $M_1 \neq M_2$.

LEMMA 88
If $M_1 \neq M_2 \in \{0,1\}^{n^2}$, *then by picking random* $\vec{r}, \vec{s} \in \{0,1\}^n$, $\vec{r} M_1 \vec{s}^T \neq \vec{r} M_2 \vec{s}^T$ *with probability at least* $1/4$.

PROOF: (of Lemma) For any $\vec{m_1}, \vec{m_2} \in \{0,1\}^n$ with $\vec{m_1} \neq \vec{m_2}$, $\mathbf{Pr}[\vec{r} \cdot \vec{m_1}^T \neq \vec{r} \cdot \vec{m_2}^T] = 1/2$. This is true because $\sum_i r_i (m_1)_i - \sum_i r_i (m_2)^i = \sum_{i:(m_1)_i \neq (m_2)_i} r_i$, which is 1 with probability 1/2.

This means that $\mathbf{Pr}[\vec{r} M_1 \neq \vec{r} M_2] \geq 1/2$, since M_1 and M_2 must differ in at least one row if they are different. We can now apply the argument above a second time, but this time with the vectors $(\vec{r} M_1) \cdot \vec{y}^T$ and $(\vec{r} M_2) \cdot \vec{y}^T$. Since this second test is independent of the first, their probabilities multiply, giving the desired result. □

Now, with probability at least $1/4$, the $\vec{r}, \vec{s}$ chosen in step 3 are such that $\sum_{i,j} r_i s_j a_i a_j \neq \sum_{i,j} r_i s_j b_i j$. For each such $\vec{r}, \vec{s}$, the test rejects with probability at least $1 - 3(1-0.8) = 0.4$ (all of the calls to SelfCorrect must fail, and there are three of them). Then in total, step 3 rejects with probability at least $1/4(0.4) = 0.1$. This is a contradiction with the assumption of the proof and the fact that if it accepts, it must do it with probability > 0.9. □

Claim 89
If $\vec{a}$ is not a zero of $p_1, \ldots, p_m$, then $\vec{a}$ is not a zero of p with probability at least $1/2$.

Proof: $p(a_1, \ldots, a_n) = \sum_i r_i p_i(a_1, \ldots, a_n)$, which is 0 or 1 with equal probability if at least one of the $p_i(a_1, \ldots, a_n)$ is not zero. □

Claim 90
If $p(a_1, \ldots, a_n) = 1$ then step 5 rejects with probability at least 0.64.

Proof: Recall that $\tilde{A}(c_1, \ldots, c_n) = \sum_i a_i c_i$ and $\tilde{B}(c_{11}, \ldots, c_{nn}) = \sum_{ij} a_{ij} c_{ij}$. Then with probability at least 0.8^2 (two calls to SelfCorrect; see lemma 86), $c_0 + \sum_i a_i c_i + \sum_{ij} a_{ij} c_{ij} = 1$, since the left hand side of this equation is equal to $p(a_1, \ldots, a_n)$ from step 4, which is equal to 1 by assumption from step 4. □

This means that steps 4 and 5 together fail with probability at $0.64 \cdot 1/2 = 0.32$, which is a contradiction with the assumption that the whole iteration rejects with probability < 0.1. This finishes the proof that this protocol is correct.

Chapter 25

Pseudorandomness and Derandomization

April 23, 2001, Allison Coates Today, we like to come back to

a topic we discussed before: pseudo-random generators. Before, we saw their equivalence to 1-way functions, and their power in cryptographic schemes. Here, we are interested in using pseudo-random generators to deterministically simulate probabilistic algorithms.

DEFINITION 35 *A deterministic procedure is a pseudo random generator G if the following holds: $G : \{0,1\}^t \rightarrow \{0,1\}^m, m > t$, is an (s, ϵ)- pseudo random generator if for every circuit C of size $\leq S$,*

$$|\mathbf{Pr} r C(r) = 1 - \mathbf{Pr} x C(G(x)) = 1| \leq \epsilon$$

Suppose we have a probabilistic polynomial-time algorithm, A. That is, A runs in time n^c on input of length n, and gives the right answer with probability $> 3/4$. Suppose for every n we have a generator mapping n to $G_n : \{0,1\}^{T(n)} \rightarrow \{0,1\}^{n^c}$ which is $(n^c, 1/10)$ pseudo random computable in time $T(n)$. Using the fact that every algorithm that runs in time T can be simulated by a circuit of size T^2,then for every fixed x, the same computation can be simulated by a circuit C_A that outputs the right answer for x with probability $\geq 3/4 - 1/10 > 1/2$, where A given x and gives out the right answer with probability $3/4$. Therefore, A and C_A have agreement within $1/10$.

As long as the pseudo random generator's probability is $\alpha/2$, where α is the probability by which the algorithm A is bounded away from $1/2$ then we can replace the pseudo random generator with random string and get probability $> 1/2$. Hence, for every x, $\mathbf{Pr} A(x), G(s) =$ right answer for $x > 1/2$.

The right answer for x can be found in time $2^{t(n)}(T(n) + n^c)$ be enumerating all possible random choices, where T is the time it takes to compute the generator. However, this might provide no speed-up.

In cryptography, we wanted the generator to be computable in polynomial time, and we wanted it to be secure against polynomial adversaries (recall that (s, ϵ) were super-polynomial). But for simulations of probabilistic algorithm we are not interested in these adversaries. The generator does not need to be good for all polynomials, but only for a fixed polynomial–that is, we are interested in the 2^n possible circuits resulting from hard-coding x.

Further, we simply need the running time of the generator to be no longer than exponential in length of input of generator: if $T(n) = 2^{l(t(n))}, t(n) > c \log n$ then the right answer can be found in $\leq 2^{O(t(n))}$.

Creating an efficient generator of this type (running in no longer than exponential time in length of input) is much easier to do than finding pseudo random generators in the cryptographic schemes we encountered before. Further, such generators are not equivalent to 1-way functions. Results relate to the worst-case bounds, and not to the average case bounds.

We now define an average case complexity for circuit complexity.

DEFINITION 36 *Let $f : \{0,1\}^l \to \{0,1\}$. The circuit complexity $CC(f) = \min\{S :$ there is a circuit C of size S such that C computes $f\}$.*

We define a notion hardness $H(f)$, as we defined hard-core predicates:

DEFINITION 37 $H(f) = \min\{S :$ *there is a circuit* C *of size* $\leq S$ *such that* $\mathbf{Pr} f(x) = C(x) \geq 1/2 + 1/S\}$.

We say that the hardness is $H(f) = h$ if given a circuit of size $< h$ can predict f on no more than a small fraction $+ 1/2$ of the inputs: $\mathbf{Pr} x C(x) = f(x) \leq 1/2 + \frac{1}{h-1}$. Further, we could say f is (S, ϵ) hard if for every circuit of size $\leq S$, $\mathbf{Pr} x C(x) = f(x) \leq 1/2 + \epsilon$; then $h(f) = \max\{S : f$ is $(S, 1/S)$-hard$\}$.

When we discussed circuit complexity, we said that there exists $f : \{0,1\}^l \to \{0,1\}$ such that $CC(f) \geq 2^{l(1-o(1))}$, where CC is the circuit complexity and f is $\Omega(2^l/\epsilon)$. Using Chernoff bounds, it is not hard to show that hardness of f is $H(f) \geq 2^{l(1/3-o(n))}$, where $f : \{0,1\}^l \to \{0,1\}$.

CLAIM 91
For every $f : \{0,1\}^l \to \{0,1\}$ there is a circuit of size $O(2^{l/2})$ that computes f on a fraction $> 1/2 + 1/2^{l/2}$ of the inputs.

PROOF: On input x:

- Use the same $l/2$ bits as a subset
- Answer correctly on them.
- Guess most likely answer on the rest.

If $x = x : o^{l/2}$? then we output the right answer, and then output $f(x)$. Otherwise, we output the answer that is correct for majority of remaining $2^l - 2^{l/2}$ inputs. Then, the probability that the circuit computes f correctly is

$$\begin{aligned} \mathbf{Pr} x C(x) = 1 &\geq \frac{1}{2^{l/2}} + (1 - \frac{1}{2^{l/2}})\frac{1}{2} \\ &= 1/2 + \frac{1}{2^{l/2-1}} \end{aligned}$$

□

In fact, for every $f : \{0,1\}^l \to \{0,1\}$ there is a circuit C of size $2^{l(1/3+o(n))}$ that computes a function close to f: $\mathbf{Pr}x C(x) = f(x) \geq 1/2 + \frac{1}{2^{l(1/3+O(1))}}$. One example is that of a random hash function: $h : \{0,1\}^l \to \{0,1\}^{l/3}$. Let $y \in \{0,1\}^{l/3}$, and it stores the majority $\{f(x) : h(x) = y\}$. That is, for every string y, we look at the preimages, and determine is we see more often a 1 or a 0.

In the next four lectures, we will see the result of Nisan-Wigderson: (1988):

Theorem 92
Suppose some language is solvable in deterministic time $L \in DTIME(2^{O(n)})$ and $\epsilon > 0$, such that for every large enough hn, $H(L_n) \geq 2^{\epsilon n}$, for ϵ bounded away from 0. Let L_n be the restriction of L to inputs of length n: i.e. L_n is a machine that given an input of length n decides if x is in the language. Then there is a family of pseudo random generators $G_m : \{0,1\}^t(n) \to \{0,1\}^n$ computable in poly(m) time, with $t(m) = O(\log m)$ that are $(m^2, 1/m)$ pseudo random and further, $\mathbf{P} = \mathbf{BPP}$.

We will also see the result of Impagliazzo-Wigderson (1997):

Theorem 93
Suppose there is $L \in DTIME(2^{O(n)})$ and $\epsilon > 0$, such that $CC(L_n) \geq 2^{\epsilon n}$ for all sufficiently large n. Then there is $L' \in DTIME(2^{O(n)}), \epsilon' > 0$, such that $H(L_n) > 2^{\epsilon' n}$ for all sufficiently large n and $\mathbf{P} = \mathbf{BPP}$.

So either circuits can get more than polynomial speed up over deterministic algorithms, or probabilistic algorithms do not given more than polynomial speed up over deterministic algorithms. This ties the average-case and worst-case complexity measures together:

Suppose there is $L \in DTIME(2^{o(n)})$, such that for all C and sufficiently large n, $CC(L_n \geq n^c$, then $\mathbf{BPP} = \bigcap DTIME(2^{n^\epsilon})$. This conjecture follows for certain restrictions of the previous two theorems. However, this type of argument uses a non-uniform reduction. Suppose for every problem there exists some hard problem, say in **BPP**, then the easy is in exptime. And, if exptime is hard, then **BPP** easy. So this is an interesting type of reduction. Ultimately, if the problem started from is hard, then the generator is hard to break. This is usually shown by the contrapositive: if we can break the generator, then we can solve the hard problem. Another result of Impagliazzo-Wigderson (1998) does the above in uniform way. If $\mathbf{BPP} \neq \exp$, then for every **BPP** problem L, and $\epsilon > 0$, there is an algorithm that runs in time 2^{n^ϵ} (it does not really solve, but solves on almost an inputs of length n. That is, for infinitely many n, $\mathbf{Pr}x A(x) = \text{rightanswer} > 1 - 1/n$.

Finally, there is a way of seeingAM as a version of **NP**. A pseudo random generator can give **NP** simulations of AM. This result by Arvin Kobler, Milterson, and ...

Theorem 94
We define deterministic circuit complexity (N-CC) as the size of the smallest non-deterministic circuit for that function. Suppose there is $L \in NTIME(2^{O(n)})$ and $\epsilon > 0$ such that the deterministic circuit complexity $N - CC(L_n) \geq 2^{\epsilon n}$ for sufficiently large n, the $AM \subset$ $\mathbf{NP}$.

Chapter 26

Nisan-Wigderson

April 25, 2001, Scribe: Kunal Talwar In this lecture we will look

at the Nisan-Wigderson construction of pseudorandom generators using a language L that has a $2^{O(n)}$ time algorithm but no circuit family of size less than $2^{\epsilon n}$ can identify strings in L with a significant advantage. The construction uses subset families with some desired properties, and the proof uses techniques used in the Blum-Micali construction discussed in lecture 13.

26.1 Notation

If L is a language, then for any n, L_n is the restriction of L to inputs of length n, viewed as a boolean function :

$$L_n : \{0,1\}^n \to \{0,1\}$$

$$L_n(x) = 1 \text{iff} x \in L$$

For a function $f : \{0,1\}^n \to \{0,1\}$, the *hardness* of f, $H(f)$ is the largest value of S such that for every circuit C of size at most S,

$$\mathbf{Pr}_x[C(x) = f(x)] \leq \frac{1}{2} + \frac{1}{S}$$

26.2 The main result

Theorem 95

[**Nisan Wigderson 1988**] *Suppose there is $L \in DTIME(2^{O(n)})$ and an $\epsilon > 0$ such that for all sufficiently large n, $H(L_n) \geq 2^{\epsilon n}$. Then there is a family of pseudorandom generator $G_n : \{0,1\}^{t(m)} \to \{0,1\}^m$ computable in time polynomial in m with $t(m) = O(\log m)$. Further, then,* $\mathbf{P} = \mathbf{BPP}$.

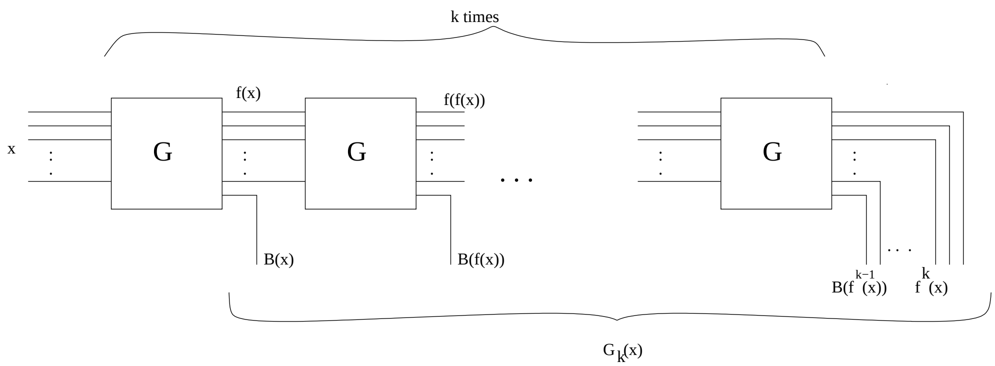

Figure 26.1: The Blum Micali construction

The construction of the generator will be based on inputs of L of logarithmic length. We will choose the length of the inputs so that computing L_n can be done in polytime, but the smallest circuit that can compute L_n with a non-negligible advantage is of size $\omega(n^2)$. Set $l = \frac{2\log m}{\epsilon}$. Then

- L on inputs of length l can be solved in time $2^{O(l)} = m^{O(\frac{1}{\epsilon})}$.
- $L_l : \{0,1\}^l \to \{0,1\}$ has hardness $2^{\epsilon l} = m^2$.

One possible approach could be the following. Consider $G : \{0,1\}^l \to \{0,1\}^{l+1}$ defined as $G(x) = x \cdot f(x)$ where $\cdot$ denotes concatenation. Since f is hard, G is a $(m^2, \frac{1}{m})$-pseudorandom generator. We could look at f as a hard core predicate for the identity permutation and mimic the Blum-Micali construction we saw in the cryptographic setting (See figure 26.1). However in the earlier proof, we did a bootstrap construction that worked because a permutation is hard to invert. Thus this approach cannot directly work here.
Another way to look at the Blum-Micali construction is as follows. From the seed x, we derived a sequence of strings $x, \pi(x), \pi(\pi(x)), \ldots$. Then we apply f to each of these strings to get $f(x), f(\pi(x)), f(\pi(\pi(x))), \ldots$. Similarly in this construction, from a seed s, we will generate a sequence $x_1, x_2, \ldots, x_m$, and output $f(x_1), f(x_2), \ldots, f(x_m)$. What is different in this case is the way of generating x_i's. We construct the x_i's to be independent in a limited manner so that given some values of f, it is difficult to predict the next.

26.3 Interlude: Combinatorial Design

To get the x_i's required above, we use a *Combinatorial design.* A *Combinatorial design* is a family of subsets $S_1, S_2, \ldots, S_m$ of $\{1, 2, \ldots, t\}$ such that

$$|S_i| = l \qquad \forall i$$
$$|S_i \cap S_j| \le \log m \qquad \forall i \neq j$$

Lemma 96

For every l and for every $\gamma < 1$, there is a family $S_1, S_2, \ldots, S_m \subseteq \{1, 2, \ldots, t\}$ where, $t = O(\frac{l}{\gamma}), m = 2^{\gamma l}, |S_i| = l, and |S_i \cap S_j| \leq \log m$. Further, the family can be computed in $2^{O(t)}$.

Proof: We describe a simple algorithm for the above. We divide the interval $[1, t]$ into l intervals of length $O(\frac{1}{\gamma})$. The algorithm basically picks the sets so that each set contains an arbitrary element from each interval. Let k be a constant to be determined later.

Input: l, γ
Output: Family $S_1, S_2, \ldots, S_m$ satisfying the above constraints.

> $t := k\frac{l}{\gamma}$
> Divide $\{1, 2, \ldots, t\}$ into l intervals of equal size.
> Pick $S_1 \subseteq \{1, 2, \ldots, t\}$ arbitrarily such that there is exactly one element from each interval.
> **for** $i := 2$ to m **do**
>   Pick $S_i \subseteq \{1, 2, \ldots, t\}$ arbitrarily such that there is exactly one element from each interval, and further $|S_i \cap S_j| \leq \log m$ for $j = 1, 2, \ldots, i-1$.
> Output $S_1, S_2, \ldots, S_m$.

This is a greedy algorithm. If it reaches the end without failing, we would be done. To prove that the algorithm works, it suffices to prove that
Claim:
Let $S_1, S_2, \ldots, S_i$ for $i \leq m-1$ be subsets of $\{1, 2, \ldots, t\}$ with one element per interval, where $t = k\frac{l}{\gamma}$. Then there is $S \subseteq \{1, 2, \ldots, t\}$ containing one element per interval so that $|S \cap S_j| \leq \log m$ for all $j = 1, 2, \ldots, i$.
Proof: We show the existence of such a set S by showing a random experiment that gives such a set with non-zero probability. Construct S by choosing, for each interval, one element uniformly at random from the interval. Now for a fixed j, consider the random variable $|S \cap S_j|$. This is the sum of l independent 0/1 random variables each with expectation $\frac{l}{t}$. Thus

$$E[|S \cap S_j|] = l\frac{l}{t} = \frac{\gamma l}{k} = \frac{\log m}{k}$$

Thus for an appropriate k the probability that $|S \cap S_j| \geq \log m$ can be made strictly smaller than $\frac{1}{m}$ (Chernoff bounds). Thus with a non-zero probability, $|S \cap S_j| \geq \log m$ for all j. $\Box$
This shows that the algorithm works.

Further, we can verify that the algorithm has the required time complexity. $\Box$

26.4 Construction

The combinatorial design above is used with the following parameters. We set $l = \frac{2\log m}{\epsilon}$ and $t = O(\frac{\log m}{\epsilon^2})$ (with $\gamma = \frac{\epsilon}{2}$). The construction of the combinatorial design then works in time $m^{O(\frac{1}{\epsilon^2})}$.

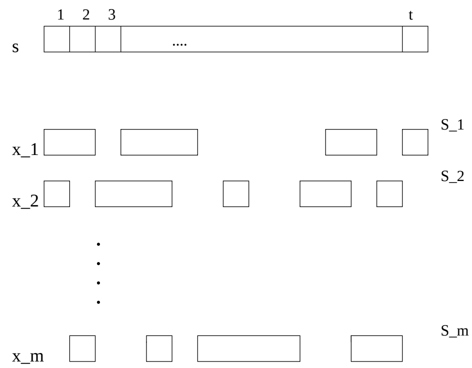

Figure 26.2: Getting x_i's from seed s and sets from a combinatorial design as masks

This combinatorial design gives us a family of sets with a small intersection. For a string $s \in \{0,1\}^t$ and $S_i \subseteq \{1, 2, \ldots, t\}, |S_i| = l$, define $s|_{S_i}$ to be the string of length l obtained by concatenating the bits in s indexed by S_i. We take $x_i = s|_{S_i}$ in our construction. Thus S_i's give us masks for the seed s(see figure 26.2). More formally, the generator is defined as

$$NW_{f,(S_1,S_2,\ldots,S_m)} : \{0,1\}^t \to \{0,1\}^m$$

$$NW_{f,(S_1,S_2,\ldots,S_m)}(s) = f(s|_{S_1})f(s|_{S_2})\ldots f(s|_{S_m})$$

Note that for $i \neq j$, $f(s|_{S_i})$ and $f(s|_{S_j})$ are almost independent in the sense that a fraction $(1-\epsilon)$ of the inputs to f are independent. This novel notion of limited independence helps us in proving that this construction gives a pseudorandom generator.

26.5 Proof

We now show that the construction is a pseudorandom generator. We use proof by contradiction. We show the following generalization of the contrapositive of theorem 95.

Theorem 97
Suppose that there is a circuit C of size S such that

$$\left|\mathbf{Pr}_z[C(NW_{f,(S_1,S_2,\ldots,S_m)}(z)) = 1] - \mathbf{Pr}_r[C(r) = 1]\right| > \delta$$

then there is a circuit C' of size $S + O(m^2)$ such that

$$\mathbf{Pr}_x[C('x) = f(x)] \geq \frac{1}{2} + \frac{\delta}{m}$$

Proof: Without loss of generality, we can assume that

$$\mathbf{Pr}_z[C(NW_{f,(S_1,S_2,\ldots,S_m)}(z)) = 1] - \mathbf{Pr}_r[C(r) = 1] > \delta \tag{26.1}$$

Now we use a hybrid argument as in the Blum-Micali generator. We define a sequence of distributions such that the first one is the output of the generator, the last one is a random input, and consecutive distributions are close to each other.

$$\begin{array}{lclllllllll}
H_m & = & f(z|_{S_1})f(z|_{S_2})\dots f(z|_{S_i})f(z|_{S_{i+1}}) & \dots f(z|_{S_{m-1}})f(z|_{S_m}) & & & & & = NW(z) \text{ for random } z \\
H_{m-1} & = & f(z|_{S_1})f(z|_{S_2})\dots f(z|_{S_i})f(z|_{S_{i+1}}) & \dots f(z|_{S_{m-1}}) & r_m & & & & \\
\vdots & & & & & & & & \\
H_i & = & f(z|_{S_1})f(z|_{S_2})\dots f(z|_{S_i}) & r_{i+1} & \dots & r_{m-1} & r_m & & \\
H_{i-1} & = & f(z|_{S_1})f(z|_{S_2})\dots \quad r_i & r_{i+1} & \dots & r_{m-1} & r_m & & \\
\vdots & & & & & & & & \\
H_1 & = & f(z|_{S_1}) \quad r_2 \quad \dots \quad r_i & r_{i+1} & \dots & r_{m-1} & r_m & & \\
H_0 & = & r_1 \quad r_2 \quad \dots \quad r_i & r_{i+1} & \dots & r_{m-1} & r_m & & = \text{ random}
\end{array}$$

So (26.1) says that

$$\mathbf{Pr}[C(H_m) = 1] - \mathbf{Pr}[C(H_0) = 1] > \delta$$

Therefore, there exists an i such that

$$\mathbf{Pr}[C(H_i) = 1] - \mathbf{Pr}[C(H_{i-1}) = 1] > \frac{\delta}{m} \tag{26.2}$$

Consider a string of length t as divided into x and y of length l and $(t-l)$ respectively, where x represents the bits indexed by S_i and y represents the remaining bits. For $j = 1, 2, \dots, i$, define

$$f_j(x, y) = f(z|_{S_j})$$

where z is a t-bit string defined by the constraints $z|_{S_i} = x$ and $z|_{S_i^c} = y$. In other words, z is obtained by putting x in bits indexed by S_i and y in the other bits. Note that under this definition, $f_i(x, y) = f(x)$.
With this notation, (26.2) becomes

$$\begin{aligned}
&\mathbf{Pr}[C(f_1(x,y)f_2(x,y)\dots f_{i-1}(x,y)f(x)r_{i+1}\dots r_m) = 1] \\
&-\mathbf{Pr}[C(f_1(x,y)f_2(x,y)\dots f_{i-1}(x,y)r_ir_{i+1}\dots r_m) = 1] \quad > \quad \frac{\delta}{m}
\end{aligned}$$

We now define an algorithm that predicts f on an input x with a non-negligible advantage.

Algorithm A
Input: x
Output: $f(x)$

Pick at random $y \in \{0,1\}^{t-l}, b \in \{0,1\}, r_{i+1}, \dots, r_m \in \{0,1\}$
Compute $C(f_1(x,y)f_2(x,y)\dots f_{i-1}(x,y)br_{i+1}\dots r_m)$
if C returns 0
  then return b
  else return $1 - b$

It is easy to see that

$$\mathbf{Pr}_{x,y,b,r_{i+1},\dots,r_m}[A(x) = f(x)] > \frac{1}{2} + \frac{\delta}{m}$$

However, note that in order to compute f once, A needs to compute f on $(i-1)$ inputs. We now fix this. First note that

$$\mathbf{Pr}_{x,y,b,r_{i+1},\dots,r_m}[A(x) = f(x)] = \mathbf{E}_{y,b,r_{i+1},\dots,r_m}[\mathbf{Pr}_x[A(x) = f(x)]]$$

Hence there must be fixed values of $y, b, r_{i+1}, \dots, r_m$ that can be hardwired into a circuit C' for A such that

$$\mathbf{Pr}_x[C'(x) = f(x)] > \frac{1}{2} + \frac{\delta}{m}$$

Note that now with y fixed to a particular value y_0, $f_j(x, y_0)$ is only a function of x. Moreover, it depends on at most $\log m$ bits of x, since $|S_i \cap S_j| \leq \log m$. Since any function with n input bits has a circuit of size 2^n, $f_j(x, y_0)$ can be computed by a circuit of size $2^{\log m} = m$. Thus C' can be implemented using $O(m^2) + \text{size}(C)$ gates. $\square$

Chapter 27

Extractors

April 30, 2001, Scribe: Vinayak Prabhu

We were in the middle of proof for the Nisan-Wigderson construction. We will complete the proof and then begin discussion of Extractors.

27.1 Nisan-Wigderson Construction

Let f be a boolean function $f : \{0,1\}^l \to \{0,1\}$. Let the hardness of f be $H(f) = O(m^2)$, $m = 2^{rl}$ $r > 0$. We define a family of subsets:

$$\begin{array}{l} S_1, \ldots, S_m \subseteq \{1, \ldots, t\}, \qquad t = O(l/r) \\ |S_i| = l \\ |S_i \cap S_j| \leq rl = \log(m) \end{array}$$

Then, the pseudorandom generator $NW_{f(S_1,\ldots,S_m)} : \{0,1\}^t \to \{0,1\}^m$ is given by:

$$NW_{f(S_1,\ldots,S_m)}(z) = f(z|S_1), \ldots, f(z|S_m)$$

Where $z|S = z_{s_1}, z_{s_2}, \ldots, z_{s_l}$, if $S = \{s_1, \ldots, s_l\} \subseteq \{1, \ldots, t\}$. $z|k$ denotes the kth bit in the sequence z.

Intuitively, since f is a hard function, it is easy to compute one pseudorandom bit from the input seed, since f looks random to any small circuit. The problem is to generate more than one random bit. To do this, we try to compute f on m nearly disjoint subsets of bits.

Lemma 98
If $S_1, \ldots, S_m \subseteq \{1, \ldots, t\}, t = O(l/r)$ are such that $|S_i| = l$ and $|S_i \cap S_j| \leq \log(m)$, for every f with $H(f) = O(m^2)$ and every C such that $|\mathbf{Pr}_z[C(NW_{f(S_1,\ldots,S_m)}(z) = 1] - \mathbf{Pr}_r[C(r) = 1]| > \delta$, there is a circuit A of size $\leq O(m^2) + size(C)$ such that $\mathbf{Pr}_x[A(x) = f(x)] \geq 1/2 + \delta/m$

Proof: We present the algorithm A.

Input: $x \in \{0,1\}^l$
Output: $\in \{0,1\}$

Pick at random $y \in \{0,1\}^{t-l}$; $r_i, r_{i+1}, \ldots, r_m \in \{0,1\}$
Compute $C(f_1(x,y), f_2(x,y), \ldots, f_{i-1}(x,y), r_i, \ldots, r_m)$
//Notation: $f_j(x,y) = f(z|S_j)$ where z equals x on bits in S_j and
// z equals y on bits in $\{1,\ldots,t\} - S_j$.
// Essentially we are seperating z into relevant bits (x) and irrelevant bits (y)
// $f_j(x,y) = f(z|S_j) = f(x)$
if $C(\ldots) = 1$ output r_i
else output $1 - r_i$

Lets compute the probability bounds of A. The proof is using the hybrid argument.

$$\begin{aligned}
H_0 &= r_1, r_2, \ldots, r_m \qquad \text{completely random} \\
H_1 &= f(z|S_1), r_2, \ldots, r_m \\
H_{i-1} &= f(z|S_1), \ldots, f(z|S_{i-1}), r_i, \ldots, r_m \\
H_i &= f(z|S_1), \ldots, f(z|S_i), r_{i+1}, \ldots, r_m \\
H_m &= NW(z)
\end{aligned}$$

H_{i-1} and H_i are very close. If $|\mathbf{Pr}[C(H_m) = 1] - \mathbf{Pr}[C(H_0) = 1]| \ > \ \delta$, then we must have that $|\mathbf{Pr}[C(H_i) = 1] - \mathbf{Pr}[C(H_{i-1}) = 1]| \ > \ \delta/m$

So $\mathbf{Pr}_{x,y,r_i,\ldots,m}[A(x) = f(x)] \ > \ 1/2 + \delta/m$

We are not done yet. The probability here is over $x, y, r_i, \ldots, r_m$.

$$\begin{aligned}
\mathbf{Pr}_{x,y,r_i,\ldots,m}[A(x) = f(x)] \ &= \ E_{y,r_i,\ldots,r_m}\left(\mathbf{Pr}_x\left[A(x) = f(x)\right]\right) && \geq 1/2 + \delta \\
\Rightarrow \ \exists \ y, r_i, \ldots, r_m : \ &\mathbf{Pr}_x[A_{y,r_i,\ldots,r_m}(x) = f(x)] && \geq 1/2 + \delta \qquad (27.1)
\end{aligned}$$

So our final algorithm would be a modified version of A. $A_{y,r_i,\ldots,r_m}$ would have the parameters in Eqn. 27.1 built in. Otherwise it would be the same as A.

We now bound the space required by a circuit for $A_{y,r_i,\ldots,r_m}$. If we fix y only $|x| = \log(m)$ bits are needed in the computation of $f_i(x, y)$. So $f_i(x, y)$ can be computed by a circuit of size $O(2^{\log(m)}) = O(m)$. Since we may at most have to compute m functions, $A_{y,r_i,\ldots,r_m}$ has a circuit of size $O(m^2) + \text{size}(C)$.

□

Using this lemma we can proove the following theorem

Theorem 99 (Nisan-Wigderson)

If there is $L \in \mathbf{DTIME}(2^{O(n)})$ such that for all sufficiently large n if $H(L_n) \geq 2^{\epsilon n}$, $\epsilon > 0$, then for every m, there is a $G_m : \{0,1\}^{O(\log(m))} \to \{0,1\}^m$ that is a $(m^2, 1/m)$ pseudorandom generator computable in poly(m) time.

Proof: Given m fix l such that $2^{\epsilon l} \geq O(m^2)$.

$m = 2^{\epsilon l/2 - O(1)}$; $l = \frac{2}{\epsilon}\log(m) + O(1)$

Define $f : \{0,1\}^l \to \{0,1\}$ to be L restricted on inputs of length l.

f is computable in time $2^{O(l)} = m^{O(1/\epsilon)}$

Do NW construction with $r = 2/\epsilon$.

NW: $\{0,1\}^t \to \{0,1\}^m$, $t = O(l/r) = O(\frac{1}{\epsilon^2}\log(m))$

The security bounds follow from the previous lemma.

□

Corollary 100

Under the same assumptions as in the Nisan-Wigderson theorem, if $L \in \mathbf{BPP}(t(n))$ then $L \in \mathbf{DTIME}(t(n)^{O(1/\epsilon^2)})$ and so $\mathbf{P} = \mathbf{BPP}$

27.2 Extractors

Many probabilistic algorithms are designed under the assumption that the input distribution is uniform.

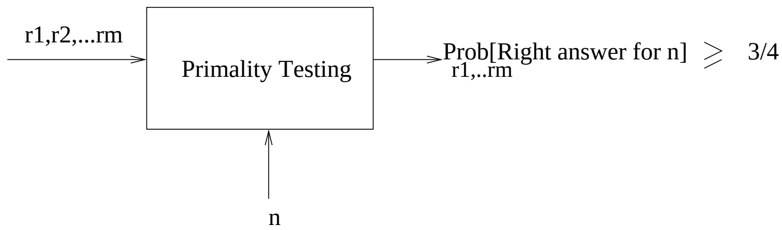

Figure 27.1: Ideal probabilistic algorithm

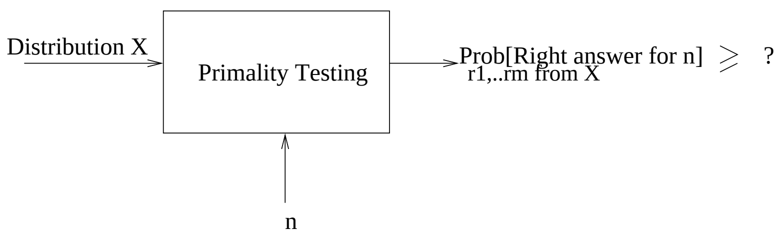

Figure 27.2: Real World

If the input distribution is not uniform, then no guarentees are given. But this might be the case in the real world. We might not know about the process giving the input to the algorithm. We want to transform a biased distribution to a uniform distribution. Its an open question as to which cases can be done.

A Random extractor attempts to make the input distribution uniform

Definition 38 *A distribution D is ϵ-close to uniform if for every $T : \{0,1\}^m \to \{0,1\}$, $|\mathbf{Pr}_{r_1,\ldots,r_m}[T(r_1,\ldots,r_m)] - \mathbf{Pr}[T(z) = 1]| \leq \epsilon$*

Remark 3 If algorithm A succeeds with probability p using a uniform distribution then it succeeds with probability at least $p - \epsilon$ under a distribution that is ϵ-close to uniform.

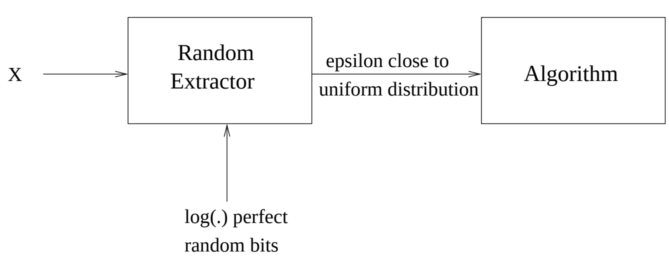

Figure 27.3: Random Extractors

DEFINITION 39 *A distribution D over $\{0,1\}^n$ has min entropy k if for every $z \in \{0,1\}^n$, $\mathbf{Pr}[D = z] \ \leq \ 1/2^k$.*
Min entropy of D $=$ $min_{z\in\{0,1\}^n} \log(1/\mathbf{Pr}[D = z])$

DEFINITION 40 *$E: \ \{0,1\}^m \times \{0,1\}^t \to \{0,1\}^m$ is a (k, ϵ) extractor if for every ditribution D with min entropy at least k, $E(D, z)$ is ϵ-close to uniform where z is uniform in $\{0,1\}^t$.*

REMARK 4 We need some restrictions on the input distribution. The output must contain approximately m random bits. t units of randomness can come from the second argument. So the input distribution should contain $m - t$ units of randomness. Randomness cannot be created out of nothing

REMARK 5 Why did we choose min-entropy? Consider the shannon entropy defined as $H(D) = \sum_z \mathbf{Pr}[D = z] \log(\frac{1}{\mathbf{Pr}[D=z]})$. Now consider a ditribution X, where $X = \{0, 0, \ldots, 0\}$ with probability $1 - 1/\sqrt{n}$ and is uniform in $\{0,1\}^n$ with probability $1/\sqrt{n}$. The shannon entropy of X is approximately $\sqrt{n}$. However X is not usable to produce a uniform distribution. No extractor can generate a uniform distribution as it would have to be $1/\sqrt{n}$ close to uniform when given all $X = 0$.

So it seems if we want to create something uniform, then we need high entropy most of the times. This is approximately equal to having high entropy all the time, which is the notion of min entropy.

Chapter 28

Extractors and Error-Correcting Codes

May 2, 2001, Scribe: Ranjit Jhala In this lecture, we shall look at *randomness extractors*

and see a beautiful connection between them and pseudo-random generators, due to which a PRG such as the Nisan-Wigderson generator may be used to construct extractors. Then we shall see how those ideas may be used to get worst-case to average case reductions.

28.1 Extractors and Error-Correcting Codes

We begin be recalling the definition of extractors. To define them we need a notion called the minimum entropy of a distribution, which is a lower bound on the bits of information contained in a distribution, it is the fewest bits required to specify the probability of a particular element in the distribution.

DEFINITION 41 (**Min Entropy**) *The* Min Entropy *of a distribution* $X = \min_a \log\left(\frac{1}{\mathbf{Pr}[X=a]}\right)$.

The min entropy is a measure of the amount of randomness of a distribution, a distribution with low min entropy is one where the weight is heavily concentrated at a particular element and hence one which contains little randomness.

An extractor is a procedure that when given an arbitrary distribution with high enough min-entropy, and a few completely random bits, will return a distribution which is as close to the uniform as is desired. The notion of closeness of distribution is the well known variation distance, we write it here as:

DEFINITION 42 (**Closeness of Distributions**) *Distributions* X, Y *over some set* S *are* $\epsilon-$*close if for every function* $T : S \to B$, *we have:* $|\mathbf{Pr}_{x\in X}[T(x) = 1] - \mathbf{Pr}_{y\in Y}[T(y) = 1]| \leq \epsilon$.

Finally, we are ready to state the definition of Extractors:

DEFINITION 43 (**Extractor**) *A function* $Ext : \{0,1\}^n \times \{0,1\}^t \to \{0,1\}^m$ *is a* $(k, \epsilon)-$*extractor if for every distribution* X *(over* $\{0,1\}^n$*) of min-entropy* $\geq k$, $Ext(X, S)$ *is* $\epsilon-$*close to being uniform (over* $\{0,1\}^m$*), where* S *is uniform in* $\{0,1\}^t$.

Basic Idea

To connect extractors with PRGs we will use *Error-Correcting Codes*. The basic idea of the connection which we shall see in detail soon is that given a particular element from the distribution, we shall encode that element and then think of the encoded string as a function to be supplied as a seed to the NW generator.

To show that the resulting distribution is close to uniform, we shall fix a(n arbitrary) distinguishing circuit T and argue that the probability of *bad* seeds, which are seeds which cause the generator to give outcomes that are predicted by T is small.

This happens as from the analysis of the NW generator and the property of the ECCs, which ensure that the functions are really very sparsely encoded, it follows that if a seed is bad, then it may be represented by a few number of bits and thus there are few of them. The min-entropy is then used to bound the weight in the distribution of such seeds.

Thus, every distinguishing circuit does well only for a negligibly small probability, and so no circuit can distinguish the output of the extractor from uniform.

We now proceed to the details of Error Correcting codes and see what "sparseness" means, and then to the construction of the extractors and the analysis.

DEFINITION 44 (**Error-Correcting Code**) *A function $C : \{0,1\}^n \to \{0,1\}^{\bar{n}}$ is an* Error-Correcting Code *with minimum distance α if for every $x_1,\ x_2 \in \{0,1\}^n$, the hamming distance between $C(x_1),\ C(x_2)$ is at least $\alpha\bar{n}$.*

The range points of the function C are called the codewords. If we draw balls of radius $\alpha/2$ around the codewords, we get disjoint sets.

There is a well known result in coding theory, that there is no code with $\alpha \geq \frac{1}{2}$, so if more than one fourth of the message bits are corrupted then there is no way that the original word can be uniquely (correctly) reconstructed. However, there is a bound on the maximum number of codewords that are within a certain distance of a particular codeword:

THEOREM 101
Suppose $C : \{0,1\}^n \to \{0,1\}^{\bar{n}}$ has minimum distance $\frac{1}{2} - \delta^2$. Then for every $a \in \{0,1\}^{\bar{n}}$ there are at most $1/(3\delta^2)$ strings $x \in \{0,1\}^n$ such that a and $C(x)$ differ in $\leq (\frac{1}{2} - \delta)\bar{n}$ co-ordinates.

We will also use the result that for every n, δ there is a $C : \{0,1\}^n \to \{0,1\}^{\bar{n}}$ computable in poly$(n, \frac{1}{\delta})$ time with minimum distance $\geq \frac{1}{2} - \delta^2$.

Nisan-Wigderson Revisited

Let us quickly review the important points from the analysis of the NW generator :

1. We assumed there was a function $f : \{0,1\}^l \to \{0,1\}$ satisfying certain complexity requirements.
2. We (randomly) selected $S_1, \ldots, S_m \subseteq \{1, \ldots, t\}$ where $t = O(\frac{l}{\gamma})$ where $m = 2^{\gamma l}$. Also, for each $i \in \{1, ..., m\}$ we had $|S_i| = l$ and for each pair $i, j \in \{1, ..., m\}$ we had $|S_i \cap S_j| \leq \log m$.

3. Finally the generator was defined as:

$$\mathcal{NW}_{f,S_1,\ldots,S_m}(z) = f(z|_{S_1}) \cdots f(z|_{S_m})$$

4. In order to study the security of this generator, we saw that if $T : \{0,1\}^m \to \{0,1\}$ is such that $|\mathbf{Pr}_z[T(\mathcal{NW}(z)) = 1] - \mathbf{Pr}_r[T(r) = 1]| \geq \epsilon$, then there exists a circuit $\widehat{C}$ of size $O(m^2) + \mathbf{SIZE}(T)$, such that $\mathbf{Pr}_x[\widehat{C}(x) = f(x)] \geq \frac{1}{2} + \frac{\epsilon}{m}$.

We shall now see how this generator may be used to construct good extractors.

28.2 Construction of Extractors

Suppose we are given a distribution X over $\{0,1\}^n$ of min-entropy $k = n^\alpha$. Also assume we have a code $C : \{0,1\}^n \to \{0,1\}^{\bar{n}}$ of minimum distance $\frac{1}{2} - \frac{\epsilon^2}{m^2}$, where $m = k^{1/3}$. The extractor is as follows:

1. Sample a from X, and encode a with C to get $C(a) \in \{0,1\}^{\bar{n}}$, where $\bar{n} = \mathrm{poly}(n)$.
2. View $C(a)$ as a function $f : \{0,1\}^l \to \{0,1\}$, where $l = \log \bar{n}$.
3. Pick a random seed $z \in \{0,1\}^t$.
4. Output $\mathcal{NW}_{f,S_1,\ldots,S_m}(z)$.

We will now see that the output of the above procedure is $2\epsilon-$close to uniform. Note firstly that $l = \log \bar{n} = \Theta(\log n) = \Theta(\log m)$ where $m = k^{1/3}$, and k was the min-entropy and equal to n^α.

Recall now that the length of the random seed that $\mathcal{NW}$ takes is $t = O(\frac{l}{\gamma})$, where $\gamma = \log m/l$. As $l = \Theta(\log m) = c \log m$, it follows that $t = O(l/c) = O(\log n)$. Thus, as required for the original purpose of derandomizing, the length of the purely random seed that the extractor requires is indeed logarithmic.

Let us now look at the proof of the claim. We shall see that for any distribution X having the required min-entropy, the output of the extractor is close to uniform.

Lemma 102
For any distribution X of min-entropy $\geq k$, and any $T : \{0,1\}^m \to \{0,1\}$ we have $|\mathbf{Pr}[T(\mathcal{NW}_{C(a),S_1,\ldots,S_m}(z)) = 1] - \mathbf{Pr}[T(r) = 1]| \leq 2\epsilon$, where a is sampled from X, and z and r are uniformly from $\{0,1\}^t$ and $\{0,1\}^m$ respectively.

Proof: We call $a \in \{0,1\}^n$ in the support of X *bad* for T if

$$|\mathbf{Pr}[T(\mathcal{NW}_{C(a),S_1,\ldots,S_m}(z)) = 1] - \mathbf{Pr}[T(r) = 1]| > \epsilon$$

where now we only sample z, r.

We know from the entropy assumption on X that all $a \in \{0,1\}^n$ that are in the support of X have probability at most $1/2^k$. We wish to bound the number of *bad* a's, and putting the two together we shall see that the total weight of bad a's in the distribution X is small.

Suppose that a is bad for T. Let $f : \{0,1\}^l \to \{0,1\}$ be the function corresponding to $C(a)$, and we know that T "breaks" $\mathcal{NW}_f$. Thus, from the (point 4) in the analysis of the NW generator, we know that there exists a function $g : \{0,1\}^l \to \{0,1\}$, such that g may be described by T and a circuit of size $O(m^2)$ and $\mathbf{Pr}[f(x) = g(x)] \geq \frac{1}{2} + \frac{\epsilon}{m}$.

Note now that the function f is "close" to the function g, and moreover these functions are both in the coded space, where the data points are very sparse. Thus, given g, there are in fact very few $b \in \{0,1\}^n$ such that $C(b)$ is close to g, and from the small set of b's that are close (which includes a), we need only to index into the correct one to completely specify a.

Formally, we have that in a code of minimum distance $1/2 - \delta^2$, in a ball of radius $\frac{1}{2} - \delta$ there are at most $1/(3\delta^2)$ other words, where here we have $\delta = \epsilon/m$. Thus, to index into the element of this set of words that corresponds to a we need $\log(\frac{m^2}{3\delta^2})$ bits.

Moreover, any circuit of size $O(m^2)$ can be represented using $O(m^2 \log m)$ bits. Thus, given T, we can completely specify a using at most $O(m^2 \log m + \frac{m^2}{3\delta^2})$ bits of information. Thus, we have that $\sharp$ *bad* $a \leq 2^{O(m^2 \log m)}$. Combining this with the observation about the max probability of elements in the support of X we have $\mathbf{Pr}_a[a \text{ is bad}] \leq 2^{O(m^2 \log m)} . 2^{-k} \leq \epsilon$ as $m = k^{1/3}$. It follows now that:

$$
\begin{aligned}
&|\mathbf{Pr}[T(\mathcal{NW}_{C(a),S_1,\ldots,S_m}(z)) = 1] - \mathbf{Pr}[T(r) = 1]| \\
&\qquad \leq |\mathbf{Pr}_{\text{bad } a}[T(\mathcal{NW}_{C(a),S_1,\ldots,S_m}(z)) = 1] - \mathbf{Pr}[T(r) = 1]| \\
&\qquad\quad + |\mathbf{Pr}_{\text{good } a}[T(\mathcal{NW}_{C(a),S_1,\ldots,S_m}(z)) = 1] - \mathbf{Pr}[T(r) = 1]|
\end{aligned}
$$

As the max contributions of the bad, good a's is at most 1, ϵ respectively

$$
\begin{aligned}
&\leq \epsilon \cdot 1 + (1 - \epsilon) \cdot \epsilon \\
&\leq 2\epsilon - \epsilon^2 \\
&< 2\epsilon
\end{aligned}
$$

□

Putting the procedure and the claim together we get the following theorem:

Theorem 103
For every $k = n^{\Omega(1)}$ there is a poly(n) time computable extractor $Ext : \{0,1\}^n \times \{0,1\}^t \to \{0,1\}^m$, where $t = O(\log n)$, and $m = k^{1/3}$.

Indeed, the above theorem may be generalised to the following existential result:

Theorem 104
For every n, k, ϵ there is a function : $\{0,1\}^n \times \{0,1\}^t \to \{0,1\}^m$ that is a $(k, \epsilon)-$extractor with $t = \log(n - k) + 2\log(1/\epsilon) + O(1)$, $m = k + t - 2\log(1/\epsilon) + O(1) = k + \log(n - k)$.

Open Problem: Is it possible to construct extractors where $t = O(\log n)$ and $m = k$.

28.3 Worst-Case to Average Case Reductions

We will now see how a function with high worst case complexity, may be used to construct functions that have high worst case complexity.

Given $f : \{0,1\}^l \to \{0,1\}$, where the *Circuit Complexity* of f is large say $CC(f) \geq 2^{\epsilon l}$. Let us view f as a string $\langle f \rangle \in \{0,1\}^{2^l}$. Compute $C(\langle f \rangle)$ where C is as before, an Error Correcting Code of minimum distance $1/2 - \delta^2$. View $C(\langle f \rangle)$ as a function $\overline{f} : \{0,1\}^{O(l)} \to \{0,1\}$.

Suppose now there is an algorithm A that computes $\overline{f}$ on a fraction $1/2 + \delta$ of the inputs. Such an algorithm "describes" f upto just $1/(3\delta^2)$ possibilities, and so using it, we can reconstruct f. In particular, given the algorithm A we can precisely define f with little extra overhead. Reasoning again as before, that as the circuit complexity of f was high, it must be that there can be no such succint algorithm A (with a small circuits) as otherwise we would have a small circuit for f. Thus, there is no small circuit that produces the correct result for $\overline{f}$ for even a fraction $1/2 + \delta$ of the inputs and hence $\overline{f}$ has high average case complexity.

Polynomials and Schwartz-Zippel

Indeed we can see now how the properties of polynomials used by the Schwartz-Zippel test for equivalence, can be viewed as a worst case to average case reduction in a manner similar to above, however the reduction is weaker than the one obtained using ECCs.

Say we are given some field $\mathcal{F}$ and polynomials $p : \mathcal{F}^k \to \mathcal{F}$ of degree d and $f : \mathcal{F}^k \to \mathcal{F}$, which has agreement on a fraction $1 - \frac{1}{4d}$ of the points with p. Now, given f we can reconstruct p at *every* point in time $\mathrm{poly}(d, \log \mathcal{F})$.

To see the implications on complexity, say now we have some function $f : \{0,1\}^l \to \{0,1\}$. Pick a field $|\mathcal{F}| = l^3$, pick a subset $H \subseteq F$ such that $|H| = l$. Thus, now $f : H^{\frac{l}{\log l}} \to \{0,1\}$. Define the polynomial $p : \mathcal{F}^{\frac{l}{\log l}} \to \mathcal{F}$ that agrees with f at the points in $H^{\frac{l}{\log l}}$, and of degree $l^2/\log l$.

Note, by doing this, we have effectively encoded f by a bitvector of length 2^l and p a vector of length $2^{O(l)}$.

Suppose there is an algorithm A that computes p in a fraction $1 - 1/l^2$ of the points. Then we know that we can perfectly reconstruct p, in which case, as f has the same values as p on the points in its domain, we have also reconstructed f !

Say f has worst case complexity $= 2^{\epsilon l}$. Then, if S is the size of the circuit for A, then we must have $2^{\epsilon l} < S + \mathrm{poly}(l)$, where the $\mathrm{poly}(l)$ term is the complexity of the reduction. Thus, we get the lower bound $S > 2^{\epsilon l} - \mathrm{poly}(l)$ on the complexity of A.

Thus it is difficult to compute correctly even "most" of the points of p (as opposed to "more than half", the stronger result achieved above) via a reduction from a function which has high worst case complexity.

Chapter 29

Miltersen-Vinodchandran

May 7, 2001, Scribe: Iordanis Kerenidis

In this lecture we are going to discuss a result by Miltersen and Vinodchandran, which is the first derandomization result that does not use the Nissan-Wigderson construction.

THEOREM 105
If there is a language $L \in \mathbf{DTIME}(2^{O(n)})$ such that for every sufficient large n, the non-deterministic circuit complexity of L_n is greater than $2^{\delta n}$, $\delta > 0$, then $\mathbf{P} = \mathbf{BPP}$ and $\mathbf{NP} = \mathbf{AM}$.

The non-deterministic circuit complexity S of a function $f : \{0,1\}^n \to \{0,1\}$ is the size of the smallest non-deterministic circuit that computes f. A non-deterministic circuit has two inputs x, y, where x is the real input, and $C(x)$ accepts iff there exists a y such that $C(x, y) = 1$.

We have already seen the result by Impagliazzo, that $\mathbf{P} = \mathbf{BPP}$ if there is a language with standard complexity $2^{\delta n}$. Here we use the stronger assumption of non-deterministic circuit complexity, but this is the only way we can get the second part of the theorem, namely that $\mathbf{NP} = \mathbf{AM}$. Before going into the proof of the theorem, let us give the following definitions:

DEFINITION 45 *A set $H \subseteq \{0,1\}^n$ is a (S, ϵ)-hitting set if for every circuit C of size S:
If $\mathbf{Pr}_x[C(x) = 1] > \epsilon$ then there exists an $x \in H$: $C(x) = 1$.
This means that a hitting set is a set of strings, such that if the circuit accepts with some probability, then there exists an element of the set, for which the circuit accepts.*

If $G : \{0,1\}^t \to \{0,1\}^n$ is a (S, ϵ) pseudorandom generator then the set $G(s) : s \in \{0,1\}^t$ is a (S, ϵ)-hitting set.

A hitting set generator is an efficient algorithm that given n, outputs a $(S(n), \epsilon(n))$ hitting set in $\{0,1\}^n$.

PROPOSITION 106
Given n, if we can compute in poynomial time a $(n^2, \frac{1}{2})$ hitting set in $\{0,1\}^n$, then $\mathbf{P} = \mathbf{RP}$.

PROOF: Let A be an **RP** algorithm for a language $L \in \mathbf{RP}$. x is an input of length n on which A runs in time t and uses $\leq t$ random bits. Then we know that A can be simulated by a circuit C, so that $A(x,r) = C(r)$ and the size of C is t^2.

$$\text{if} \quad x \in L, \ \mathbf{Pr}_r[C(r) = 1] > \frac{1}{2}$$
$$\text{if} \quad x \notin L, \ \mathbf{Pr}_r[C(r) = 1] = 0$$

By the hypothesis, we can construct a $(t^2, \frac{1}{2})$-hitting set $H \subseteq \{0,1\}^t$ in poly(t) =poly(n) time. Then, we run the circuit on all input in H and we accept iff the circuit accepts for at least one element of H. $\square$

In the same way, we can also prove that $\mathbf{P} = \mathbf{BPP}$.

Now, if we can construct a hitting set which is good against non-deterministic circuits we can derandomize **AM**. In the paper they construct a hitting set generator which is good against non-determinism and therefore they can derandomize $\mathbf{RP}, \mathbf{BPP}, \mathbf{AM}$.

LEMMA 107
Let $L \in \mathbf{RP}$, $\epsilon > 0$, then there is an algorithm A for L such that for every x, A runs in poly$(|x|)$, uses m =poly$(|x|)$ random bits and

$$x \in L \Rightarrow \mathbf{Pr}_r[A(x,r) = 1] \geq 1 - 2^{m^\epsilon - m}$$
$$x \notin L \Rightarrow \mathbf{Pr}_r[A(x,r) = 1] = 0$$

The way to lower down the probability of error to this small amount is by the use of extractors. This resultb also holds for the class AM.

So, according to the previous lemma in order to derandomize the class **RP** it suffices to prove:

PROPOSITION 108
For every ϵ, for evehry c and given m, we can construct in poly(m) time a $(m^c, 1 - 2^{m^\epsilon - m})$-hitting set.

PROOF: Let us start with a language $L \in \mathbf{DTIME}(2^{O(n)})$ and $f : \{0,1\}^l \to \{0,1\}$ a restriction of L to inputs of length l. We can compute f in $2^{O(l)}$ time and the non-deterministic circuit complexity of f is $NCC(f) \geq 2^{\delta n}$.

The proof goes as following: first, we are going to encode the function f with low-degree polynomials and then construct the desired hitting set. If the hitting set cannot be constructed, then this would mean that the function has a small non-deterministic circuit.

- Encoding of f with low-degree polynomials.

  We pick a field $\mathcal{F}$ of size $4 \cdot 2^{\delta l/2c}$ and a subset $I \subseteq \mathcal{F}$ of size $2^{\delta l/2c} = d$. Now, we can view the function f as $f : I^t \to \{0,1\}$, where $t = \frac{2c}{\delta}$. We are going to encode the function f as a polynomial $p : \mathcal{F}^t \to \mathcal{F}$ with degree $|I|$ in each variable.

- Construction of the hitting set.

  Let's consider all parallel-axis lines in $\mathcal{F}^t$ and the set of points

$$l^i(y) = (a_1, a_2, \ldots, a_{i-1}, y, a_{i+1}, \ldots, a_t).$$

This is the set of all points, when we fix all but one coordinates and let the other one range over the field.

Now, the polynomial $p(l^i(y)) = p(a_1, a_2, \ldots, a_{i-1}, y, a_{i+1}, \ldots, a_t)$ is a univariate degree $|I| = d$ polynomial from $\mathcal{F}$ to $\mathcal{F}$, represented by $\mathcal{F} \log \mathcal{F} = m$ bits.

The hitting set contains exactly the restriction of p to each axis-parallel line. This means, that for each restriction we write the values of p as a string and this string belongs to the hitting set. The size of the hitting set is equal to the number of axis-parallel lines, which is $t\mathcal{F}^{t-1} \approx m^{t-1} = \text{poly}(m)$.

The time of the construction is polynomial in m. This is easy to see, since the construction time is the time to compute f on the field for each line. Hence, it is equal to $m^{t-1} \cdot m \cdot 2^{O(l)} = \text{poly}(m)$, since $2^{O(l)} = m^{O(c/\delta)}$.

So, we have constructed in polynomial time a set of strings, which we claim that is a hitting set.

- The constructed set is a hitting set or else there exists a small non-deterministic circuit for f.

Suppose the construction failed. This means that there exists a circuit C of size m^c such that:

  - C accepts all but 2^{m^ϵ} inputs and
  - C accepts no element from our set.

This means that there is a small subset of $\{0,1\}^m$, where our circuit returns 0 and our set lies entirely in this subset. Let us now consider the set

$$P = \{\text{set of all } q : \mathcal{F} \to \mathcal{F} \text{ univariate polynomials of degree } d \text{ such that } C(q) = 0\}.$$

The input of the cirquit is the binary representation of the polynomial. We know that $|P| \leq 2^{m^\epsilon}$. Moreover, we can see that there exists a set $S \subseteq \mathcal{F}, |S| \leq m^\epsilon$ such that

$$\forall q_1, q_2 \in P, q_1 \neq q_2 \ \ \exists x \in S : q_1(x) \neq q_2(x).$$

Now, let us consider the complete set of values of all polynomials in P. As we have said, the degree of the polynomials is $d = \frac{1}{4}\mathcal{F}$, so any pair of polynomials agree on at most $\frac{1}{4}$ of the values. Suppose we pick an entry at random. With probability greater than $\frac{3}{4}$ the q_1, q_2 are different. If we pick now $k = m^\epsilon$ entries then the probability that the two polynomials are not distinguished is

$$\mathbf{Pr}[q_1, q_2 \text{ are not distinguished}] \leq \left(\frac{1}{4}\right)^k = \frac{1}{(2^{m^\epsilon})^2}$$

Since this probability is less than the inverse of the total number of possible pairs, we can definitely distinguish between the polynomials.

- Non-deterministic circuit to compute f.

  We start by hard-wiring the value of the polynomial p on all elements of S^t. If we can compute p in the whole field, then we have the value of f as well. Also, if f is restricted on a line, then this will be a string of our set which is rejected by the circuit.

  The circuit knows the value of p on S^t. Let us consider the 2-dimensional case for the rest of the construction. So, for every $q(x) = p(x, a),\ a \in S$ we know $q(x),\ x \in S$. The polynomial q is a unique degree polynomial such that $C(q) = 0$ and q agrees with the hard-wired values on S. Now, we'll try to compute the value of $p(a_1, a_2),\ a_2 \in S, a_1 \notin S$. We can do that in a non-deterministic way, by guessing a degree d polynomial q and then check that $q(x) = p(x, a_2)$ for every $x \in S$ and $C(q) = 0$. We output $q(a_1)$. This procedure will either reject or give the right value.

  In this way, if we have precomputed p in $S \times S$, then we can compute p in lines. Hence, we can compute the value of p in the whole field.

  In a 3-dimensional case, the computation of $p(a_1, a_2, a_3)$ reduces to computing $p(a_1, a_2, z)$, $\forall z \in S$, which reduces to computing $p(a_1, y, z),\ \forall y, z \in S$ which reduces to knowing $p(x, y, z),\ \forall x, y, z \in S$.

  The complexity of the construction is propotional to S^t. In the worst case we need all parallel lines from all points in S^t. We reduce $S^t \to S^{t-1} \to \ldots \to S$. The computation time is

  $$T = t \cdot S^t + \{\text{interpolation time}\}$$

  The size of the non-deterministic circuit is

  $$\text{Size} = S^t(m^c + |\mathcal{F} \log \mathcal{F}|)$$

  Since $t = \frac{2c}{\delta},\ \ m, \mathcal{F} \approx O(2^{\frac{\delta l}{2c}}),\ |S| = m^\epsilon$, we have

  $$\text{Size} = 2^{\epsilon l} 2^{\frac{\delta l}{2}} \approx 2^{\delta l}, \quad \text{for } \epsilon = \frac{\delta}{2}$$

So, we proved that if the set we had constructed is not a hitting set, then we have a small circuit to compute f. This concludes the proof of the theorem. $\square$

Note that in order to achieve the derandomization, we used a hard worst-case complexity and not hard average-case complexity. Also, we used the same kind of polynomials as in other constructions, but the evaluation was just on axis-parallel lines and not any more sophisticated subsets of inputs.

Bibliography

[ATSWZ97] R. Armoni, A. Ta-Shma, A. Wigderson, and S. Zhou. $SL \subseteq L^{\frac{4}{3}}$. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 230–239, 1997.

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

[Imm88] N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17:935–938, 1988.

[Kar72] R.M. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[Lev73] L. A. Levin. Universal search problems. *Problemi Peredachi Informatsii*, 9:265–266, 1973.

[Lev86] Leonid A. Levin. Average case complete problems. *SIAM Journal on Computing*, 15(1):285–286, February 1986.

[Nis94] N. Nisan. RL $\subseteq$ SC. *Computational Complexity*, 4(1), 1994.

[NSW92] N. Nisan, E. Szemeredi, and A. Wigderson. Undirected connectivity in $O(\log^{1.5} n)$ space. In *Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science*, pages 24–29, 1992.

[Sze88] R. Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26:279–284, 1988.