CS 278 -- Computational Complexity -- Fall 2004

[general info] [lecture notes] [Midterm and Project]

The [midterm solutions]

General Information

Lecturer: Luca Trevisan, luca@eecs, 615 Soda Hall, Tel. 642 8006

Classes are Monday-Wednesday, 10:30-12noon, 320 Soda

Office hours: Mondays, 2-3pm, or by appointment

References: the main reference for the course will be lecture notes. New lecture notes will be distributed after each lecture. Meanwhile, you can also refer to older notes

- Fall 2002 notes (171 pages) [ps] [pdf]
- Spring 2001 notes (148 pages) [ps] [pdf]

It is also good to have a copy of


About this course: Computational Complexity theory looks at the computational resources (time, memory, communication, ...) needed to solve computational problems that we care about, and it is especially concerned with the distinction between "tractable" problems, that we can solve with reasonable amount of resources, and "intractable" problems, that are beyond the power of existing, or conceivable, computers. It also looks at the trade-offs and relationships between different "modes" of computation (what if we use randomness, what if we are happy with approximate, rather than exact, solutions, what if we are happy with a program that works only for most possible inputs, rather than being universally correct, and so on).

This course will roughly be divided into two parts: we will start with "basic" and "classical" material about time, space, P versus NP, polynomial hierarchy and so on, including moderately modern and advanced material, such as the power of randomized algorithm, the complexity of counting problems, the average-case complexity of problems, and interactive protocols. In the second part, we will focus on more research oriented material, to be chosen among PCP and hardness of approximation; circuit, proof complexity, and communication lower bounds; and derandomization, average-case complexity and extractors.

For reasons that are only partially understood, a disproportionate number of the most beautiful results in Complexity theory in the 80s and 90s have been found by Berkeley graduate students. After taking this course, you will hopefully feel upon you the mission of continuing this tradition into...
Notes on Discrete Probability

The following notes cover, mostly without proofs, the basic notions and results of discrete probability. Two important topics (the birthday paradox and universal hash functions) are not covered, and they will be the subject of future notes.

1 Basic Definitions

In cryptography we typically want to prove that an adversary that tries to break a certain protocol has only minuscule (technically, we say “negligible”) probability of succeeding. In order to prove such results, we need some formalism to talk about the probability that certain events happen, and also some techniques to make computations about such probabilities.

In order to model a probabilistic system we are interested in, we have to define a sample space and a probability distribution. The sample space is the set of all possible elementary events, i.e. things that can happen. A probability distribution is a function that assigns a non-negative number to each elementary event, this number being the probability that the event happen. We want probabilities to sum to 1. Formally,

**Definition 1.1** For a finite sample space $\Omega$ and a function $\Pr: \Omega \to \mathbb{R}$, we say that $\Pr$ is a probability distribution if

1. $\Pr(a) \geq 0$ for every $a \in \Omega$;
2. $\sum_{a \in \Omega} \Pr(a) = 1$.

For example, if we want to model a sequence of three coin flips, our sample space will be $\{\text{Head, Tail}\}^3$ (or, equivalently, $\{0, 1\}^3$) and the probability distribution will assign 1/8 to each element of the sample space (since each outcome is equally likely).

If we model an algorithm that first chooses at random a number in the range $1, \ldots, 10^{200}$ and then does some computation, our sample space will be the set $\{1, 2, \ldots, 10^{200}\}$, and each element of the sample space will have probability $1/10^{200}$.

We will always restrict ourselves to finite sample spaces, so we will not remark it each time. Discrete probability is the restriction of probability theory to finite sample spaces. Things are much more complicated when the sample space can be infinite.

An event is a subset $A \subseteq \Omega$ of the sample space. The probability of an event is defined in the intuitive way
Handout 3: Notes on Discrete Probability

\[ \Pr[A] = \sum_{a \in A} \Pr(a) \]

(Conventionally, we set \( \Pr[\emptyset] = 0 \).)

We use square brackets to remind us that now we are considering a different function: while \( \Pr(\cdot) \) is a function whose inputs are elements of the sample space, \( \Pr[\cdot] \) is a function whose inputs are subsets of the sample space.

For example, suppose that we want to ask what is the probability that, when flipping three coins, we get two heads. Then \( \Omega = \{0, 1\}^3 \), \( \Pr(a) = 1/8 \) for every \( a \in \Omega \), we define \( A \) as the subset of \( \{0, 1\}^3 \) containing strings with exactly two 1s, and we ask what is \( \Pr[A] \). As it turns out, \( A \) has 3 elements, that is \( 011, 101, 110 \), and so \( \Pr[A] = 3/8 \). Very often, as in this example, computing the probability of an event reduces to counting the number of elements of a set.

When \( \Pr(\cdot) \) assigns the same value \( 1/|\Omega| \) to all the elements of the sample space, then it is called the uniform distribution over \( \Omega \).

The following distribution arises very often, and it is good to know about it. Consider the situation where you flip \( n \) biased coins, and the probability that each coin turns out head is \( p \) (where \( 0 \leq p \leq 1 \) is some fixed parameter). The outcome of each flip is independent of all the other outcomes. Then the sample space is \( \Omega = \{0, 1\}^n \), identifying heads with 1s; the probability distribution is

\[ \Pr(a) = p^k(1-p)^{n-k} \text{ where } k \text{ is the number of 1s in } a \]

When \( p = 1/2 \) then we have the uniform distribution over \( \{0, 1\}^n \). If \( p = 0 \) then all \( a \) have probability zero, except \( 00 \cdots 0 \), which has probability one. (Similarly if \( p = 1 \).) The other cases are more interesting.

These distributions are called Bernoulli distributions or binomial distributions.

If we have a binomial distribution with parameter \( p \), and we ask what is the probability of the event \( A_k \) that we get a string with \( k \) ones, then such a probability is

\[ \Pr[A_k] = \binom{n}{k} p^k(1-p)^{n-k} \]

2 Random Variables and Expectation

Very often, when studying a probabilistic system (say, a randomized algorithm) we are interested in some values that depend on the elementary event that takes place. For example, when we play dice, we are interested in the probabilistic system where two dice are rolled, and the sample space is \( \{1, 2, \ldots, 6\}^2 \), with the uniform distribution over the 36 elements of the sample space, and we are interested in the sum of the outcomes of the two dice. Similarly,
when we study a randomized algorithm that makes some internal random choices, we are interested in the running time of the algorithm, or in its output. The notion of a random variable gives a tool to formalize questions of this kind.

A random variable $X$ is a function $X : \Omega \rightarrow V$ where $\Omega$ is a sample space and $V$ is some arbitrary set ($V$ is called the range of the random variable). One should think of a random variable as an algorithm that on input an elementary event returns some output. Typically, $V$ will either be a subset of the set of real numbers or of the set of binary strings of a certain length.

Let $\Omega$ be a sample space, $\Pr$ a probability distribution on $\Omega$ and $X$ be a random variable on $\Omega$. If $v$ is in the range of $X$, then the expression $X = v$ denotes an event, namely the event $\{a \in \Omega : X(a) = v\}$, and thus the expression $\Pr[X = v]$ is well defined, and it is something interesting to try to compute.

Let’s look at the example of dice. In that case, $\Omega = \{1, \ldots, 6\}^2$, for every $(a, b) \in \Omega$ we have $\Pr(a, b) = 1/36$. Let us define $X$ as the random variable that associates $a + b$ to an elementary event $(a, b)$. Then the range of $X$ is $\{2, 3, \ldots, 12\}$. For every element of the range we can compute the probability that $X$ take such value. By counting the number of elementary events in each event we get

$$
\Pr[X = 2] = 1/36 \quad \Pr[X = 3] = 2/36 \quad \Pr[X = 4] = 3/36 \\
\Pr[X = 5] = 4/36 \quad \Pr[X = 6] = 5/36 \quad \Pr[X = 7] = 6/36
$$

and the other probabilities can be computed by observing that

$$
\Pr[X = v] = \Pr[X = 14 - v]
$$

It is possible to define more than one random variable over the same sample space, and consider expressions more complicated than equalities.

When the range of a random variable $X$ is a subset of the real numbers (e.g. if $X$ is the running time of an algorithm — in which case the range is even a subset of the integers) then we can define the expectation of $X$. The expectation of a random variable is a number defined as follows.

$$
E[X] = \sum_{v \in V} v \Pr[X = v]
$$

where $V$ is the range of $X$. We can assume without loss of generality that $V$ is finite, so that the expression above is well defined (if it were an infinite series, it could diverge or even be undefined).

Expectations can be understood in terms of betting. Say that I am playing some game where I have a probability $2/3$ of winning, a probability $1/6$ of losing and a probability $1/6$ of a draw. If I win, I win $\$1$; if I lose I lose $\$2$; if there is a draw I do not win or lose anything.

We can model this situation by having a sample space $\{L, D, W\}$ with probabilities defined
as above, and a random variable $X$ that specifies my wins/losses. Specifically $X(L) = -2$, $X(D) = 0$ and $X(W) = 1$. The expectation of $X$ is

$$
\mathbb{E}[X] = \frac{1}{6} \cdot (-2) + \frac{1}{6} \cdot 0 + \frac{2}{3} \cdot 1 = \frac{1}{3}
$$

so if I play this game I “expect” to win $\$ 1/3. The game is more than fair on my side.

When we analyze a randomized algorithm, the running time of the algorithm typically depends on its internal random choices. A complete analysis of the algorithm would be a specification of the running time of the algorithm for each possible sequence of internal choices. This is clearly impractical. If we can at least analyse the expected running time of the algorithm, then this will be just a single value, and it will give useful information about the typical behavior of the algorithm (see Section 4 below).

Here is a very useful property of expectation.

**Theorem 2.1 (Linearity of Expectation)** Let $X$ be a random variable and $a$ be real; then $\mathbb{E}[aX] = a\mathbb{E}[X]$. Let $X_1, \ldots, X_n$ be random variables over the same sample space; then $\mathbb{E}[X_1 + \cdots + X_n] = \mathbb{E}[X_1] + \cdots + \mathbb{E}[X_n]$.

**Example 1** Consider the following question: if we flip a coin $n$ times, what is the expected number of heads? If we try to answer this question without using the linearity of expectation we have to do a lot of work. Define $\Omega = \{0, 1\}^n$ and let $\mathbf{Pr}$ be the uniform distribution; let $X$ be the random variable such that $X(a) =$ the number of 1s in $a \in \Omega$. Then we have, as a special case of Bernoulli distribution, that

$$
\mathbf{Pr}[X = k] = \binom{n}{k} 2^{-n}
$$

In order to compute the average of $X$, we have to compute the sum

$$
\sum_{k=0}^{n} \binom{n}{k} k 2^{-n}
$$

which requires quite a bit of ingenuity. We now show how to solve Expression (1) just to see how much work can be saved by using the linearity of expectation. An inspection of Expression (1) shows that it looks a bit like the expressions that one gets out of the Binomial Theorem, except for the presence of $k$. In fact it looks pretty much like the derivative of an expression coming from the Binomial Theorem (this is a standard trick). Consider $(1/2 + x)^n$ (we have in mind to substitute $x = 1/2$ at some later point), then we have

$$
\left(\frac{1}{2} + x\right)^n = \sum_{k=0}^{n} \binom{n}{k} 2^{-(n-k)} x^k
$$

and then

$$
\frac{d((1/2 + x)^n)}{dx} = \sum_{k=0}^{n} \binom{n}{k} 2^{-(n-k)} k x^{k-1}
$$
but also
\[ \frac{d((1/2 + x)^n)}{dx} = n \left( \frac{1}{2} + x \right)^{n-1} \]
and putting together
\[ \sum_{k=0}^{n} \binom{n}{k} 2^{-(n-k)} k x^{k-1} = n \left( \frac{1}{2} + x \right)^{n-1} . \]

Now we substitute \( x = 1/2 \), and we have
\[ \sum_{k=0}^{n} \binom{n}{k} 2^{-(n-k)} k 2^{-(k-1)} = n . \]
Here we are: dividing by 2 we get
\[ \sum_{k=0}^{n} \binom{n}{k} 2^{-n} = \frac{n}{2} . \]

So much for the definition of average. Here is a better route: we can view \( X \) as the sum of \( n \) random variables \( X_1, \ldots, X_n \), where \( X_i = 1 \) if the \( i \)-th coin flip is 1 and \( X_i = 0 \) otherwise. Clearly, for every \( i \), \( \mathbb{E}[X_i] = \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1 = \frac{1}{2} \), and so
\[ \mathbb{E}[X] = \mathbb{E}[X_1 + \cdots + X_n] = \mathbb{E}[X_1] + \cdots + \mathbb{E}[X_n] = \frac{n}{2} . \]

### 3 Independence

#### 3.1 Conditioning and Mutual Independence

Suppose I toss two coins, without letting you see the outcome, and I tell you that at least one of the coins came up heads, what is the probability that both coin are heads?

In order to answer to this question (I will give it away that the answer is 1/3), one needs some tools to reason about the probability that a certain event holds given (or conditioned on the fact) that a certain other event holds.

Fix a sample space \( \Omega \) and a probability distribution \( \Pr \). Suppose we are given that a certain event \( A \subseteq \Omega \) holds. Then the probability of an elementary event \( a \) given the fact that \( A \) holds (written \( \Pr(a|A) \)) is defined as follows: if \( a \notin A \), then it is impossible that \( a \) holds, and so \( \Pr(a|A) = 0 \); otherwise, if \( a \in A \), then \( \Pr(a|A) \) has a value that is proportional to \( \Pr(a) \). One realizes that the factor of proportionality has to be \( 1/\Pr[A] \), so that probabilities sum to 1 again. Our definition of conditional probability of an elementary event is then

\[
\Pr(a|A) = \begin{cases} 
0 & \text{If } a \notin A \\
\frac{\Pr(a)}{\Pr[A]} & \text{Otherwise}
\end{cases}
\]
The above formula already lets us solve the question asked at the beginning of this section. Notice that probabilities conditioned on an event $A$ such that $\Pr[A] = 0$ are undefined. Then we extend the definition to arbitrary events, and we say that for an event $B$

$$\Pr[B|A] = \sum_{b \in B} \Pr(b|A)$$

One should check that the following (more standard) definition is equivalent

$$\Pr[B|A] = \frac{\Pr[A \cap B]}{\Pr[A]}$$

**Definition 3.1** Two events $A$ and $B$ are independent if

$$\Pr[A \cap B] = \Pr[A] \cdot \Pr[B]$$

If $A$ and $B$ are independent, and $\Pr[A] > 0$, then we have $\Pr[B|A] = \Pr[B]$. Similarly, if $A$ and $B$ are independent, and $\Pr[B] > 0$, then we have $\Pr[A|B] = \Pr[A]$. This motivates the use of the term “independence.” If $A$ and $B$ are independent, then whether $A$ holds or not is not influenced by the knowledge that $B$ holds or not.

When we have several events, we can define a generalized notion of independence.

**Definition 3.2** Let $A_1, \ldots, A_n \subseteq \Omega$ be events is a sample space $\Omega$; we say that such events are mutually independent if for every subset of indices $I \subseteq \{1, \ldots, n\}$, $I \neq \emptyset$, we have

$$\Pr[\bigcap_{i \in I} A_i] = \prod_{i \in I} \Pr[A_i]$$

All this stuff was just to prepare for the definition of independence for random variables, which is a very important and useful notion.

**Definition 3.3** If $X$ and $Y$ are random variables over the same sample space, then we say that $X$ and $Y$ are independent if for any two values $v, w$, the event $(X = v)$ and $(Y = w)$ are independent.

Therefore, if $X$ and $Y$ are independent, knowing the value of $X$, no matter which value it is, does not tell us noting about the distribution of $Y$ (and vice versa).

**Theorem 3.4** If $X$ and $Y$ are independent, then $\mathbf{E}[XY] = \mathbf{E}[X] \mathbf{E}[Y]$.

This generalizes to several random variables

**Definition 3.5** Let $X_1, \ldots, X_n$ be random variables over the same sample space, then we say that they are mutually independent if for any sequence of values $v_1, \ldots, v_n$, the events $(X_1 = v_1), \ldots, (X_n = v_n)$ are mutually independent.

**Theorem 3.6** If $X_1, \ldots, X_n$ are mutually independent random variables, then

$$\mathbf{E}[X_1 \cdot X_2 \cdots X_n] = \mathbf{E}[X_1] \cdot \mathbf{E}[X_2] \cdots \mathbf{E}[X_n]$$
3.2 Pairwise Independence

It is also possible to define a weaker notion of independence.

**Definition 3.7** Let $X_1, \ldots, X_n$ be random variables over the same sample space, then we say that they are pairwise independent if for every $i, j \in \{1, \ldots, n\}, i \neq j$, we have that $X_i$ and $X_j$ are independent.

It is important to note that a collection of random variables can be pairwise independent without being mutually independent. (But a collection of mutually independent random variables is always pairwise independent for a stronger reason).

**Example 2** Consider the following probabilistic system: we toss 2 coins, and we let the random variables $X, Y, Z$ be, respectively, the outcome of the first coin, the outcome of the second coin, and the XOR of the outcomes of the two coins (as usual, we interpret outcomes of coins as 0/1 values). Then $X, Y, Z$ are not mutually independent, for example

$$\Pr[Z = 0 | X = 0, Y = 0] = 1$$

while

$$\Pr[Z = 0] = 1/2$$

in fact, intuitively, since the value of $Z$ is totally determined by the values of $X$ and $Y$, the three variables cannot be mutually independent. On the other hand, we will now show that $X, Y, Z$ are pairwise independent. By definition, $X$ and $Y$ are independent, so we have to focus of $X$ and $Z$ and on $Y$ and $Z$. Let us prove that $X$ and $Z$ are independent (the proof for $Y$ and $Z$ is identical). We have to show that for each choice of two values $v, w \in \{0, 1\}$, we have

$$\Pr[X = v, Z = w] = \Pr[X = v] \Pr[Z = w] = \frac{1}{4}$$

and this is true, since, in order to have $Z = w$ and $X = v$, we must have $Y = w \oplus v$, and the event that $X = v$ and $Y = w \oplus v$ happens with probability $1/4$. $\blacksquare$

Let us see two additional, more involved, examples.

**Example 3** Suppose we flip $k$ coins, whose outcomes be $a_1, \ldots, a_n \in \{0, 1\}^k$. Then for every non-empty subset $I \subseteq \{0, 1\}^k$ we define a random variable $X_I$, whose value is $\bigoplus_{i \in I} a_i$. It is possible to show that $\{X_I\}_{I \subseteq \{0, 1\}^k, I \neq \emptyset}$ is a pairwise independent collection of random variables. Notice that we have $2^k - 1$ random variables defined over a sample space of only $2^k$ points. $\blacksquare$

**Example 4** Let $p$ be a prime number; suppose we pick at random two elements $a, b \in \mathbb{Z}_p$ — that is, our sample space is the set of pairs $(a, b) \in \mathbb{Z}_p \times \mathbb{Z}_p = \Omega$, and we consider the uniform distribution over this sample space. For every $z \in \mathbb{Z}_p$, we define one random variable $X_z$ whose value is $az + b \pmod{p}$. Thus we have a collection of $p$ random variables. It is possible to show that such random variables are pairwise independent. $\blacksquare$
4 Deviation from the Expectation

4.1 Markov’s Inequality

Say that $X$ is the random variable expressing the running time in seconds of an algorithm on inputs of a certain size, and that we computed $E[X] = 10$. Since this is the order of magnitude of the time that we expect to spend while running the algorithm, it would be devastating if it happened that, say, $X \geq 1,000,000$ (i.e. more than 11 days) with large probability. However, we quickly realize that if $E[X] = 10$, then it must be $\Pr[X \geq 1,000,000] \leq 1/100,000$, as otherwise the contribution to the expectation of the only events where $X \geq 1,000,000$ would already exceed the value 10. This reasoning can be generalized as follows.

**Theorem 4.1 (Markov’s Inequality)** If $X$ is a non-negative random variable then

$$\Pr[X \geq k] \leq \frac{E[X]}{k}$$

Sometimes the bound given by Markov’s inequality are extremely bad, but the bound is as strong as possible if the only information that we have is the expectation of $X$.

For example, suppose that $X$ counts the number of heads in a sequence of $n$ coin flips. Formally, $\Omega = \{0,1\}^n$ with the uniform distribution, and $X$ is the number of ones in the string. Then $E[X] = n/2$. Suppose we want to get an upper bound on $\Pr[X \geq n]$ using Markov. Then we get

$$\Pr[X \geq n] \leq \frac{E[X]}{n} = \frac{1}{2}$$

This is ridiculous! The right value is $2^{-n}$, and the upper bound given by Markov’s inequality is totally off, and it does not even depend on $n$.

However, consider now the experiment where we flip $n$ coins that are glued together, so that the only possible outcomes are $n$ heads (with probability 1/2) and $n$ tails (with probability 1/2). Define $X$ again as the number of heads. We still have that $E[X] = n/2$, and we can apply Markov’s inequality as before to get

$$\Pr[X \geq n] \leq \frac{E[X]}{n} = \frac{1}{2}$$

But, now, the above inequality is tight, because $\Pr[X \geq n]$ is precisely 1/2.

The moral is that Markov’s inequality is very useful because it applies to every non-negative random variables having a certain expectation, so we can use it without having to study our random variable too much. On the other hand, the inequality will be accurate when applied to a random variable that typically deviates a lot from its expectation (say, the number of heads that we get when we toss $n$ glued coins) and the inequality will be very bad when we apply it to a random variable that is concentrated around its expectation (say, the number of heads that we get in $n$ independent coin tosses). In the latter case, if we want accurate estimations we have to use more powerful methods. One such method is described below.
4.2 Variance

For a random variable $X$, the random variable $X' = |X - \mathbb{E}[X]|$ gives all the information that we need in order to decide whether $X$ is likely to deviate a lot from its expectation or not. All we need to do is to prove that $X'$ is typically small. However this idea does not lead us very far (analysing $X'$ does not seem to be any easier than analysing $X$).

Here is a better tool. Consider

$$(X - \mathbb{E}[X])^2$$

This is again a random variable that tells us how much $X$ deviates from its expectation. In particular, if the expectation of such an auxiliary random variable is small, then we expect $X$ to be typically close to its expectation. The variance of $X$ is defined as

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2]$$

Here is an equivalent expression (we use linearity of expectation in the derivation of the final result):

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - 2\mathbb{E}[X]\mathbb{E}[X] + (\mathbb{E}[X])^2$$

The variance is a useful notion for two reasons: it is often easy to compute and it gives rise to sometimes strong estimations on the probability that a random variable deviates from its expectation.

**Theorem 4.2 (Chebyshev’s Inequality)**

$$\Pr[|X - \mathbb{E}[X]| \geq k] \leq \frac{\text{Var}(X)}{k^2}$$

The proof uses Markov’s inequality and a bit of ingenuity.
The nice idea is in the first step. The second step is just an application of Markov’s inequality and the last step uses the definition of variance.

The value \( \sigma(X) = \sqrt{\text{Var}(X)} \) is called the standard deviation of \( X \). One expects the value of a random variable \( X \) to be around the interval \( \mu \pm \sigma(X) \). We can restate Chebyshev’s Inequality in terms of standard deviation.

**Theorem 4.3 (Chebyshev’s Inequality, Alternative Form)**

\[
\Pr(\lvert X - \mu \rvert \geq c \cdot \sigma(X)) \leq \frac{1}{c^2}
\]

Let \( Y \) be a random variable that is equal to 0 with probability 1/2 and to 1 with probability 1/2. Then \( \mu[Y] = 1/2 \), \( Y = Y^2 \), and

\[
\text{Var}(Y) = \mu[Y^2] - (\mu[Y])^2 = \frac{1}{2} - \frac{1}{4} = \frac{1}{4}
\]

Let \( X \) the random variable that counts the number of heads in a sequence of \( n \) independent coin flips. We have seen that \( \mu[X] = n/2 \). Computing the variance according to the definition would be painful. We are fortunate that the following result holds.

**Lemma 4.4 (Tools to Compute Variance)**

1. Let \( X \) be a random variable, \( a, b \) be reals, then

\[
\text{Var}(aX + b) = a^2\text{Var}(X)
\]

2. Let \( X_1, \ldots, X_n \) be pairwise independent random variables on the same sample space. Then

\[
\text{Var}(X_1 + \cdots + X_n) = \text{Var}(X_1) + \cdots + \text{Var}(X_n)
\]

Then we can view \( X \) as \( X_1 + \cdots + X_n \) where \( X_i \) are mutually independent random variables such that for each \( i \) \( X_i \) takes value 1 with probability 1/2 and value 0 with probability 1/2.

As computed before, \( \text{Var}(X_i) = 1/4 \). Therefore \( \text{Var}(X) = n/4 \) and the standard deviation is \( \sqrt{n}/2 \). This means that when we flip \( n \) coins we expect to get about \( n \pm \sqrt{n} \) heads.
Let us test Chebyshev’s inequality on the same example of the previous subsection. Let $X$ be a random variable defined over $\Omega = \{0, 1\}^n$, where $\Pr$ is uniform, and $X$ counts the number of 1s in the elementary event: suppose we want to compute $\Pr[X \geq n]$. As computed above, $\text{Var}(X) = n/4$, so
\[
\Pr[X \geq n] \leq \Pr[|X - E[X]| \geq n/2] \leq \frac{1}{n}
\]
This is still much less than the correct value $2^{-n}$, but at least it is a value that decreases with $n$. It is also possible to show that Chebyshev’s inequality is as strong as possible given its assumption.

Let $n = 2^k - 1$ for some integer $k$ and let $X_1, \ldots, X_n$ be the collection of pairwise independent random variables as defined in Example 3. Let $X = X_1 + \ldots + X_n$. Suppose we want to compute $\Pr[X = 0]$. Since each $X_i$ has variance $1/4$, we have that $X$ has variance $n/4$, and so
\[
\Pr[X = 0] \leq \Pr[|X - E[X]| \geq n/2] \leq \frac{1}{n}
\]
which is almost the right value: the right value is $2^{-k} = 1/(n + 1)$.

A Appendix

A.1 Some Combinatorial Facts

Consider a set $\Omega$ with $n$ elements. $\Omega$ has $2^n$ subsets (including the empty set and $\Omega$ itself). For every $0 \leq k \leq n$, $\Omega$ has $\binom{n}{k}$ subsets of $k$ elements. The symbol $\binom{n}{k}$ is read “$n$ choose $k$” and is defined as
\[
\binom{n}{k} = \frac{n!}{k!(n-k)!}
\]
Then we must have
\[
\sum_{k=0}^{n} \binom{n}{k} = 2^n
\]
which is a special case of the following result

Theorem A.1 (Binomial Theorem) For every two reals $a, b$ and non-negative integer $n$,
\[
(a + b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}
\]
We can see that Equation (2) follows from the Binomial Theorem by simply substituting $a = 1$ and $b = 1$.

Sometimes we have to deal with summations of the form $1 + 1/2 + 1/3 + \ldots + 1/n$. It’s good to know that $\sum_{k=1}^{n} 1/k \approx \ln n$. More precisely

**Theorem A.2** $\lim_{n \to \infty} \frac{\sum_{k=1}^{n} 1/k}{\ln n} = 1$.

In particular, $\sum_{k=1}^{n} 1/k \leq 1 + \ln n$ for every $n$, and $\sum_{k=1}^{n} 1/k \geq \ln n$ for sufficiently large $n$. The following inequality is exceedingly useful in computing upper bounds of probabilities of events:

$$1 + x \leq e^x$$

This is easy to prove by looking at the Taylor series of $e^x$:

$$e^x = 1 + x + \frac{x^2}{2} + \ldots + \frac{1}{k!}x^k + \ldots$$

Observe that Equation (3) is true for *every* real $x$, not necessarily positive (but it becomes trivial for $x < -1$).

Here is a typical application of Equation (3). We have a randomized algorithm that has a probability $\epsilon$ over its internal coin tosses of succeeding in doing something (and when it succeeds, we notice that it does, say because the algorithm is trying to invert a one-way function, and when it succeeds then we can check it efficiently); how many times do we have to run the algorithm before we have probability at least $3/4$ that the algorithm succeeds?

The probability that it never succeeds in $k$ runs is

$$(1 - \epsilon)^k \leq e^{-\epsilon k}$$

If we choose $k = 2/\epsilon$, the probability of $k$ consecutive failures is less than $e^{-2} < 1/4$, and so the probability of succeeding (at least once) is at least $3/4$.

### A.2 Examples of Analysis of Error Probability of Algorithms

**Example 5** Suppose that we have an algorithm whose worst-case running time (on inputs of a certain length) is bounded by a random variable $T$ (whose sample space is the set of random choices made by the algorithm). For concreteness, suppose that we are considering the randomized algorithm that given a prime $p$ and an element $a \in \mathbb{Z}_p^*$ decides whether $a$ is a quadratic residue or not. Suppose that we are given $t = \mathbb{E}[T]$ but no additional information on the algorithm, and we would like to know how much time we have to wait in order to have a probability at least $1 - 10^{-6}$ that the algorithm terminates. If we only know $\mathbb{E}[T]$, then we can just use Markov’s inequality and say that
\[ \Pr[T \geq kt] \leq \frac{1}{k} \]

and if we choose \( k = 10^6 \) we have that
\[ \Pr[T \geq 10^6t] \leq 10^{-6} . \]

However there is a much faster way of guaranteeing termination with high probability. We let the program run for \( 2t \) time. There is a probability \( 1/2 \) that the algorithm will stop before that time. If so we are happy. If not, we terminate the computation, and start it over (in the second iteration, we let the algorithm use independent random bits). If the second computation does not terminate within \( 2t \) time, we reset it once more, and so on.\(^1\) Let \( T' \) be the random variable that gives the time taken by this new version of the algorithm (with the stop and reset actions). Now we have that the probability that we use more than \( 2kt \) time is equal to the probability that for \( k \) consecutive (independent) times the algorithm takes more than \( 2t \) time. Each of these events happen with probability at most \( 1/2 \), and so
\[ \Pr[T' \geq 2kt] \leq 2^{-k} \]
and if take \( k = 20 \), the probability is less than \( 10^{-6} \), and the time is only \( 40t \) rather than \( 1,000,000t \).

Suppose that \( t = t(n) \) is the average running time of our algorithm on inputs of length \( n \), and that we want to find another algorithm that finishes always in time \( t'(n) \) and that reports a failure only with negligible probability, say with probability at most \( n^{-\log n} \). How large do we have choose \( t' \), and what the new algorithm should be like?

If we just put a timeout \( t' \) on the original algorithm, then we can use Markov’s inequality to say that \( t'(n) = n^{\log n} t(n) \) will suffice, but now \( t' \) is not polynomial in \( n \) (even if \( t \) was). Using the second method, we can put a timeout \( 2t \) and repeat the algorithm \( (\log n)^2 \) times. Then the failure probability will be as requested and \( t'(n) = 2(\log n)^2 t(n) \).

If we know how the algorithm works, then we can make a more direct analysis.

**Example 6** Suppose that our goal is, given \( n \), to find a number \( 2 \leq a \leq n - 1 \) such that \( \gcd(a, n) = 1 \). To simplify notation, let \( l = ||n|| \approx \log n \) be the number of digits of \( n \) in binary notation (in a concrete application, \( l \) would be a few hundreds). Our algorithm will be as follows:

- Repeat no more than \( k \) times:
  1. Pick uniformly at random \( a \in \{2, \ldots, n - 1\} \);
  2. Use Euclid’s algorithm to test whether \( \gcd(a, n) = 1 \).

\(^1\)This is reminiscent of the way one works with Windows’98.
3. If \( \gcd(a, n) = 1 \) then output \( a \) and halt.

- Output “failure”.

We would like to find a value of \( k \) such that the probability that the algorithm reports a failure is negligible in the size of the input (i.e. in \( l \)).

At each iteration, the probability that algorithm finds an element that is coprime with \( n \) is

\[
\frac{\phi(n)}{n - 2} \geq \frac{1}{6 \log \log n} = \frac{1}{6 \log l}
\]

So the probability that there is a failure in one iteration is at most

\[
\left(1 - \frac{1}{6 \log l}\right)
\]

and the probability of \( k \) consecutive independent failures is at most

\[
\left(1 - \frac{1}{6 \log l}\right)^k \leq e^{-k/6 \log l}
\]

if we set \( k = (\log l)^3 \) then the probability of \( k \) consecutive failures is at most

\[
e^{- (\log l)^3 / 6 \log l} = l^{-(\log l)/6}
\]

that is negligible in \( l \).
Midterm Solutions

1. A directed graph \( G = (V, E) \) is strongly connected if for any two vertices \( u, v \in V \) there is a directed path in \( G \) from \( u \) to \( v \). Let strong-CONN be the problem of deciding whether a given graph is strongly connected.

(a) Show that strong-CONN is in \( \text{NL} \).

(b) Prove the \( \text{NL} \)-completeness of strong-CONN by giving a log-space reduction from ST-CONN to strong-CONN.

Solutions

The NL algorithm just goes through all pairs of vertices \( u, v \) and guesses a path from \( u \) to \( v \), rejecting if a path is not found in less than \( |V| \) steps. If the graph is strongly connected, there is at least an accepting computations that guesses the paths right correctly each time, and if there is an accepting computation, it must be the case that the graph is strongly connected.

In the reduction, given \( G = (V, E), s, t, \) create \( G' = (V, E') \) where \( E' \) has all the edges of \( E \) and also all edges \((v, s)\) and \((t, v)\). If there is a path from \( s \) to \( t \), then from every vertex \( u \) we can go to every vertex \( v \) by using the edge \((u, s)\), the path from \( s \) to \( t \) and the edge \((t, v)\), so \( G' \) is strongly connected. But if \( G' \) is strongly connected, there is a simple path in \( G' \) from \( s \) to \( t \), and such simple path will not use any of the new edges, so there was already a path from \( s \) to \( t \) in \( G \).
2. Suppose that there is a deterministic polynomial-time algorithm \( A \) that on input (the description of) a circuit \( C \) produces a number \( A(C) \) such that
\[
\Pr_x[C(x) = 1] - \frac{2}{5} \leq A(C) \leq \Pr_x[C(x) = 1] + \frac{2}{5}.
\]
(a) Prove that it follows \( P = BPP \).
(b) Prove that there exists a deterministic algorithm \( A' \) that, on input a circuit \( C \) and a parameter \( \epsilon \), runs in time polynomial in the size of \( C \) and in \( 1/\epsilon \) and produces a value \( A'(C, \epsilon) \) such that
\[
\Pr_x[C(x) = 1] - \epsilon \leq A'(C, \epsilon) \leq \Pr_x[C(x) = 1] + \epsilon.
\]
(c) Prove that there exists a deterministic algorithm \( A'' \) that, on input a circuit \( C \) computing a function \( f : \{0,1\}^n \to \{1, \ldots, k\} \) and a parameter \( \epsilon \), runs in time polynomial in the size of \( C \), in \( 1/\epsilon \) and in \( k \), and produces a value \( A''(C, \epsilon) \) such that
\[
E_x[f(x)] - \epsilon \leq A''(C, \epsilon) \leq E_x[f(x)] + \epsilon.
\]
[For this question, you can think of \( C \) as being a circuit with \( \log k \) outputs, and the outputs of \( C(x) \) are the binary representation of \( f(x) \).]

Solutions

Part (2a) was easy: let \( L \) be a \( BPP \) language and \( A_L \) the probabilistic polynomial time algorithm that decides \( L \), we may assume that the error probability of \( L \) is \( \leq 1/10 \). Here is a deterministic polynomial time algorithm for \( L \): on input \( x \), construct the circuit \( C \) such that \( C(r) = 1 \) if and only if \( A(x, r) \) accepts. Then accept if and only if \( A(C) > 1/2 \).

Several people said that, once we have \( P = BPP \), part (2b) can be solved by just giving a probabilistic algorithm that outputs the right answer with high probability. This is not quite right, because \( P = BPP \) is a statement that concerns only languages, not function computations, and if you try to prove formally that \( P = BPP \) implies a solution to part (2b) you will run into trouble. In fact, it is a major open question to show that \( P = BPP \) implies part (2b). However, it is known that if \( P = BPP \) for promise problems then part (2b) follows, and the argument above for part (2a) also extends to promise problems. Instead of formulating the solution in terms of promise problems, it seems simpler to just give it directly.

We solve part (2b) by giving a deterministic algorithm \( ApxComp \) that, on input a circuit \( C \), an accuracy parameter \( \epsilon > 0 \) and a threshold parameter \( 0 \leq p \leq 1 \), runs in time polynomial in the size of \( C \) and in \( 1/\epsilon \) and:

(a) If \( \Pr_x[C(x) = 1] \geq p + \epsilon/2 \), then \( ApxComp(C, p, \epsilon) \) accepts;
(b) If \( \Pr_x[C(x) = 1] \leq p - \epsilon/2 \), then \( ApxComp(C, p, \epsilon) \) rejects;
We run $ApxComp(C, i \cdot \epsilon/2, \epsilon)$ for $i = 0, 1, \ldots, 2/\epsilon$, and we find the smallest $i$ such that $ApxComp(C, (i - 1) \cdot \epsilon/2, \epsilon)$ accepts but $ApxComp(C, i \cdot \epsilon/2, \epsilon)$ rejects. Then $\Pr[C(x) = 1] > (i - 1) \cdot \epsilon/2 - \epsilon/2$ and $\Pr[C(x) = 1] < i\epsilon/2 + \epsilon/2$, so we can safely output $i\epsilon/2$ as our solution.

In order to devise the procedure $ApxComp$, it is enough to show how, given $C$, $\epsilon$ and $p$, to construct a new circuit $C'$ in time polynomial in the size of $C$ and in $1/\epsilon$ such that:

(a) If $\Pr_x[C(x) = 1] \geq p + \epsilon/2$, then $\Pr_z[C'(z) = 1] \geq 9/10$;
(b) If $\Pr_x[C(x) = 1] \leq p - \epsilon/2$, then $\Pr_z[C'(z) = 1] \leq 1/10$.

Then we just compute $A(C')$ and accept if and only if $A(C') > 1/2$.

The circuit $C'$ has $t = O(1/\epsilon^2)$ inputs $x_1, \ldots, x_t$, where each $x_i$ is an $n$-bit string (here $n$ is the input length of $C$). The circuits $C'$ computes $C(x_1), \ldots, C(x_t)$ and accepts iff at least $pt$ of such computations output 1. One can use Chebyshev inequality to prove correctness.

Regarding part (2c), there are various ways of reducing the problem to the problem solved in part (2b). For example, one can compute the probability that each of the $\log k$ wires carries a one (up to an additive error $\epsilon/\log k$), and then use such estimates to compute an estimate of the average up to an additive error $\epsilon$.

Otherwise, given a circuit $C$ with $n$-bit input and $\log k$ bit output, one can define the circuit $C'$ with $n + \log k$ bit input and one-bit output such that $C'(x, i) = 1$ if and only if $C(x) \geq i$. Then, it should be easy to see that $\Pr_{x,i}[C'(x, i) = 1] = \frac{1}{\log k} \operatorname{E}_x[C(x)]$. 

3
3. Prove that, for every constant $t$, $\Sigma_2 \not\subseteq \text{SIZE}(n^t)$.

[Hint: first prove $\Sigma_4 \not\subseteq \text{SIZE}(n^t)$, which should be easy. Then argue about what happens depending on whether or not $\text{SAT} \in \text{SIZE}(n^t)$.

Solutions

Let us fix an arbitrary $t$, and see that $\Sigma_4 \not\subseteq \text{SIZE}(n^t)$.

As a first step, let us see that there is a polynomial $p(n)$ such that $\text{SIZE}(p(n)) \not\subseteq \text{SIZE}(n^t)$. This can be proved in many different ways, providing different bounds on $p(n)$. Here are three possible arguments:

(a) We know that every function $f : \{0, 1\}^k \to \{0, 1\}$ can be computed by a circuit of size $O(2^k)$, and that there are functions $f : \{0, 1\}^k \to \{0, 1\}$ that cannot be computed by circuit of size $2^k/4k$. Let $k(n)$ be such that $2^{k(n)/4k(n)} > n^t$, for example $k(n) = \log_2(5n^t \log_2 n^t)$ will do for sufficiently large $n$. Then for every sufficiently large $n$ there is a function $f_{k(n)} : \{0, 1\}^{k(n)} \to \{0, 1\}$ that cannot be computed by circuits of size $n^t$ but that can be computed by circuits of size $O(2^{k(n)}) = O(n^t \log n^t)$. Define the language $L$ such that $x \in L$ if and only if $f_{k(n)}$ applied to the first $k(n)$ bits of $x$ equals 1, where $n$ is the length of $x$. Then $L \in \text{SIZE}(O(n^t \log n^t))$ but $L \not\subseteq \text{SIZE}(n^t)$.

(b) We know that there are at most $2^{2n^t \log n^t + 5n^t}$ circuits of size $n^t$. Suppose that we are able to define a family of functions such that each function in the family can be computed by a circuit of size $\leq p(n)$ and such that the family contains more than $2^{2n^t \log n^t + 5n^t}$ distinct functions. Then the family contains a function computable by a circuit of size $p(n)$ but not computable by a circuit of size $\leq n^t$.

The above proof can be seen as an application of this method where the family is the family of all functions that depend on only the first $5n^t \log n$ bits of the input. Another approach would be to fix a set $S$ of $2^{n^t \log n^t + 5n^t} + 1$ elements of $\{0, 1\}^n$ and then consider all the functions that are zero outside of $S$. There are $2^{|S|}$ such function, each computable by a circuit of size $O(|S| : n)$.

(c) Suppose that $\text{SIZE}(n^t + 2n + 1) = \text{SIZE}(n^t)$. We claim that this implies that every function $f : \{0, 1\}^n \to B$ can be computed by a circuit of size $n^t$, which we know to be false (for large enough $n$). We prove the claim by induction on the number of inputs $x$ such that $f(x) = 1$.

i. If $f(x) = 0$ for every $x$, then $f$ can be computed by a circuit of constant size, and, for a stronger reason, by a circuit of size $n^t$.

ii. Suppose that we have proved that, for every $g : \{0, 1\}^n \to \{0, 1\}$ that outputs 1 on $\leq K$ inputs, $g \in \text{SIZE}(n^t)$. Let $f : \{0, 1\}^n \to \{0, 1\}$ be a function that outputs 1 on $K + 1$ inputs, and let $a$ be one such input. Define the function $g$ such that $g(x) = f(x)$ for $x \neq a$, and $g(a) = 0$. Then $g$ outputs 1 on $K$ inputs, the induction hypothesis can be applied, and $g$ can be computed by a circuit of size $\leq n^t$. But we can write $f(x) = g(x) \lor (x = a)$, where the
expression \((x = a)\) can be realized by a circuit of size at most \(2n\). Then \(f \in \text{SIZE}(n^t + 2n + 1)\), and, by the assumption, we also have \(f \in \text{SIZE}(n^t)\).

Let us use the last solution. So we know that is a circuit \(C\) with \(n\) inputs of size \(n^t + n + 1\) that computes a function that cannot be computed by circuits of size \(n^t\).

We define a language \(L\) in \(\Sigma_4\) that, for every input length \(n\), agrees with the output of the lexicographically first circuit like that. Let \(C_s\) be the set of circuits of size \(s\), and let \(\prec_L\) denote the lexicographic ordering, then we can define \(L\) formally as follows. For every \(x \in \{0, 1\}^n\), we have

\[
x \in L \text{ if and only if } \exists C \in C_{n^t+n+1} : \forall C' \in C_{n^t} : \exists y \in \{0, 1\}^n. C(y) \neq C'(y) \land \exists C'' \in C_{n^t+n+1} : C'' <_L C',
\]

\[
\exists C''' \in C_{n^t} : \forall z. C''(z) = C'''(z) \land C(x) = 1
\]

Which is logically equivalent to the \(\Sigma_4\) formulation

\[
x \in L \text{ if and only if } \exists C \in C_{n^t+n+1} : \forall C' \in C_{n^t}, C'' \in C_{n^t+n+1} : \exists y \in \{0, 1\}^n, C''' \in C_{n^t} : \forall z. C(y) \neq C'(y) \land ((C'' <_L C) \Rightarrow (C''(z) = C'''(z))) \land C(x) = 1
\]

And so we have \(\Sigma_4 \not\subseteq \text{SIZE}(n^t)\). Now, if \(\text{NP} \subseteq \text{SIZE}(n^t)\), then, by Karp-Lipton, we have \(\Sigma_2 = \Sigma_4 \not\subseteq \text{SIZE}(n^t)\); if \(\text{NP} \not\subseteq \text{SIZE}(n^t)\), then, for a stronger reason, \(\Sigma_2 \not\subseteq \text{SIZE}(n^t)\).
4. Let $f$ be a one-way permutation and $g$ be a polynomial time computable permutation. Show that $g(f(\cdot))$ and $f(g(\cdot))$ are one-way permutations.

[Ideally, do the proof in the finite setting: show that if $f$ is $(S, \epsilon)$-one way and $g$ can be computed by a circuit of size $t$, then $f(g(\cdot))$ and $g(f(\cdot))$ are $(S - t, \epsilon)$-one way. Then derive the asymptotic result from the finite one. Be as detailed as you can in the analysis.]

Solutions

Let $g$ be computable by a circuit of size $\leq t$, and suppose that $f(g(\cdot))$ is not $(S - t, \epsilon)$ one way. Then there is a function $A$ computable by a circuit of size $S - t$ such that

$$\Pr[A(f(g(x))) = x] \geq \epsilon$$

which also implies that

$$\Pr[g(A(f(g(x)))) = g(x)] \geq \epsilon$$

Let us do the change of variable $y \leftarrow g(x)$, notice that the uniform distribution over $x$ corresponds to the uniform distribution over $y$, call $B() := g(A())$, and we have

$$\Pr[B(f(y)) = y] \geq \epsilon$$

where $B$ is computable by a circuit of size $\leq S$. That is, if $f(g(\cdot))$ is not $(S - t, \epsilon)$ one way, then $f$ is not $(S, \epsilon)$ one way.

Suppose now that $g(f(\cdot))$ is not $(S - t, \epsilon)$ one way, so that there is $A$ computable in size $S - t$ such that

$$\Pr[A(g(f(x))) = x] \geq \epsilon.$$ 

Define $B() := A(g())$, and note that $B$ can be computed in size $\leq S$ and

$$\Pr[B(f(x)) = x] \geq \epsilon,$$

proving that $f$ is not $(S, \epsilon)$ one way. That is, if $f(g(\cdot))$ is not $(S - t, \epsilon)$ one way, then $f$ is not $(S, \epsilon)$ one way.

Asymptotically, let $t(n)$ be a polynomial such that $g$ is computable in time $t(n)$.

Suppose that $f(g(\cdot))$ is not one-way, then there are polynomials $p, q$ such that, for infinitely many $n$, $f(g(n))$ is not $(p(n), 1/q(n))$-one way on inputs of length $n$, and so $f$ is not $(p(n) + t(n), 1/q(n))$ one way on inputs of length $n$. This contradicts the assumption that $f$ is one-way.

Suppose that $g(f(\cdot))$ is not one-way, then there are polynomials $p, q$ such that, for infinitely many $n$, $g(f(n))$ is not $(p(n), 1/q(n))$-one way on inputs of length $n$, and so $f$ is not $(p(n) + t(n), 1/q(n))$ one way on inputs of length $n$. This contradicts the assumption that $f$ is one-way.
Midterm

This is due in class on November 8.

1. A directed graph $G = (V, E)$ is strongly connected if for any two vertices $u, v \in V$ there is a directed path in $G$ from $u$ to $v$. Let strong-CONN be the problem of deciding whether a given graph is strongly connected.

   (a) Show that strong-CONN is in $\text{NL}$.
   (b) Prove the $\text{NL}$-completeness of strong-CONN by giving a log-space reduction from ST-CONN to strong-CONN.

2. Suppose that there is a deterministic polynomial-time algorithm $A$ that on input (the description of) a circuit $C$ produces a number $A(C)$ such that
   \[
   \Pr_x [C(x) = 1] - \frac{2}{5} \leq A(C) \leq \Pr_x [C(x) = 1] + \frac{2}{5}.
   \]

   (a) Prove that it follows $\mathbf{P} = \mathbf{BPP}$.
   (b) Prove that there exists a deterministic algorithm $A'$ that, on input a circuit $C$ and a parameter $\epsilon$, runs in time polynomial in the size of $C$ and in $1/\epsilon$ and produces a value $A'(C, \epsilon)$ such that
   \[
   \Pr_x [C(x) = 1] - \epsilon \leq A'(C, \epsilon) \leq \Pr_x [C(x) = 1] + \epsilon.
   \]
   (c) Prove that there exists a deterministic algorithm $A''$ that, on input a circuit $C$ computing a function $f : \{0,1\}^n \to \{1,\ldots,k\}$ and a parameter $\epsilon$, runs in time polynomial in the size of $C$, in $1/\epsilon$ and in $k$, and produces a value $A''(C, \epsilon)$ such that
   \[
   \mathbf{E}_x [f(x)] - \epsilon \leq A''(C, \epsilon) \leq \mathbf{E}_x [f(x)] + \epsilon.
   \]

   [For this question, you can think of $C$ as being a circuit with log $k$ outputs, and the outputs of $C(x)$ are the binary representation of $f(x)$.] 

3. Prove that, for every constant $t$, $\Sigma_2 \not\subseteq \text{SIZE}(n^t)$.

   [Hint: first prove $\Sigma_4 \not\subseteq \text{SIZE}(n^t)$, which should be easy. Then argue about what happens depending on whether or not $\mathbf{SAT} \in \text{SIZE}(n^t)$] 

4. Let $f$ be a one-way permutation and $g$ be a polynomial time computable permutation. Show that $g(f(\cdot))$ and $f(g(\cdot))$ are one-way permutations.

   [Ideally, do the proof in the finite setting: show that if $f$ is $(S, \epsilon)$-one way and $g$ can be computed by a circuit of size $t$, then $f(g(\cdot))$ and $g(f(\cdot))$ are $(S - t, \epsilon)$-one way. Then derive the asymptotic result from the finite one. Be as detailed as you can in the analysis.]
Solutions to Exercises

1. Show that if $P = NP$ for decision problems, then every $NP$ search problem can be solved in polynomial time.

Solution Sketch. This is similar to something done in Lecture 7. Let $R$ be the relation that defines an $NP$ search problem. Define the language $L$ that contains all the pairs $(x, z)$ such that $z$ is the prefix of a solution $zz'$ such that $(x, zz') \in R$. Under the assumption that $P = NP$, there is a polynomial time algorithm $A$ that decides $L$. Now, given $x$, we can construct a solution $y$ such that $(x, y) \in R$, if such a solution exists, one bit a time using $A$. (See notes of lecture 7 for more details.)

2. Generalize Theorem 3 in the Notes for Lecture 1. Say that a monotone non-decreasing function $t : \mathbb{N} \to \mathbb{N}$ is time-constructible if, given $n$, we can compute $t(n)$ in $O(t(n))$ time. Show that if $t(n)$ is a time-constructible functions then $\text{DTIME}(o(t(n))) \not\subseteq \text{DTIME}(O(t(n) \log t(n)))$.

Solution Sketch. Let $U$ be the efficient universal Turing machine of Lecture 1. Define the language $L$ that contains all pairs $(\langle M \rangle, x)$ such that $U$ rejects $(\langle M \rangle, (\langle M \rangle, x))$ within $t(n)$ steps, where $n$ is the length of $(\langle M \rangle, x)$. Then $L$ is solvable in time $O(t(n) \log t(n))$. Suppose towards a contradiction that $L$ were solvable in time $o(t(n))$ by a machine $T$. Then, $U((T), (\langle T \rangle, x))$ also runs in time $o(t(n))$, where $n$ is the length of $(\langle M \rangle, x)$. For every sufficiently long $x$, the running time of $U((T), (\langle T \rangle, x))$ is less than $t(n)$, and so $(\langle T \rangle, x) \in L$ if and only if $U((T), (\langle T \rangle, x))$ rejects, which happens if and only if $T((\langle T \rangle, x) \not\in L$.

3. Define the class $BPL$ (for bounded-error probabilistic log-space) as follows. A decision problem $L$ is in $BPL$ if there is a log-space probabilistic Turing machine $M$ such that

- For every $r$ and every $x$, $M(r, x)$ halts;
- If $x \in L$ then $\Pr_r[M(r, x) \text{ accepts }] \geq 2/3$;
- If $x \notin L$ then $\Pr_r[M(r, x) \text{ accepts }] \leq 1/3$.

Then

(a) Prove that $RL \subseteq BPL$.

Solution Sketch. We only need to reduce the error probability from $1/2$ to below $1/3$, for example by doing two independent repetitions of the algorithm and accepting if and only if both repetitions accept.

(b) Prove that $BPL \subseteq \text{SPACE}(O((\log n)^2))$. 

1
Solution Sketch. This is similar to Savitch’s algorithm. Let $L$ be a BPL language and $M$ the machine required by the definition of BPL. Let $p(n)$ be a polynomial upper bound on the running time of $M$ on inputs of length $n$. We now describe a deterministic $O(\log^2 n)$-space algorithm for $L$. On input $x$ of length $n$, we consider the configuration graph of $M(\cdot, x)$. A vertex in the graph is a description of a configuration of $M$ on input $x$: state of the machine, content of the work tape, position of the head on the input tape, position of the head on the work tape. Each vertex has two outgoing edges, corresponding to the two possible next configurations of the machine depending on the bit read from the random tape. The accepting and rejecting configurations have no outgoing edges.

We define a recursive procedure $\text{prob}(c_1, c_2, k)$ that given two configurations $c_1$ and $c_2$ computes the probability that the machine reaches configuration $c_2$ starting from configuration $c_1$ in exactly $k$ steps. The base case when $k = 1$ are treated separately. When $k \geq 2$, then we compute

$$\sum_c \text{prob}(c_1, c, \lceil k/2 \rceil) \cdot \text{prob}(c, c_2, \lfloor k/2 \rfloor)$$

where the summation is taken over all configurations $c$. It is enough to represent the probabilities as truncated decimals using $O(\log p(n))$ digits. The truncation introduces errors, but the error in $\text{proc}(\cdot, \cdot, p(n))$ is small provided that we use $O(\log p(n))$ digits. The analysis of the recursion is as in Savitch’s algorithm.

(c) This last question requires a somewhat different approach: prove that $\text{BPL} \subseteq \text{P}$.

Solution Sketch. Let $L$ be a BPL language and $M$ the machine required by the definition of BPL. On input $x$ of length $n$, let $C$ be the number of configurations of $M(\cdot, x)$. Construct a $C \times C$ matrix $P$ such that $P[c_1, c_2] = 1/2$ if $c_2$ is reachable from $c_1$ in one step, and $P[c_1, c_2] = 0$ otherwise. For every $t$, $P^t[c_1, c_2]$ is the probability of reaching configuration $c_2$ from configuration $c_1$ in $t$ steps, where $P^t$ is the matrix obtained by multiplying $P$ with itself $t$ times. By computing all powers of $P$ up to the running time of $M(\cdot, x)$ we can compute the accepting probability of $M(r, x)$, and decide if $x \in L$. This time the calculations can be exact: each probability is an integer multiple of $1/2^{p(n)}$, and so it can be represented using a polynomial number of digits.

4. Show that $\text{SIZE}(n^{\Omega(1)}) \not\subseteq \text{P}$.

Solution Sketch. Fix an enumeration $M_1, M_2, \ldots, M_n, \ldots$ of Turing machines. Consider the language $L$ that contains all strings $x$ such that $M_n$ halts on an empty input, where $n$ is the length of $x$. Then $L$ is undecidable, and, in particular, it is not in $\text{P}$. On the other hand, on each input length $n$, there is a very simple circuit of size $O(1)$ (that either accepts all inputs or rejects all inputs) that solves $L$ on inputs of length $n$.

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

2
Solution Sketch. We know that for every sufficiently large $n$ there is a function $f : \{0,1\}^n \rightarrow \{0,1\}$ such that, say, there is no circuit of size $\leq 2^{n/2}$ that computes $f$. Let $h_n$ be the lexicographically first such function. (The lexicographic order is normally defined over strings, but it is easy to adapt it to functions.) Then define $L$ to be the language that contains all strings $x$ such that $h_n(x) = 1$, where $n$ is the length of $x$. Then $L$ cannot clearly be solved by circuits of size $\leq 2^{n/2}$ on inputs of length $n$, but it can be solved in exponential space as follows: given $x$ of length $n$, enumerate all functions $f : \{0,1\}^n \rightarrow \{0,1\}$ in lexicographic order until we find a function that cannot be solved by circuits of size $\leq 2^{n/2}$. The first such function that we encounter is $h_n()$, and then we simply evaluate $h_n(x)$. We can enumerate all functions by enumerating all possible truth-tables, which can be done using $O(2^n)$ space (after we are done considering a function, we re-use the same space to write down the truth-table of the next function). In order to check if a given function can be solved by a circuit of size $2^{n/2}$, it is enough to enumerate all circuits of size $\leq 2^{n/2}$, using $O(n \cdot 2^{n/2})$ space, and verify if any of the circuits agrees with the function on all inputs. This computation can certainly be done in $2^{O(n)}$ space.

6. In the MAX SAT problem we are given a formula $\varphi$ in conjunctive normal form and we want to find the assignment of values to the variables that maximizes the number of satisfied clauses. (For example, if $\varphi$ is satisfiable, the optimal solution satisfies all the clauses and the MAX SAT problem reduces to finding a satisfying assignment.) Consider the following decision problem: given a formula $\varphi$ in conjunctive normal form and an integer $k$, determine if $k$ is the number of clauses of $\varphi$ satisfied by an optimal assignment.

- Prove that this problem is in $\mathbf{NP}$ if and only if $\mathbf{NP} = \mathbf{coNP}$.

Solution Sketch. Suppose that MAX SAT is in $\mathbf{NP}$, and let $V(\cdot, \cdot)$ be the verifier for MAX SAT. Then we can deduce that the $\mathbf{coNP}$-complete problem UNSAT (where, given a CNF formula $\varphi$ we want to decide if it is unsatisfiable) is also in $\mathbf{NP}$: a witness that $\varphi$ is unsatisfiable is a pair $(k, y)$, where $k$ is an integer smaller than the number of clauses of $\varphi$ and $y$ is such that $V((\varphi, k), y)$ accepts. But if a $\mathbf{coNP}$-complete problem belongs to $\mathbf{NP}$, it follows that $\mathbf{coNP} = \mathbf{NP}$.

Suppose now that $\mathbf{NP} = \mathbf{coNP}$, and consider the language $L$ that contains pairs $(\varphi, k)$ such that at least $k$ clauses of $\varphi$ can be satisfied, and the language $L'$ that contains the pairs $(\varphi, k)$ such that it is impossible to satisfy $k$ or more clauses of $\varphi$. By definition, $L$ is in $\mathbf{NP}$, and let $V()$ be its verifier; also $L'$ is in $\mathbf{coNP}$, and, by the assumption, also in $\mathbf{NP}$, and let $V'()$ be its verifier. We can now put MAX SAT in $\mathbf{NP}$ by noting that a witness for $(\varphi, k) \in \text{MAX SAT}$ is a pair $(y, y')$ such that $V((\varphi, k), y)$ and $V'((\varphi, k + 1), y')$ both accept.

- Prove that this problem is in $\Sigma_2$.

Solution Sketch. By definition, $(\varphi, k) \in \text{MAX SAT}$ if and only if

$$\exists a. a \text{ satisfies } k \text{ clauses of } \varphi \land \forall a'. a' \text{ satisfies } \leq k \text{ clauses of } \varphi$$
which is logically equivalent to the $\Sigma_2$ formulation

$$\exists a.\forall a'. (a \text{ satisfies } k \text{ clauses of } \varphi \text{ and } a' \text{ satisfies } \leq k \text{ clauses of } \varphi)$$

7. Define $\mathbf{EXP} = \mathbf{DTIME}(2^{n^{O(1)}})$. Prove that if $\mathbf{EXP} \subseteq \mathbf{SIZE}(n^{O(1)})$ then $\mathbf{EXP} = \Sigma_2$.

**Solution Sketch.** Let $L \in \mathbf{EXP}$ and let $M$ be the Turing machine that solves $L$ in time $\leq 2^{p(n)}$ on inputs of length $n$, where $n$ is the length of the input. Fix a representation of the configurations of $M(x)$; each configuration can be written using $c \cdot 2^{p(n)}$ bits for some constant $c$. There is a machine $M'$ that, given a string $x$ of length $n$, and integers $t \leq 2^p(n)$ and $i \leq c \cdot 2^p(n)$, outputs the $i$-th bit of the configuration reached by $M(x)$ after $t$ steps. Furthermore, we can implement $M'$ so that it runs in time $2^{O(p(n))}$ on input $(x, t, i)$, where $n$ is the length of $x$. Since $M'$ solves a decision problem in $\mathbf{EXP}$, there is a family of polynomial size circuits that simulate $M'$. Let $q(n)$ be a polynomial upper bound to the size of these circuits.

Our $\Sigma_2$ simulation of $M$, on input $x$, will “guess” a circuit $C$ of size $q(n)$, then it will verify that for every $i$ and $t$, the value of $C(x, t, i)$ is consistent with the (constant number of) values $C(x, t-1, \cdot)$ that it depends on. Finally, it will accept if and only if $C()$ predicts that after $2^p(n)$ steps $M(x)$ accepts.

$$x \in L \iff \exists C. |C| \leq q(|x|)$$
$$\forall t \leq 2^{p(|x|)}, i \leq c \cdot 2^p(x)$$
$$C(x, t, i) \text{ is consistent with } C(x, t-1, \cdot) \text{ and}$$
$$C(x, 2^p(|x|), \cdot) \text{ describes an accepting configuration}$$

(A few details are missing, for example one needs to treat the case $t = 0$ separately.)

8. Prove that $\mathbf{ZPP} = \mathbf{RP} \cap \text{coRP}$.

**Solution Sketch.** We gave the proof in class.

9. Show that if $\mathbf{NP} \subseteq \mathbf{BPP}$ then $\mathbf{NP} = \mathbf{RP}$.

**Solution Sketch.** It is enough to show that if $\mathbf{NP} \subseteq \mathbf{BPP}$ then $\text{3SAT} \in \mathbf{RP}$. Let $A$ be a $\mathbf{BPP}$ algorithm for $\text{3SAT}$. Given a formula $\varphi$ with $n$ variables, we first run $A$ on $\varphi$. If $A$ rejects, we reject. Otherwise, we try to construct a satisfying assignment for $\varphi$ one variable at a time. That is, we try instantiating $x_1$ to 0, and then use $A$ to decide if the resulting formula is satisfiable: if so, then we permanently set $x_1$ to 0 and proceed with $x_2$; otherwise we set $x_1$ to 1 and proceed with $x_2$. If we manage to construct a satisfying assignment we accept, otherwise we reject, and so on. If $\varphi$ is unsatisfiable, then we always reject. If $\varphi$ is satisfiable, then we construct a satisfying assignment and accept provided that the $n+1$ invocations of $A$ that we make are all correct. We can ensure that this happens with high probability by replacing each invocation of $A$ with $O(\log n)$ independent ones and taking the majority answer.
10. Prove that \( \text{SPACE}(O(n^{\log n})) \not\subseteq \text{BPP} \).

**Solution Sketch.** Consider the language \( L \) that contains all pairs \((M, x)\), where \( M \) is a probabilistic machine, such that
\[
\Pr[M((M), x) \text{ rejects within } n^{(\log n)/3} \text{ steps } ] \leq \frac{1}{2}
\]
We see that \( L \in \text{SPACE}(O(n^{\log n})) \) and we reach a contradiction if we assume \( L \in \text{BPP} \).

11. Change the assumption of Theorem 12 in the notes of Lecture 8 to having a probabilistic polynomial time algorithm that on input a formula with exactly one satisfying assignment finds that assignment with probability at least 1/2. Prove that it still follows that \( \text{NP} = \text{RP} \).

**Solution Sketch.** Use the same proof: the probability of finding an assignment becomes 1/16 instead of 1/8, and it is enough to set \( t = 11 \).

12. Let \( \{X_n\}_{n \geq 1} \) and \( \{Y_n\}_{n \geq 1} \) be ensembles (sets) of random variables, where \( X_n \) and \( Y_n \) take values over \( \{0, 1\}^n \). Say that \( \{X_n\} \) and \( \{Y_n\} \) are indistinguishable if for every two polynomials \( p \) and \( q \) and for every large enough \( n \) we have that \( X_n \) and \( Y_n \) are \( (p(n), 1/q(n)) \)-indistinguishable.

Prove that if \( \{X_n\} \) and \( \{Y_n\} \) are computationally indistinguishable, and \( f \) is a length-preserving (meaning that the length of the output is always equal to the length of the input) polynomial time computable function, then \( \{f_n(X_n)\} \) and \( \{f_n(Y_n)\} \) are also computationally indistinguishable.

**Solution Sketch.** Let \( t(n) \) be a polynomial upper bound to the size of a circuit that computes \( f_n \). Suppose that \( \{f_n(X_n)\} \) and \( \{f_n(Y_n)\} \) are not computationally indistinguishable. Then there are polynomials \( p(n) \) and \( q(n) \) such that, for infinitely many \( n \), \( f_n(X_n) \) and \( f_n(Y_n) \) are not \( (p(n), 1/q(n)) \)-indistinguishable, that is, there is a circuit \( C_n \) of size \( \leq p(n) \) such that
\[
| \Pr[C_n(f_n(X_n)) = 1] - \Pr[C_n(f_n(Y_n))] | \geq 1/q(n) .
\]
Considering that \( C_n(f_n(\cdot)) \) is computable by a circuit of size \( t(n) + p(n) \), we deduce that there are infinitely many \( n \) such that \( X_n \) and \( Y_n \) are not \( (t(n) + p(n), 1/q(n)) \)-indistinguishable, and so \( \{X_n\}_{n \geq 1} \) and \( \{Y_n\}_{n \geq 1} \) are not indistinguishable.

13. Prove that there is an ensemble \( \{X_n\} \) that is computationally indistinguishable from the ensemble of uniform distributions \( \{U_n\} \), even though only \( n^{\log n} \) elements of \( \{0, 1\}^n \) have non-zero probability in \( X_n \).
Solution Sketch. For every sufficiently large \( n \) we show that there is a multi-set \( S_n \) of size \( n^{\log n} \) such that the uniform distribution over \( S \) is \( (n^{(\log n)/4}, n^{-(\log n)/4}) \)-indistinguishable from uniform.

Fix a circuit \( C \), pick at random a multiset \( S \) by picking (with replacement) \( n^{\log n} \) from \( \{0, 1\}^n \). We want to compute

\[
\Pr_S[|\Pr_x \sim S[C(x) = 1] - \Pr[C(U_n) = 1]| \geq \epsilon]
\]

Define \( p := \Pr[C(U_n) = 1] \) and \( N = n^{\log n} \), then, equivalently, we want to compute

\[
\Pr_x[|\{x \in S : C(x) = 1\}| - p \cdot N| \geq \epsilon \cdot N].
\]

Define random 0/1 variables \( X_1, \ldots, X_n \) where \( X_i = 1 \) iff \( C(x_i) = 1 \), where \( x_i \) is the \( i \)-th element that we select to be in \( S \). Then the \( X_i \) are independent, and each of them has a probability \( p \) of being 1. If then follows from Chernoff bounds that

\[
\Pr_x[|\{x \in S : C(x) = 1\}| - p \cdot N| \geq \epsilon \cdot N] \leq e^{-\Omega(\epsilon^2 N)}.
\]

In particular, if we fix \( \epsilon = n^{-(\log n)/4} \), we have

\[
\Pr_S[|\Pr_x \sim S[C(x) = 1] - \Pr[C(U_n) = 1]| \geq n^{-(\log n)/4}] \leq e^{-\Omega(n^{(\log n)/2})}
\]

The number of circuits of size \( n^{(\log n)/4} \) is \( e^{O((\log n)^2 \cdot n^{(\log n)/4})} \) which is much smaller than \( e^{\Omega(n^{(\log n)/2})} \), so even after taking a union bound we have

\[
\Pr_S[\exists C. \text{size}(C) \leq n^{(\log n)/4} \land |\Pr_x \sim S[C(x) = 1] - \Pr[C(U_n) = 1]| \geq n^{-(\log n)/4}] < 1
\]

In particular, there exists a set \( S \) such that

\[
\forall C. \text{size}(C) \leq n^{(\log n)/4}, |\Pr_x \sim S[C(x) = 1] - \Pr[C(U_n) = 1]| \leq n^{-(\log n)/4}
\]

and we let \( X_n \) be the uniform distribution over such a set \( S \).

14. Prove that if pseudorandom generators of stretch \( 2n \) exist, then one-way functions exist.

Solution Sketch. Let \( G \) be a pseudorandom generator of stretch \( 2n \), we prove that it is a one-way function. If it were not a one-way function, then there would a family of polynomial size circuits \( C_n \) and a polynomial \( q(n) \) such that for infinitely many \( n \)

\[
\Pr_x[C_n(G(x)) = x' : G(x') = G(x)] \geq 1/q(n)
\]

Construct now a family of circuits \( C_n' \), still of polynomial size, such that \( C_n'(y) = 1 \) if and only if \( G(C_n(y)) = y \). Then we have \( \Pr[C_n'(G(x)) = 1] \geq 1/q(n) \), while \( \Pr[C_n'(y) = 1] \leq 1/2^n \), contradicting the assumption that \( G \) is a pseudorandom generator.

15. Prove that if a permutation \( f \) has a hard-core predicate \( B \), then \( f \) is a one-way permutation.
Solution Sketch. Let $f$ be a permutation and $B$ be a hard-core predicate for $f$. Suppose that $f$ is not one-way, then there are polynomials $p()$ and $q()$ such that for infinitely many $n$ there is a circuit $C_n$ such that

$$\Pr[C_n(f(x)) = x] \geq 1/q(n)$$

Consider now the following probabilistic process: on input $y$ of length $n$, we compute $x' = C_n(y)$. If $f(x') = y$ then we output $B(x')$, otherwise we output a random bit. The output of this process, with probability at least $1/2 + 1/2q(n)$, correctly equals $B(f^{(-1)}(y))$, and there is a fixed choice for the final random choice such that the correctness probability is at least as good. For that fixed choice, the above described process can be realized by a polynomial size circuit. This proves that $B$ is not a hard-core predicate for $f$, and we reach a contradiction.

16. Prove that if $P = NP$ then there cannot be any pseudorandom generators, even of stretch $n + 1$.

Solution Sketch. Suppose $P = NP$ and let $\{G_n\}$ be a family of polynomial time computable functions mapping $n$ bits into $n + 1$ bits. Let $C_n$ be a circuit that on input $y$ outputs $1$ if $y = G_n(x)$ for some $x$ and $0$ otherwise. Since the family $\{C_n\}$ solves a problem in $NP$, that, by assumption, can be solved in polynomial time, $C_n$ can be realized as a family of polynomial size circuits.

We have $\Pr_x[C_n(G(x)) = 1] = 1$, but $\Pr_r[C_n(r) = 1] \leq 1/2$ because each string $r$ has probability $1/2^{n+1}$ but only $2^n$ of them are possible outputs of $G_n$. So we have

$$|\Pr[C_n(G_n(x)) = 1] - \Pr[C_n(r) = 1]| \geq 1/2$$

where the $C_n$ have polynomial size, and $\{G_n\}$ cannot be a pseudorandom generator.


[Han90] Thomas Hancock. Identifying \( \mu \)-decision trees with queries. In 3rd COLT, pages 23–37, August 1990.

[Han91] Thomas Hancock. Learning \( 2\mu \) DNF and \( k\mu \) decision trees. In 4th COLT, pages 199–209, August 1991.
$g$ such that
\[ \forall x \in Z^n_2 \quad g(x) = f(x) \]
and the algorithm runs in time polynomial in $n$, and $2^d$.

An interesting special case is when the depth of the tree is logarithmic in $n$. In such a case, the algorithm will run in \textit{polynomial} time.

6 Extensions and Open Problems

The characterization of the decision trees can be extended easily to boolean functions of the form $f : \{0,1,\ldots,k-1\}^n \to \{0,1\}$ that can be computed by a polynomial-size $k$-ary decision tree. Namely, a tree in which each inner node $v$ has $k$ outgoing edges. When the computation arrives the node $v$, labeled by $S_v \in \{0,1,\ldots,k-1\}^n$, it assigns this node the value $\sum_{i=1}^n S_i \cdot x_i \mod k$, and the computation continues to the appropriate child of $v$. For extending the results to such functions and decision trees we have to define the appropriate characters and modify the proofs accordingly. For each $z \in \{0,1,\ldots,k-1\}^n$, define the basis function $\chi_z$:

\[
\chi_z(x_1, \ldots, x_n) \triangleq w^{z_1 \cdot x_1 + \ldots + z_n \cdot x_n},
\]

where $w = e^{2\pi i / k}$ is the root of unity of order $k$. In this case, a straightforward extension of our proof for $k = 2$ shows that the sum of the magnitudes of the coefficients is bounded by the number of leaves.

Another issue is decision trees with real outputs, where the leaves have real values from the interval $[0,M]$, i.e. $f : \{0,1\}^n \to [0,M]$. In a similar way to the boolean case, one can show that any function $f$ that has a real decision tree with $m$ leaves then $L_1(f) \leq mM$. In this case the running time of the learning algorithm would be polynomial in $M$.

An open problem related to this work is to find other classes of functions that can be learned in polynomial-time. In particular, it is very interesting whether functions that can be represented by a polynomial-size DNF formula can be learned in polynomial-time. One possible direction to resolve this open problem is to show that for any polynomial size DNF there is a polynomially sparse function that approximates it in $L_2$. So far we have not found any counter examples to this claim.

While our algorithm can be derandomized in the case of functions with polynomial $L_1$-norm, it is an open problem to derandomize it in the more general case of functions that can be approximated by polynomially sparse functions.

References

where \( I(S) \) is the indicator vector of the set \( S \), e.g. \( I(\{j\}) \) is equal to the vector who has the 
\( j \)th coordinate one and all the other coordinates zero. From the expansion of \( h_i \) it is clear that 
\[ |\hat{f}(z)| = 2^{-\ell}, \] for any \( z \), and therefore \( L_1(f) = 2^{d-2}\ell = 2^{\ell} \). We will show that there is a decision 
tree with linear operations of size \( O(2^\ell) \) that computes \( f \).

The following is a description of the decision tree that computes \( f \). The first \( \ell \) levels of the 
decision tree form a complete binary tree. In each node of level \( i \) \((1 \leq i \leq \ell)\) we test \( x_i \oplus x_{\ell+i} \). 
For every leaf \( v \) of the tree, let \( b_1^v, \ldots, b_\ell^v \) be the sequence of the replies to the queries \( x_i \oplus x_{\ell+i} \), 
along the path from the root of the tree to \( v \). Let, \( S_v = \{ i | b_i^v = 0 \} \). We now test for the parity 
of all \( x_i \)’s with \( i \in S_v \). Let the value of the computation be the value of the parity. The tree has 
only depth \( \ell + 2 \), and hence only \( O(2^\ell) \) nodes. The reason that it computes the inner product 
correctly is the following. If \( b_i^v = 1 \), then exactly one of \( x_i, x_{\ell+i} \) is 0 and in particular \( x_i x_{\ell+i} = 0 \). 
This implies that the \( i \)-th term in the inner product is zero, and therefore we can ignore it. 
If \( b_i^v = 0 \), then either both \( x_i, x_{\ell+i} \) are 0 or both \( x_i, x_{\ell+i} \) are 1. In both cases, \( x_i x_{\ell+i} = x_i \). 
Therefore, instead of considering the value of the \( i \)-th term (i.e., \( x_i x_{\ell+i} \)), we can consider the 
variable \( x_i \). Therefore the parity of \( S_v \) is the parity of all the relevant terms.

**Exact Reconstruction**

We show that a boolean decision trees with linear operations can be recovered exactly in time 
polynomial in \( n \) and \( 2^d \), where \( d \) is the depth of the tree.

It follows from the proof of Lemma 5.1 that all the coefficients of a tree of depth \( d \), can be 
written as \( k/2^d \), where \( k \) is an integer in the range \([-2^d, +2^d] \). The idea is to first find a good 
approximation of all the non-zero coefficients and then, using the above fact, to compute them 
**exactly**.

By Theorem 4.12, we have a *deterministic* algorithm that for every function \( f \) and \( \varepsilon > 0 \) 
outputs a function \( g \) such that 
\[
\sum_z (\hat{f}(z) - \hat{g}(z))^2 = E[(f - g)^2] \leq \varepsilon
\]
in time polynomial in \( n, L_1(f) \) and \( 1/\varepsilon \). In particular, it follows that 
\[ |\hat{f}(z) - \hat{g}(z)| \leq \sqrt{\varepsilon}, \] for every \( z \). We use this algorithm, with \( \varepsilon < \left( \frac{1}{2^{2d}} \right)^2 \), which ensures that \( g \) satisfies 
\[ |\hat{f}(z) - \hat{g}(z)| < \frac{1}{2^{2d}}, \] for every \( z \). Since the real coefficient is of the form \( k/2^d \), where \( k \) is integer, the difference 
between possible values that a coefficient can have is \( 1/2^d \); since the error is smaller than \( 1/2^{2d} \), 
by rounding we find the exact coefficient.

This implies that we recovered all the Fourier coefficients of the function *exactly*. Therefore, 
we found a function whose Fourier transform is identical to the tree’s Fourier transform, this 
implies that the two functions are identical. By the choice of \( \varepsilon \) and as \( L_1(f) \leq m \leq 2^{d+1} \), 
the running time of the algorithm is polynomial in \( n \) and \( 2^d \). Thus, we have established the following 
theorem,

**Theorem 5.3** There is a (deterministic) polynomial time algorithm, that for any boolean function \( f \) that can be represented by a depth \( d \) decision tree with linear operations, outputs a function
vectors that are a linear combination of the $y_i$'s. (Note that we consider $z = \vec{0}$ as a linear combination of the $y_i$'s.) On the other hand, if $z \neq \vec{0}$ is not a linear combination of the $y_i$'s then the number of $x \in I(v)$ satisfying $x \odot z = 0$ is the same as the number of $x \in I(v)$ satisfying $x \odot z = 1$. Therefore, in this case $E_{x \in I(v)}[\chi_z(x)] = 0$. Combining the two claims, we have that

$$\sum_z |E_{x \in I(v)}[\chi_z]| = 2^{d(v)}.$$

Intuitively, each leaf $v$ contributes to at most $2^{d(v)}$ coefficients, and to each coefficient it contributes $2^{-d(v)}$. This implies that leaf $v$ contributes at most one to the sum of the absolute value of all the coefficients. Therefore, $L_1(f)$ is bounded by the number of leaves which is at most $m$. The following calculations shows this formally:

$$L_1(f) = \sum_{z \in \{0,1\}^n} |\hat{f}(z)| = \sum_{z \in \{0,1\}^n} |E[f\chi_z]| = \sum_{z \in \{0,1\}^n} \left| \sum_{v \in \text{leaf}(T_f)} 2^{-d(v)}v\text{al}(v)E_{x \in I(v)}[\chi_z] \right|$$

$$\leq \sum_{z \in \{0,1\}^n} \sum_{v \in \text{leaf}(T_f)} 2^{-d(v)} |E_{x \in I(v)}[\chi_z]|$$

$$= \sum_{v \in \text{leaf}(T_f)} 2^{-d(v)} \sum_{z \in \{0,1\}^n} |E_{x \in I(v)}[\chi_z]|$$

$$= \sum_{v \in \text{leaf}(T_f)} 2^{-d(v)} 2^{d(v)}$$

$$= |\text{leaf}(T_f)| \leq m.$$

Combining Lemma 5.1 and Claim 2.1 with Theorem 4.12, we get the following theorem:

**Theorem 5.2** There is a polynomial time (deterministic) algorithm, that for any boolean function $f$ that can be represented by an $m$ node decision tree with linear operations, and for any $\varepsilon > 0$, outputs a function $g$ such that

$$\Prob[f \neq \text{sign}(g)] \leq \varepsilon,$$

and the algorithm runs in time polynomial in $n$, $m$, and $1/\varepsilon$.

We now show that the bound given in Lemma 5.1 is tight. Consider the inner product function on inputs of $n = 2\ell$ variables:

$$f(x_1, \ldots, x_{2\ell}) = (-1)^{x_1 x_{\ell+1} \cdots x_{\ell} x_{2\ell}} = \prod_{i=1}^{\ell} (-1)^{x_1 x_{i+1}}.$$

Let $h_i(x_1, \ldots, x_{2\ell}) = (-1)^{x_i x_{i+1}}$, for $1 \leq i \leq \ell$. Clearly, $f = \prod_{i=1}^{\ell} h_i$. The Fourier transform of $h_i$ is

$$h_i(x) = \frac{1}{2} + \frac{1}{2} \chi_{I(i)}(x) + \frac{1}{2} \chi_{I(i+\ell)}(x) - \frac{1}{2} \chi_{I(i,i+\ell)}(x),$$

$$18$$
5 Decision trees

We consider decision trees, whose input is $n$ boolean variables, and the branching in each node is based on a linear combination (over $GF(2)$) of a subset of the variables (as described in section 2.2). In this section we show that for any function corresponding to such a boolean decision tree, the sum of the absolute values of its coefficients (i.e., the $L_1$ norm) is bounded by $m$, the number of nodes in the tree. This implies, using the result in the previous section, that such decision trees can be approximated in polynomial time.

Lemma 5.1 Let $f$ be computed by a decision tree with $m$ nodes, then $L_1(f) \leq m$.

Proof: A non-redundant decision tree is a tree in which for every leaf there is some input that ends up in that leaf. By the claim of the lemma, $f$ has a decision tree with $m$ nodes, therefore there is a non-redundant decision tree $T_f$, with at most $m$ nodes, that computes $f$. Denote by leaf$(T_f)$ the set of leaves in the tree $T_f$, and by $d(v)$ the depth of node $v$. That is, $d(v)$ is the number of nodes on the path from the root to node $v$, not including $v$.

We claim that every node at depth $d$ has exactly a $2^{-d}$ fraction of the inputs reaching it. The inputs that reach a node at depth $d$ pass through $d$ internal nodes; therefore, they satisfy a set of $d$ linear constraints over $GF(2)$. Each such linear constraint is satisfied by exactly $\frac{1}{2}$ of the inputs. Since $T_f$ is non-redundant, the linear constraints are linearly independent. Therefore, the fraction of the inputs that satisfy all the $d$ constraints is $2^{-d}$.

By the definition of a decision tree each input reaches a unique leaf. Let $I(v)$ be the set of all the inputs that reach leaf $v$. Then for every $z$,

$$\hat{f}(z) = \langle f, \chi_z \rangle = E[f\chi_z] = \sum_{v \in \text{leaf}(T_f)} 2^{-d(v)} \text{val}(v) E_{x \in I(v)}[\chi_z(x)].$$

In the following we show that $|E_{x \in I(v)}[\chi_z(x)]| = 1$ for exactly $2^{d(v)}$ values of $z$, and zero for the rest. This implies that each leaf can contribute at most one to the value of $L_1(f)$.

Consider a leaf $v$. Any input $x \in I(v)$ satisfies $d(v)$ linear constraints, i.e.

$$x \odot y_1 = b_1,$$

$$x \odot y_2 = b_2,$$

$$\vdots$$

$$x \odot y_{d(v)} = b_{d(v)},$$

where $\odot$ denotes the inner product of two $n$-bit vectors $x, y \in \{0,1\}^n$, i.e. $x \odot y = \sum_i x_i y_i \text{ mod } 2$.

The argument has two parts, depending on whether or not $z$ is a linear combination of the $y_i$’s. If $z$ is a linear combination of the $y_i$’s then clearly the value of $x \odot z$ is fixed, for every $x \in I(v)$. Since the value of $x \odot z$ is fixed, by definition, the value of $\chi_z(x)$ is fixed to either $+1$ or $-1$, hence $|E_{x \in I(v)}[\chi_z(x)]| = 1$. Since the tree is non-redundant there are exactly $2^{d(v)}$
Proof: Let \( h_x(y) \overset{\Delta}{=} f(yx) \) and \( g_x(y) \overset{\Delta}{=} h_x(y) \chi_x(y) \). By Lemma 4.6 \( L_1(h_x) \leq L_1(f) \). First, we show that \( L_1(g_x) = L_1(h_x) \) by showing that they have the same set of coefficients:

\[
\hat{g}_x(z) = \langle g_x, \chi_z \rangle = \langle h_x \chi_x, \chi_z \rangle = \langle h_x, \chi_x \chi_z \rangle = \langle h_x, \chi_x \chi_z \rangle = \hat{h}_x(\alpha \oplus z).
\]

This implies that \( L_1(g_x) \leq L_1(f) \). By Lemma 4.5,

\[
|E_\mu[g_x] - E_u[g_x]| \leq \lambda L_1(g_x) \leq \lambda L_1(f).
\]

\( \square \)

We now show a few basic relations about the \( L_1 \) norm of the coefficients of a function, so that we can show that \( L_1(f^2) \leq L_1^2(f) \).

Claim 4.8 For any function \( f \) and \( \alpha \in \{0, 1\}^k \), \( k \leq n \), then \( L_1(f_\alpha) \leq L_1(f) \).

This is because \( f_\alpha \), by definition, includes only a subset of the coefficients of \( f \). The second claim establishes a relation between the \( L_1 \) norm of two functions and the \( L_1 \) norm of their product.

Claim 4.9 For any functions \( g \) and \( h \), \( L_1(gh) \leq L_1(g)L_1(h) \).

Proof: Note that

\[
g(x)h(x) = \sum_{z_1, z_2 \in \{0, 1\}^n} \hat{g}(z_1) \hat{h}(z_2) \chi_{z_1}(x) \chi_{z_2}(x) = \sum_{z_1} \left( \sum_{z_3} \hat{g}(z_1) \hat{h}(z_3 \oplus z_1) \right) \chi_{z_2}(x).
\]

(To see the last transformation take \( z_3 \) to be \( z_1 \oplus z_2 \).) Therefore,

\[
L_1(gh) = \sum_{z_2} \left| \sum_{z_1} \hat{g}(z_1) \hat{h}(z_3 \oplus z_1) \right| \leq \sum_{z_1, z_2} |\hat{g}(z_1)||\hat{h}(z_2)| = L_1(g)L_1(h).
\]

\( \square \)

We use the above two claims to bound \( L_1(f^2) \).

Claim 4.10 For any function \( f \) and \( \alpha \in \{0, 1\}^k \), \( k \leq n \), then,

\[
L_1(f^2_\alpha) \leq L_1^2(f_\alpha) \leq L_1^2(f).
\]

This implies that we can compute \( B_\alpha \) (the outer loop of subroutine Approx) using only \( \lambda \)-bias distributions.

Lemma 4.11 For any function \( f \), and \( \alpha \in \{0, 1\}^k \), \( k \leq n \),

\[
|E_\mu[f^2_\alpha] - E_u[f^2_\alpha]| \leq \lambda L_1^2(f).
\]

Proof: Combine Lemma 4.5 with Claim 4.10. \( \square \)

Lemma 4.11 can be used to derandomize the outer loop, by choosing \( \lambda = \varepsilon / L_1^2(f) \). Lemma 4.7 can be used to derandomize the inner loop, by choosing \( \lambda = \varepsilon / L_1(f) \). This implies that we have established the following theorem.

Theorem 4.12 There is a deterministic algorithm, that receives as an input a boolean function \( f \), \( L_1(f) \) and \( \varepsilon > 0 \), and outputs a function \( g \) such that \( E[(f - g)^2] \leq \varepsilon \), and the algorithm runs in time polynomial in \( n \), \( L_1(f) \), and \( 1/\varepsilon \).
Clearly, if \( z = z' \) then \( \sum_x \chi_z(x)\chi_{z'}(x) = 2^n \), and if \( z \neq z' \) then \( \sum_x \chi_z(x)\chi_{z'}(x) = \sum_x \chi_z\chi_{z'}(x) = 0 \). Therefore the above sum equals
\[
2^n \sum_z \mu(z)\tilde{f}(z).
\]

As \( \hat{\mu}(0) = \frac{1}{2^n} \), the lemma follows. \( \square \)

Our goal now is to show that the algorithm behaves “similarly” when its coin tosses are chosen from the uniform distribution, \( u \), or from a \( \lambda \)-bias distribution, \( \mu \). We show it by proving that the \( A_i \)'s and the \( B_\alpha \) computed by Subroutine \textbf{Approx} are “similar”. The main tool for this is the following lemma.

**Lemma 4.5** Let \( f \) be any function, \( u \) be the uniform distribution, and \( \mu \) be a \( \lambda \)-bias distribution, then
\[
|E_\mu[f] - E_u[f]| \leq \lambda L_1(f).
\]

**Proof:** By definition, \( E_u[f] = \hat{f}(\emptyset) \). From Lemma 4.4 we have
\[
E_\mu[f] = \hat{f}(\emptyset) + 2^n \sum_{z \neq 0} \mu(z)\tilde{f}(z).
\]

The definition of \( \lambda \)-bias distributions ensures that \( |\hat{\mu}(z)| \leq \lambda/2^n \), therefore we get
\[
|E_\mu[f] - E_u[f]| \leq |\hat{f}(\emptyset) + 2^n \sum_{z \neq 0} \mu(z)\tilde{f}(z) - \hat{f}(\emptyset)| \leq 2^n \sum_{z \neq 0} \frac{\lambda}{2^n} |\tilde{f}(z)| = \lambda L_1(f),
\]

which completes the proof. \( \square \)

**Lemma 4.6** Let \( h(y) = f(yx) \), for some fixed \( x \in \{0,1\}^k \). Then, \( L_1(h) \leq L_1(f) \).

**Proof:** The proof is by induction \( k \). Let \( k = 1 \). Then \( h(y) = f(yb) \), where \( b \in \{0,1\} \). One can verify easily that if \( b = 0 \), then \( \hat{h}(z) = \hat{f}(z0) + \hat{f}(z1) \), and if \( b = 1 \), then \( \hat{h}(z) = \hat{f}(z0) - \hat{f}(z1) \). In both cases \( L_1(h) \leq L_1(f) \).

The induction step follows from the fact that we can restrict the function bit after bit. \( \square \)

This implies that we can compute the \( A_i \)'s (the inner loop of subroutine \textbf{Approx}) with \( \lambda \)-bias distributions.

**Lemma 4.7** Let \( u \) be the uniform distribution, and \( \mu \) be a \( \lambda \)-bias distribution on \( \{0,1\}^k \). For any function \( f, k \leq n, \alpha \in \{0,1\}^k \), and \( x \in \{0,1\}^{n-k} \),
\[
|E_{y \in \mu}[f(yx)\chi_\alpha(y)] - E_{y \in u}[f(yx)\chi_\alpha(y)]| \leq \lambda L_1(f).
\]

15
4.1 Derandomization

For functions with a “small” $L_1$ norm we can efficiently derandomize the algorithm. One drawback of the derandomization is that it requires that we have a bound on the $L_1$-norm, since we cannot test hypotheses using randomization as before. The main idea in the derandomization is the usage of $\lambda$-bias distributions. The notion of a $\lambda$-bias distribution was first suggested by [NN90], and other constructions were given later by [AGHP90]. One way to formalize the notion of $\lambda$-bias is the following.

**Definition 4.1** Every distribution $\mu$ over $\{0, 1\}^n$ can be considered as a real function $\mu(x) = \sum_z \hat{\mu}(z) \chi_z(x)$. A distribution $\mu(x)$ is $\lambda$-bias if for any $z \neq \overline{0}$, $|\hat{\mu}(z)| \leq \lambda 2^{-n}$.

Note that the uniform distribution $u(x) = \frac{1}{2^n}$ has $\hat{u}(z) = 0$, for $z \neq \overline{0}$, and therefore it is 0-bias. Also for any distribution $\mu$,

$$\hat{\mu}(\overline{0}) = \langle \mu, \chi_0 \rangle = E[\mu] = \frac{1}{2^n} \sum_x \mu(x) = \frac{1}{2^n}.$$  

One of the applications of $\lambda$-bias distributions is to derandomize algorithms. The derandomization of an algorithm is done by showing that the output of the algorithm when its coin tosses are chosen from a uniform distribution, and the output of the algorithm when its coin tosses are chosen from a $\lambda$-bias distribution are very similar. If this holds, then the deterministic algorithm is the following: (1) enumerate all the strings in the $\lambda$-bias distribution, (2) for each such string compute the value of the randomized algorithm, and (3) output the average of the replies in step (2). To have an efficient derandomization we would like that the sample space of the $\lambda$-bias probability distribution would be enumerable “efficiently” (in particular, it has to be “small”).

**Theorem 4.3** ([NN90, AGHP90]) There are $\lambda$-bias distributions whose sample spaces are of size $\left(\frac{n}{\lambda}\right)^2$ and are constructible in polynomial time.

Using the definition (and basic properties) of the Fourier transform we show the following identity.

**Lemma 4.4** For any function $f$ and any distribution $\mu$,

$$E[\mu][f] = \hat{f}(\overline{0}) + 2^n \sum_{z \neq \overline{0}} \hat{\mu}(z) \hat{f}(z).$$

**Proof:** By the definitions,

$$E[\mu][f] = \sum_x \mu(x) f(x) = \sum_x \left( \sum_z \hat{\mu}(z) \chi_z(x) \sum_{z'} \hat{f}(z') \chi_{z'}(x) \right).$$
4 Functions with small $L_1$ norm

In this section we show that a function whose sum of the absolute value of the coefficients is “small” has a “small” number of “large” coefficients that “almost” determine the function. Therefore, in order to get a good approximation of the function, it is sufficient to approximate those coefficients. Saying it differently, we show that functions with “small” $L_1$ norm can be approximated by sparse functions.

Let $f$ be a boolean function, and recall that $L_1(f) \triangleq \sum_z |\hat{f}(z)|$. The following lemma shows that it is sufficient to approximate a small number of the (large) coefficients of $f$.

**Lemma 4.1** Let $\varepsilon > 0$. Let $S = \{ z : |\hat{f}(z)| \geq \varepsilon / L_1(f) \}$, and let $g(x) = \sum_{z \in S} \hat{f}(z) \chi_z(x)$. Then

$$E[(f - g)^2] \leq \varepsilon$$

**Proof:** By the definition of $g$, we have

$$(f - g)(x) = \sum_z (\hat{f}(z) - \hat{g}(z)) \chi_z(x) = \sum_{z \not\in S} \hat{f}(z) \chi_z(x).$$

Therefore, using Parseval’s identity, we have

$$E[(f - g)^2] = \sum_{z \not\in S} \hat{f}^2(z).$$

This is clearly bounded above by

$$\left( \max_{z \in S} |\hat{f}(z)| \right) \cdot \left( \sum_{z \in \{0,1\}^n} |\hat{f}(z)| \right) \leq \frac{\varepsilon}{L_1(f)} \cdot L_1(f) = \varepsilon.$$

This implies that if we can find all the coefficients that are greater, in absolute value, than $\varepsilon / L_1(f)$, we can approximate $f$. The procedure in the previous section gives a way to find all such coefficients in time $\text{poly}(n, L_1(f), 1/\varepsilon, \log 1/\delta)$. Note that in order to use subroutine $\text{Coeff}$ we need to know the value of $L_1(f)$. If this is not the case we can search for an upper bound on it. This will add a multiplicative factor of $O(\log L_1(f))$ to the time complexity. We have established the following theorem.

**Theorem 4.2** There is a randomized algorithm, that for any boolean function $f$, and $\varepsilon, \delta > 0$, outputs a function $g$ such that $\text{Prob}[E[(f - g)^2] \leq \varepsilon] \geq 1 - \delta$, and the algorithm runs in time polynomial in $n, L_1(f), 1/\varepsilon$ and $\log 1/\delta$. 

13
Lemma 3.9 Let \( m_1 = \Theta\left(\frac{1}{\delta^2} \log \frac{n}{\delta^2}\right) \) and \( m_2 = \Theta\left(\frac{1}{\delta} \log \frac{n m_1}{\delta}\right) \). With probability \( 1 - \delta \) the procedure Coef outputs all the coefficients \( z \), such that \( |f(z)| \geq \theta \), and does not output any coefficient \( z \), such that \( |\hat{f}(z)| \leq \theta/2 \).

**Proof:** As shown above we have \( O(n/\theta^2) \) calls to the subroutine Coef (and the same number of calls to the subroutine Approx). To guarantee a total error probability of \( \delta \) we choose \( m_1 \) and \( m_2 \) so that the probability of error in each of the calls is no more than \( \frac{\delta^2}{n} \). Also recall that if \( z = \alpha \beta \), for some \( \alpha \in \{0, 1\}^k \) and \( \beta \in \{0, 1\}^{n-k} \), and if \( |\hat{f}(z)| \geq \theta \), then \( E[f_\alpha^2] \geq \theta^2 \).

Consider an \( \alpha \) such that \( E[f_\alpha^2] \geq \theta^2 \). By Lemma 3.6, with probability at least \( 1 - 2e^{-m_1 \theta^2/48} \), the value of \( B'_\alpha \geq \frac{3}{4} \theta^2 \). By Lemma 3.7, with probability \( 1 - 2m_1 e^{-\theta^2 m_2/2^9} \), all the values of \( A_i \) (\( 1 \leq i \leq m_1 \)) satisfy \( |f_\alpha(x_i) - A_i| \leq \theta^2/16 \). In this case, using Lemma 3.8, \( B_\alpha \geq \frac{19}{16} \theta^2 > \frac{\theta^2}{2} \).

Consider an \( \alpha \) such that \( E[f_\alpha^2] \leq \theta^2/4 \). Note that \( B_\alpha \) is monotonic in \( E[f_\alpha^2] \) and therefore it is enough to consider the case \( E[f_\alpha^2] = \theta^2/4 \). By Lemma 3.6, with probability \( 1 - 2e^{-m_1 \theta^2/192} \), the value of \( B'_\alpha \leq \frac{5}{16} \theta^2 \). By Lemma 3.7, with probability \( 1 - 2m_1 e^{-\theta^2 m_2/2^9} \), all the values of \( A_i \) (\( 1 \leq i \leq m_1 \)) satisfy \( |f_\alpha(x_i) - A_i| \leq \theta^2/16 \). In this case, using Lemma 3.8, \( B_\alpha \leq \frac{7}{16} \theta^2 < \frac{\theta^2}{2} \).

So far we have shown that the algorithm performs the “right” recursive calls. This implies that, with probability \( 1 - \delta \), the number of recursive calls is at most \( 4n/\theta^2 \). It also implies that all the required coefficients will be output. Now we need to show that in such a case no coefficient \( z \), such that \( |\hat{f}(z)| \leq \theta/2 \) is output. Note that the probability of outputting such a coefficient is at least the probability that we made one “wrong” recursive call, and this probability is bounded by \( \delta \).

Once the procedure outputs the list of vectors, \( z_1, \ldots, z_t \), we can approximate each coefficient \( \hat{f}(z_i) \). Let the approximate value be \( \gamma_i \). (Since by definition, \( \hat{f}(z_i) = E[f \chi_{z_i}] \) then Lemma 3.5 guarantees that a “small” sample will give with a high probability a “good” approximation for all these coefficients.) The prediction hypothesis is \( h(x) = \sum_{i=1}^t \gamma_i \chi_{z_i}(x) \). To conclude, the algorithm has the following performance:

**Theorem 3.10** There is a randomized algorithm, that for any boolean function \( f \), any \( \delta > 0 \), and any \( \theta > 0 \) outputs a list of vectors \( \alpha_i \in \{0, 1\}^n \) such that

- with probability \( \geq 1 - \delta \) the list contains every vector \( \alpha \) for which \( |\hat{f}(\alpha)| \geq \theta \) and does not contain any vector \( \alpha \) for which \( |\hat{f}(\alpha)| \leq \theta/2 \). (This implies that the list may contain at most \( 4n/\theta^2 \) vectors.)

- the algorithm runs in time polynomial in \( n, 1/\theta \) and \( \log 1/\delta \).

To summarize, in this section we have shown that if \( f \) can be \( \varepsilon \)-approximated by a \( t \)-sparse function, then it is sufficient to find all its coefficients larger than \( \varepsilon/t = \theta \). Therefore we have established the following theorem.

**Theorem 3.11** Let \( f \) be a boolean function such that there exists a \( t \)-sparse function \( g \) that \( \varepsilon \)-approximates \( f \) (in norm \( L_2 \)). Then there exists a randomized algorithm, that on input \( f \) and \( \delta > 0 \) outputs a function \( h \), such that with probability \( 1 - \delta \) the function \( h \) \( O(\varepsilon) \)-approximates, in norm \( L_2 \), the input function \( f \). The algorithm runs in time polynomial in \( n, t, 1/\varepsilon \) and \( \log 1/\delta \).
compute $B_\alpha$ with the $A_i$’s (instead of $f_\alpha(x_i)$) it is still a “good” approximation of $E_\varepsilon[f_\alpha^2(x)]$. For the proof we use Chernoff bounds (see [HR89]):

**Lemma 3.5 (Chernoff)** Let $X_1, \ldots, X_m$ be independent, identically distributed random variables, such that $E[X_i] = p$ and $S_m = \sum_{i=1}^{m} X_i$.

- If $X_i \in [0, 1]$ then
  \[
  \text{Prob} \left[ (1 - \varepsilon) \cdot p \leq \frac{S_m}{m} \leq (1 + \varepsilon) \cdot p \right] \geq 1 - 2e^{-\varepsilon^2 mp/3}
  \]

- If $X_i \in [-1, +1]$ then
  \[
  \text{Prob} \left[ \left| \frac{S_m}{m} - p \right| \geq \lambda \cdot \frac{1}{\sqrt{m}} \right] \leq 2e^{-\lambda^2/2}.
  \]

Using this bound, we claim that by choosing at random $m_1$ values $x_i$ and computing the average $f_\alpha^2(x_i)$, we get a value that is very close to $E[f_\alpha^2]$.

**Lemma 3.6** Let $B'_\alpha = \frac{1}{m_1} \sum_{i=1}^{m_1} f_\alpha^2(x_i)$, where $x_i \in \{0, 1\}^{n-k}, 1 \leq i \leq m_1$, are chosen uniformly at random. Then,

\[
\text{Prob} \left[ \frac{3}{4} E[f_\alpha^2] \leq B'_\alpha \leq \frac{5}{4} E[f_\alpha^2] \right] \geq 1 - 2e^{-\frac{m_1}{16} E[f_\alpha^2]}
\]

**Proof:** Follows immediately from the first part of Lemma 3.5 with $\varepsilon = \frac{1}{4}$ (and $p = E[f_\alpha^2]$).

The next lemma claims that $A_i$ is a “good” approximation for $f_\alpha(x_i)$. It is based on the identity of Lemma 3.2 (i.e., $f_\alpha(x_i) = E_y[f(yx_i)\chi_\alpha(y)]$).

**Lemma 3.7** For any value of $x_i$,

\[
\text{Prob} \left[ |A_i - f_\alpha(x_i)| \geq \theta^2/16 \right] \leq 2e^{-\theta^4 m_2/16}
\]

**Proof:** Follows immediately from the second part of Lemma 3.5 with $\lambda = \frac{\theta^2 \sqrt{m_2}}{16}$.\qed

Intuitively, if we approximate each $f_\alpha(x_i)$ well, the difference between $B_\alpha$ (which uses the approximate values) and $B'_\alpha$ (which uses the true values) should be small. The following lemma formalizes this intuition.

**Lemma 3.8** If $|f_\alpha(x_i) - A_i| \leq \frac{\theta^2}{16}$, for $1 \leq i \leq m_1$, then $|B_\alpha - B'_\alpha| \leq \frac{2\theta^2}{16}$.

**Proof:** From Corollary 3.3 it follows that $|f_\alpha(x_i)| \leq 1$. From the definition of the $A_i$’s it follows that $|A_i| \leq 1$. Therefore,

\[
|B_\alpha - B'_\alpha| = \left| \frac{1}{m_1} \sum_{i=1}^{m_1} (f_\alpha^2(x_i) - A_i^2) \right| \leq \frac{1}{m_1} \sum_{i=1}^{m_1} |f_\alpha(x_i) - A_i| \cdot |f_\alpha(x_i) + A_i| \leq \frac{1}{m_1} \sum_{i=1}^{m_1} \frac{\theta^2}{16} \cdot 2 = \frac{2\theta^2}{16}
\]

Using the above lemmas, we now fix the values of $m_1$ and $m_2$ so that $B_\alpha$ will be a “good” approximation for $E[f_\alpha^2]$.

11
SUBROUTINE Coef(α)
  \( B_α = \text{Approx}(α) \) /* \( B_α \) is approximating \( E[f_α^2]\) */
  IF \( B_α \geq \theta^2/2 \) THEN
    IF \( |α| = n \) THEN OUTPUT \( α \)
    ELSE Coef(α0); Coef(α1);*

Figure 2: The Modification of Subroutine Coef

Detailed analysis

We are still not done, since this algorithm assumes that we can compute \( E[f_α^2] \) exactly, something that is not achievable in polynomial time. On the other hand, we can approximate \( E[f_α^2] \) very accurately in polynomial time. Therefore we modify Subroutine Coef: instead of testing whether \( E[f_α^2] \geq \theta^2 \) we approximate \( E[f_α^2] \) and test whether the approximated value is greater than \( \theta^2/2 \) (see Figure 2).

The approximation of \( E[f_α^2] \) is such that with very high probability the error in the approximation is small. That is, with high probability, every coefficient satisfying \( |\hat{f}(z)| \geq \theta \) will be output, which guarantees the correctness condition of the algorithm. Also, this approximation guarantees that with high probability, no coefficient satisfying \( |\hat{f}(z)| \leq \theta/2 \) will be output, which bounds (by lemma 3.4) the number of coefficients the algorithm outputs to at most \( 4/\theta^2 \). Moreover, it implies that for every \( k \) at most \( 4/\theta^2 \) strings \( α \in \{0,1\}^k \) of length \( k \) will pass the test of the subroutine, which bounds the number of calls to the recursive subroutine by \( O(n/\theta^2) \).

What we are left with is to bound the computation required to approximate \( E[f_α^2] \).

Let \( m_1, m_2 \) be parameters (to be fixed later). We approximate \( E[f_α^2(x)] \) as follows.

SUBROUTINE Approx(α)
  Choose at random \( x_i \in \{0,1\}^{n-k} \), for \( 1 \leq i \leq m_1 \).
  For each \( x_i \):
    Choose at random \( y_{i,j} \in \{0,1\}^k \), for \( 1 \leq j \leq m_2 \).
    Let \( A_i = \frac{1}{m_2} \sum_{j=1}^{m_2} \hat{f}(y_{i,j}x_i)\chi_α(y_{i,j}). \) /* \( A_i \) is approximating */
    \( * \ f_α(x_i) = E_y[\hat{f}(y|x_i)\chi_α(y)]. * / \)
  Let \( B_α = \frac{1}{m_1} \sum_{i=1}^{m_1} A_i^2. \) /* \( B_α \) is approximating \( E_x[f_α^2(x)] \). */
  RETURN \( B_α \).

The value of \( B_α \) is the approximation to \( E[f_α^2(x)] \). We now need to find the “right” values of \( m_1 \) and \( m_2 \), such that \( B_α \) will be a “good” approximation to \( E[f_α^2(x)] \). That is, with high probability, if \( E[f_α^2] \geq \theta^2 \) then \( B_α \geq \theta^2/2 \) and if \( E[f_α^2] \leq \theta^2/4 \) then \( B_α < \theta^2/2 \).

To prove that \( B_α \) is a “good” approximation of \( E_x[f_α^2(x)] \), we first prove that \( B_α \) would be a “good” approximation of \( E_x[f_α^2(x)] \) if we compute it with the real values of \( f_α(x_i) \). Then we show that the \( A_i \)'s are “good” approximations for \( f_α(x_i) \). Finally, we show that even if we
SUBROUTINE Coef(α)
    IF E[f_α^2] ≥ θ^2 THEN
        IF |α| = n THEN OUTPUT α
    ELSE Coef(α0); Coef(α1);

Figure 1: Subroutine Coef

We showed how to decompose a function f into functions f_α, α ∈ {0, 1}^k, such that each coefficient of f appears in a unique f_α. Recall that our aim is to find the coefficients f(z) such that |f(z)| ≥ θ. The next lemma claims that this cannot hold for “too many” values of z, and that the property E[f_α^2] ≥ θ^2 cannot hold for “many” α (of length k) simultaneously.

Lemma 3.4 Let f be a boolean function, and θ > 0. Then,

1. At most 1/θ^2 values of z satisfy |f(z)| ≥ θ.
2. For any 1 ≤ k < n, at most 1/θ^2 functions f_α with α ∈ {0, 1}^k satisfy E[f_α^2] ≥ θ^2.

Proof: By the assumption that f is a boolean function combined with Parseval’s equality, we get

\[ \sum_{z \in \{0, 1\}^n} \hat{f}(z) = E[f^2] = 1. \]

Therefore, (1) immediately follows. Similarly, using the definition of f_α,

\[ E[f_α^2] = \sum_{β \in \{0, 1\}^{n-k}} \hat{f}^2(αβ). \]

Thus, if |\hat{f}(αβ)| ≥ θ, for some β ∈ {0, 1}^{n-k}, then E[f_α^2] ≥ θ^2. By the above two equalities the following holds,

\[ \sum_{α \in \{0, 1\}^k} E[f_α^2] = E[f^2] = 1. \]

Therefore, at most 1/θ^2 functions f_α have E[f_α^2] ≥ θ^2, which completes the proof of (2).

The Algorithm

By now the algorithm for finding the large coefficients of a function f should be rather obvious. It is described by the recursive subroutine Coef, which appears in Figure 1. We start the algorithm by calling Coef(λ), where λ is the empty string.

As mentioned earlier in this section, we know that each coefficient of f_α appears in exactly one of f_α0 and f_α1, and also that if |\hat{f}(αβ)| ≥ θ, for some β ∈ {0, 1}^{n-k}, then E[f_α^2] ≥ θ^2 (note that when |α| = n, then E[f_α^2] = f^2(α)). Therefore the correctness of the algorithm follows; namely, the algorithm outputs all the required coefficients.

By lemma 3.4, we also know that the number of α’s for which E[f_α^2] ≥ θ^2 is bounded by 1/θ^2, for each length of α. Thus, the total number of recursive calls is bounded by O(n/θ^2).
The procedure runs in polynomial time in $n, 1/\theta$ and $\log 1/\delta$. This procedure is based on the ideas of [GL89], although the context is completely different.

Let $f(x) = \sum_{z \in \{0, 1\}^n} \hat{f}(z) \chi_z(x)$. For every $\alpha \in \{0, 1\}^k$, we define the function $f_\alpha : \{0, 1\}^{n-k} \to \mathbb{R}$ as follows:

$$f_\alpha(x) \triangleq \sum_{\beta \in \{0, 1\}^{n-k}} \hat{f}(\alpha \beta) \chi_\beta(x).$$

In other words, the function $f_\alpha(x)$ includes all the coefficients $\hat{f}(z)$ of $f$ such that $z$ starts with $\alpha$ (and all the other coefficients are 0). This immediately gives the key idea for how to find the large coefficients of $f$: find (recursively) the large coefficients of $f_0$ and $f_1$. Note that during the learning process we can only query for the value of the target function $f$ in certain points. Therefore, we first have to show that $f_\alpha(x)$ can be efficiently computed using such queries to $f$. Actually, we need not compute the exact value of $f_\alpha(x)$ but just need to approximate it. The following lemma gives an equivalent formulation of $f_\alpha$, which is computationally much more appealing:

**Lemma 3.2** For any function $f$, any $1 \leq k < n$, any $\alpha \in \{0, 1\}^k$, and any $x \in \{0, 1\}^{n-k}$,

$$f_\alpha(x) = E_y \in \{0, 1\}^k [f(y) \chi_\alpha(y)].$$

This formulation implies that even though we do not know how to compute the value of $f_\alpha(x)$ we can approximate it, by approximating the above expectation.

**Proof:** Let $f(yx) = \sum_z \hat{f}(z) \chi_z(yx)$. Note that if $z = z_1z_2$, where $z_1 \in \{0, 1\}^k$, then $\chi_z(yx) = \chi_{z_1}(yx)\chi_{z_2}(x)$. Therefore,

$$E_y[f(yx)\chi_\alpha(y)] = E_y \left[ \left( \sum_{z_1} \sum_{z_2} \hat{f}(z_1z_2) \chi_{z_1}(y) \chi_{z_2}(x) \right) \chi_\alpha(y) \right]$$

$$= \sum_{z_1} \sum_{z_2} \hat{f}(z_1z_2) \chi_{z_1}(y) E_y[\chi_{z_2}(x) \chi_\alpha(y)],$$

where $y$ and $z_1$ denote strings in $\{0, 1\}^k$ and $z_2$ denotes strings in $\{0, 1\}^{n-k}$. By the orthonormality of the basis, (see Section 2.1) it follows that $E_y[\chi_{z_1}(y) \chi_\alpha(y)]$ (which is the same as $\langle \chi_{z_1}, \chi_\alpha \rangle$) equals 0 if $z_1 \neq \alpha$, and equals 1 if $z_1 = \alpha$. Therefore, only the terms with $z_1 = \alpha$ contributes in the sum. Thus, the last term equals

$$\sum_{z_2 \in \{0, 1\}^{n-k}} \hat{f}(\alpha z_2) \chi_{z_2}(x) = f_\alpha(x).$$

□

Since both $|f(x)| = 1$ and $|\chi_\alpha(y)| = 1$ we derive the following corollary on the value of $f_\alpha(x)$.

**Corollary 3.3** For any boolean function $f$, any $1 \leq k < n$, any $\alpha \in \{0, 1\}^k$, and any $x \in \{0, 1\}^{n-k}$,

$$|f_\alpha(x)| \leq 1.$$
3 Approximation by sparse functions

In this section we show how to find an approximation by a sparse function. The main result in this section is that if \( f \) can be \( \varepsilon \)-approximated by some polynomially-sparse function \( g \) then there is a randomized polynomial time procedure that finds some function \( h \) that \( O(\varepsilon) \)-approximates \( f \). (A function \( g \) is \( t \)-sparse if it has at most \( t \) Fourier coefficients that are not zero.)

The first step is to show that if \( f \) can be approximated by a polynomially sparse function \( g \), it can be approximated by a polynomially sparse function that has only “large” coefficients. We remark that we do not make a “direct” use of \( g \) (e.g., by approximating \( g \) instead of approximating \( f \)) but only use its existence in the analysis.

**Lemma 3.1** If \( f \) can be approximated by a \( t \)-sparse function \( g \) such that \( E[(f - g)^2] \leq \varepsilon \), then there exists a \( t \)-sparse function \( h \) such that \( E[(f - h)^2] \leq \varepsilon + O(\varepsilon^2 / t) \) and all the non-zero coefficients of \( h \) are at least \( \varepsilon / t \).

**Proof:** Let \( g(x) = \sum_{i=1}^{t} \hat{g}(z_i) \chi_{z_i}(x) \). Note that the Fourier coefficients of the function \( f - g \) are exactly \( \hat{f}(z) - \hat{g}(z) \). Therefore, by Parseval’s equality,

\[
E[(f - g)^2] = \sum_z (\hat{f}(z) - \hat{g}(z))^2.
\]

Thus, requiring that \( \hat{g}(z_i) = \hat{f}(z_i) \) can only reduce the expected error squared. Therefore, without loss of generality, the non-zero coefficients of \( g \) are the coefficients of \( f \), i.e. \( g(x) = \sum_{i=1}^{t} \hat{f}(z_i) \chi_{z_i}(x) \). Let \( h \) be the function obtained from \( g \) by replacing the “small” coefficients by 0. Namely,

\[
h(x) = \sum_{\hat{f}(z_i) \geq \varepsilon / t} \hat{f}(z_i) \chi_{z_i}(x).
\]

We now show that \( E[(f - h)^2] \leq \varepsilon + O(\varepsilon^2 / t) \). Consider the expression,

\[
E[(f - h)^2] - E[(f - g)^2].
\]

By the above arguments, this is equal to

\[
\sum_z (\hat{f}(z) - \hat{h}(z))^2 - \sum_z (\hat{f}(z) - \hat{g}(z))^2 = \sum_{\hat{f}(z_i) < \varepsilon / t} (\hat{f}(z_i) - \hat{h}(z_i))^2 = \sum_{\hat{f}(z_i) < \varepsilon / t} \hat{f}^2(z_i) < (\frac{\varepsilon}{t})^2 t = \varepsilon^2 / t.
\]

Since \( E[(f - g)^2] \leq \varepsilon \), the lemma follows. \( \square \)

The above lemma has reduced the problem of approximating \( f \) by a \( t \)-sparse function to the problem of finding all the coefficients of \( f \) that are greater than a threshold of \( \varepsilon / t \). Note that the function \( h \) defined above does not necessarily contain all the coefficients of \( f \) that are greater than \( \varepsilon / t \), but only those that appear also in \( g \). However, adding these coefficients to \( h \) will clearly make \( h \) a better approximation for \( f \). In any case, the number of these coefficients, as follows from Lemma 3.4 below, cannot be too high.

In the remainder of this section we show a randomized polynomial time procedure that given a function \( f \) and a threshold \( \theta \) outputs (with prob. \( 1 - \delta \)) all the coefficients for which \(|\hat{f}(z)| \geq \theta \).
at an inner node \( v \), labeled by \( S_v \), it assigns the node \( v \) the value \( \sum_{i \in S_v} x_i \mod 2 \), which we denote by \( \text{val}(v) \). If \( \text{val}(v) = 1 \) then the computation continues to the right son of \( v \), otherwise it continues to the left son. The computation terminates at a leaf \( u \) and outputs the label of \( u \) (which is also the value of the leaf). The value of the tree on an input is the value of the output of the computation.

Note that if, for example, \( |S_v| = 1 \) then the meaning of the operation is testing the value of a single variable which is the only permitted operation in the traditional decision tree model. If, for example, \( |S_v| = 2 \) then the meaning of the operation is testing whether the two corresponding variables are equal, and if \( |S_v| = n \) then in a single operation we have a test for the parity of all variables. In the traditional decision tree model computing the parity of all the variables requires \( 2^n \) nodes.

### 2.3 Learning Model

The learner in our model uses only membership queries. That is, it can query the unknown function \( f \) on any input \( x \in \{0, 1\}^n \) and receive \( f(x) \). After performing a finite number of membership queries, the learner outputs an hypothesis \( h \). The error of an hypothesis \( h \), with respect to the function \( f \), is defined to be \( \text{error}(f, h) \triangleq \text{Prob}[f(x) \neq h(x)] \), where \( x \) is distributed uniformly over \( \{0, 1\}^n \).

A randomized algorithm \( A \) learns a class of functions \( \mathcal{F} \) if for every \( f \in \mathcal{F} \) and \( \varepsilon, \delta > 0 \) the algorithm outputs an hypothesis \( h = A(f, \varepsilon, \delta) \) such that

\[
\text{Prob}[\text{error}(f, h) \geq \varepsilon] \leq \delta.
\]

The algorithm \( A \) learns in polynomial time if its running time is polynomial in \( n, 1/\varepsilon \), and \( \log 1/\delta \).

We also discuss deterministic learning algorithms. An algorithm \( A \) deterministically learns a class of functions \( \mathcal{F} \) if for every \( f \in \mathcal{F} \) and \( \varepsilon > 0 \) the algorithm outputs an hypothesis \( h = A(f, \varepsilon) \) such that

\[
\text{error}(f, h) \leq \varepsilon.
\]

The algorithm \( A \) learns in deterministic polynomial time if its running time is polynomial in \( n \) and \( 1/\varepsilon \). Note that in a deterministic algorithm we do not have a parameter \( \delta \). That is, the algorithm always succeeds in finding a “good” hypothesis.

A (real) function \( g \) \( \varepsilon \)-approximates \( f \) (in norm \( L_2 \)) if \( E[(f(x) - g(x))^2] \leq \varepsilon \). In the case that \( f \) is a boolean function, we can convert a real prediction function \( g \) to a boolean prediction by predicting the sign of \( g \). In such a case, if \( f(x) \neq \text{sign}(g(x)) \) then \( |f(x) - g(x)| \geq 1 \), which implies

\[
\text{Prob}[f(x) \neq \text{sign}(g(x))] \leq E[(f(x) - g(x))^2] \leq \varepsilon.
\]

Thus, we have

**Claim 2.1** If \( g \) \( \varepsilon \)-approximates a boolean function \( f \), then

\[
\text{Prob}[f(x) \neq \text{sign}(g(x))] \leq \varepsilon.
\]
the cube $Z^n_2$ is a $2^n$-dimensional real vector space with an inner product defined by:

$$<g,f> = \frac{1}{2^n} \sum_{x \in \{0,1\}^n} g(x)f(x) = E[gf].$$

The norm of a function $f$ is defined by $\|f\|_2 = \sqrt{<f,f> = E[f^2]}.$

Define a basis for the linear space of real functions on the cube $Z^n_2$, using the characters of $Z^n_2$ as follows: For each $z \in \{0,1\}^n$, define the basis function $\chi_z$:

$$\chi_z(x_1, \ldots, x_n) = \begin{cases} +1 & \text{if } \sum_i z_ix_i \text{ mod } 2 = 0 \\ -1 & \text{if } \sum_i z_ix_i \text{ mod } 2 = 1 \end{cases}$$

The following properties of these functions can be verified easily:

- For every two vectors $z_1, z_2 \in \{0,1\}^n$: $\chi_{z_1}\chi_{z_2} = \chi_{z_1 \oplus z_2}$, where $\oplus$ denotes bitwise exclusive-or.

- The family of functions $\{\chi_z : z \in \{0,1\}^n\}$ forms an orthonormal basis. That is, (1) any function $f(x)$ on the cube $Z^n_2$ can be uniquely expressed as $\sum_z f(z)\chi_z(x)$, where $f(z)$ are real constants; and (2) if $z_1 \neq z_2$, then $\langle \chi_{z_1}, \chi_{z_2} \rangle = 0$, and for every $z$, $\langle \chi_z, \chi_z \rangle = 1$.

The Fourier transform of $f$ is just the expansion of $f$ as a linear combination of the $\chi_z$'s. Since the $\chi_z$'s are an orthonormal basis, Fourier coefficients are found via

$$\hat{f}(z) = \langle f, \chi_z \rangle = E[f\chi_z].$$

The orthonormality of the basis implies Parseval's identity:

$$E[f^2] = \|f\|_2^2 = \sum_{z \in \{0,1\}^n} \hat{f}^2(z).$$

Note that if for every $x \in Z^n_2$, $|f(x)| \leq 1$, then $\|f\|_2 \leq 1$ and therefore for every $z \in \{0,1\}^n$, $|\hat{f}(z)| \leq 1$. Finally, we define $L_1(f)$ as the $L_1$ norm of the coefficients of $f$, i.e. $L_1(f) = \sum_z |\hat{f}(z)|$.

2.2 Boolean decision trees

In this section we give a precise definition of the decision tree model used in this work. This model is much stronger than the traditional decision tree model.

A boolean decision tree $T$ consists of a labeled binary tree. Each inner node $v$ of the tree is labeled by a set $S_v \subseteq \{1, \ldots, n\}$, and has two outgoing edges. Every leaf of the tree is labeled by either $+1$ or $-1$. (Throughout this paper a function is called boolean if its range is $\{+1, -1\}$.)

Given an input, $x = (x_1, \ldots, x_n)$, the decision tree defines a computation. The computation traverses a path from the root to a leaf and assigns values to the nodes on the path in the following way. The computation starts at the root of the tree $T$. When the computation arrives
In the work of [RB91] the same learning model was considered (i.e., using membership queries and testing the hypothesis with respect to the uniform distribution). They show that any polynomial over $GF(2)$ with polynomial number of terms can be learned in polynomial time in such a model. The class of polynomials with polynomial number of terms (considered in [RB91]) and the class of boolean decision trees with linear operations (considered in our work) are incomparable. On the one hand, the inner-product function has a small polynomial but does not have a small decision tree. On the other hand, consider a boolean decision list with $\log n$ nodes, where each node computes the sum of $\Omega(n)$ variables. Representing such a decision list by a polynomial may require $\Omega(n^{\log n})$ terms.

The power of polynomial size boolean decision trees with linear operations is also incomparable to $AC^0$ circuits (which are the target of the learning algorithm of [LMN89]). Such trees can compute parity, which cannot be approximated by $AC^0$ circuits (see [FSS84, Ajt83, Yao85, Has86]). We show that for boolean decision trees with linear operations the $L_1$-norm is bounded by the number of nodes; therefore, computing a polynomial-size DNF that has an exponential $L_1$-norm would require an exponential number of nodes (see [BS90] for a construction of such a DNF).

The class $\mathcal{F}$ of boolean functions whose $L_1$-norm is polynomially bounded was also studied in [Bru90, BS90, SB91]. They showed that any such function $f$ can be approximated by a sparse polynomial of a certain form. Note, however, that their notion of approximation is different than ours. Another type of approximation for boolean functions was recently suggested in [ABFR91] (and then studied by others). In that work boolean functions are approximated by the sign of a low-degree polynomial over the integers.

1.2 Organization

The rest of this paper is organized as follows. Section 2 has the definitions of Fourier transform, decision trees and the learning model. Section 3 includes the procedure that finds the approximating sparse function. In section 4 we prove the properties of functions with small $L_1$-norm. In section 5 we prove the results about boolean decision trees with linear operations. Finally, in section 6 we discuss some extensions and mention some open problems.

2 Preliminaries

In this section we give the definition of Fourier transform and recall some known properties of it (section 2.1). Then, we formally define the model of decision trees which is used in this work (section 2.2). We end by describing the membership-queries learning model which is used in this work (section 2.3).

2.1 Fourier Transform

Let $f : \{0, 1\}^n \to \mathbb{R}$ be a real function. Denote by $E[f]$ the expected value of $f(x)$ with respect to the uniform distribution on $x$, i.e., $E[f] = \frac{1}{2^n} \sum_{x \in \{0, 1\}^n} f(x)$. The set of all real functions on
• We prove that the $L_1$-norm of the coefficients of a decision tree is bounded by the number of nodes in the tree. Therefore, polynomial size decision trees are in the class $\mathcal{F}$. It follows that every polynomial size decision tree with linear operations can be learned in polynomial time.

Furthermore, for functions in the class $\mathcal{F}$ we show how to derandomize the learning algorithm. The derandomization uses constructions of "small", "almost unbiased" probability spaces, called $\lambda$-bias distributions [NN90, AGHP90]. (For a formal definition of $\lambda$-bias probability distributions see Section 4.1.) Thus, we derive a deterministic polynomial time algorithm for learning decision trees.

Our technique sheds a new light on the possibilities of using $\lambda$-bias distributions for derandomization. We show that the deviation of the expected value of a function $f$ with respect to the uniform distribution and a $\lambda$-bias distribution is bounded by $\lambda \cdot L_1(f)$. One nice example where this bound comes in handy is for showing that the deviation of the AND of a subset of the $n$ variables is bounded by $3\lambda$. (This is since $L_1(\text{AND}) \leq 3$, independent of the subset of variables or its size.)

### 1.1 Relations to Other Works

Our result could be contrasted with the result of [EH89], where an $O(n^{\log m})$ algorithm is given for learning decision trees in the $PAC$ model, where $n$ is the number of variables and $m$ is the number of nodes in the tree. Their algorithm learns traditional boolean decision trees with respect to an arbitrary distribution, and uses only examples drawn from that distribution. Therefore, it learns in a weaker model. On the other hand, it runs in time $O(n^{\log m})$ compared to the polynomial time of our algorithm. Also, our algorithm handles a stronger model of boolean decision trees, which include linear operations, while the algorithm of [EH89] does not seem to extend to such a model. In [Han90] a polynomial-time algorithm was presented for learning $\mu$-decision trees using membership queries and equivalence queries, and in [Han91] a polynomial time algorithm was presented for learning decision trees in which each variable appears at most a constant number of times. (Again, these results do not address linear operations.)

Recently, Bellare [Bel92] was able to extend a few of our results concerning decision trees, and show how to derive an upper bound on the sum of the Fourier coefficients as a function of the predicates in the nodes. He also extends the learning algorithm to the case of product distributions, and show that if the $L_1$-norm of $f$ (with respect to a product distribution $\mu$) is polynomially bounded then it can be learned (with respect to $\mu$) in polynomial time. Unfortunately, this result falls short of showing that decision-trees are learnable with respect to product distributions, since there are functions (e.g., the AND function) that have a small size decision tree but their $L_1$-norm is exponential with respect to some product distributions.

Following our work, it has been shown [Man92] how to learn DNF formulas, with respect to the uniform distribution, in $O(n^{\log \log n})$ time. The main contribution of that work is made by bounding the number of "large" coefficients in the Fourier expansion of such a function by $O(n^{\log \log n})$. Then, the algorithm of this paper is used to recover them.
the class $\mathcal{AC}^0$ (polynomial size constant depth circuits), where the quality of the approximation is judged with respect to the uniform distribution (and $n$ is the number of variables). Their main result is an interesting property of the representation of the Fourier transform of $\mathcal{AC}^0$ circuits. Using this property, they derive the learning algorithm for this class of functions. [FJS91] has extended the result to apply also to mutually independent distributions (i.e., product distributions) with a similar running time (i.e. quasi-polynomial time). In [AM91] polynomial time algorithms are given for learning both decision lists and $\mu$-decision trees (a boolean decision tree in which each variable appears only once) with respect to the uniform distribution. As in [LNN89] these algorithms make use of special properties of the Fourier coefficients and approximate the target function by observing examples drawn according to the uniform distribution. More information about Fourier transform over finite groups is found in [Dia88].

In this work we show another interesting property of the representation that is applied to achieve learnability. The learning model allows membership queries, where the learner can query the (unknown) function on any input. Our main result is a polynomial-time algorithm for learning functions computed by boolean decision trees with linear operations (over $\mathcal{GF}(2)$). In these trees each node computes a summation (modulo 2) of a subset of the $n$ boolean input variables, and branches according to whether the sum is zero or one. Clearly, this is an extension of the traditional boolean decision tree model, since we can still test single variables. On the other hand, we can test in a single operation the parity of all the input variables, compared with a lower bound of $2^n$ nodes in the traditional model (see [BHO90]).

An interesting consequence of our construction is that one can exactly find the Fourier transform representation of boolean decision trees with linear operations in time $\text{poly}(n, 2^d)$, where $d$ is the depth of the tree. This implies that we find a function that is identical to the tree for any boolean input. A corollary of this result is that decision trees with logarithmic depth can be exactly identified in polynomial time. (Note that enumeration, even of constant depth trees, would require exponential time (due to the linear operation); even eliminating the linear operations and constraining each node to contain a single variable, the number of trees of depth $d$ is $\Omega(n^d)$.)

Our main result – the learning algorithm for decision trees – is achieved by combining the following three results:

- **The algorithmic tool** – We present a randomized polynomial time algorithm that performs the following task. The algorithm receives as an input a boolean function $f$ that can be approximated by a polynomially sparse function $g$ (a function with a polynomial number of non-zero Fourier coefficients) such that the expected error square (i.e. $E(f - g)^2$) is bounded by $\varepsilon$. The algorithm finds some polynomially sparse function $h$ that approximates $f$, such that $E(f - h)^2 = O(\varepsilon)$. The algorithm we develop here is based on the ideas of [GL89].

- We consider the class of functions $\mathcal{F} = \{ f : L_1(f) \leq \text{poly}(n) \}$, where $L_1(f)$ is the $L_1$-norm of the coefficients (i.e., the sum of the absolute value of the coefficients). We show that in order to achieve an approximation of a function $f \in \mathcal{F}$ within $\varepsilon$, it is sufficient to consider only coefficients larger than $\frac{\varepsilon}{L_1(f)}$ (there are at most $\frac{L_1(f)}{\varepsilon}^2$ such coefficients). Therefore, every function in the class $\mathcal{F}$ can be approximated by a polynomially sparse function and therefore can be learned in polynomial time by our algorithm.
Learning Decision Trees using the Fourier Spectrum

Eyal Kushilevitz†  Yishay Mansour‡

Abstract

This work gives a polynomial time algorithm for learning decision trees with respect to the uniform distribution. (This algorithm uses membership queries.) The decision tree model that is considered is an extension of the traditional boolean decision tree model that allows linear operations in each node (i.e., summation of a subset of the input variables over \(GF(2)\)).

This paper shows how to learn in polynomial time any function that can be approximated (in norm \(L_2\)) by a polynomially sparse function (i.e., a function with only polynomially many non-zero Fourier coefficients). The authors demonstrate that any function \(f\) whose \(L_1\)-norm (i.e., the sum of absolute value of the Fourier coefficients) is polynomial can be approximated by a polynomially sparse function, and prove that boolean decision trees with linear operations are a subset of this class of functions. Moreover, it is shown that the functions with polynomial \(L_1\)-norm can be learned deterministically.

The algorithm can also exactly identify a decision tree of depth \(d\) in time polynomial in \(2^d\) and \(n\). This result implies that trees of logarithmic depth can be identified in polynomial time.

1 Introduction

In recent years much effort has been devoted to providing a theoretical basis for machine learning. These efforts involved formalization of learning models and algorithms, with a special emphasis on polynomial running time algorithms (see [Val84, Ang87]). This work further extends our understanding of the learning tasks that can be performed in polynomial time.

Recent work by [LMN89] has established the connection between the Fourier spectrum and learnability. They presented a quasi-polynomial-time (i.e. \(O(n^{poly \log n})\)) algorithm for learning

---

† Dept. of Computer Science, Technion, Haifa 32000, Israel. Current address: Aiken Computation Laboratory, Harvard University, Cambridge MA 02138.
‡ Aiken Computation Laboratory, Harvard University, Cambridge MA 02138. Partially supported by ONR N00014-85-K-0445.
Notes for Lecture 27

Learning Bounded-Depth Circuits

In this lecture we discuss a quasi-polynomial time algorithm to learn bounded-depth circuits with queries under the uniform distribution. As in Lecture 15 and 16, we denote bits using the set \{-1,1\} instead of \{0,1\}.

The main result is

**Theorem 1 (Main)** Given oracle access to a function \(f: \{-1,1\}^n \to \{-1,1\}\) computed by a circuit of size \(S\) and depth \(d\), and given parameters \(\epsilon\) and \(\delta\), in time polynomial in \(n^{O((14 \log S/\epsilon)^{d-1})}\) and \(\log 1/\delta\) we can output a circuit that, with probability at least \(1 - \delta\), is \(\epsilon\)-close to \(f\).

As in Lectures 15 and 16, we divide the proof in two parts. We show that every function whose Fourier coefficients satisfy certain properties can be efficiently learned, and then we show that functions computed by bounded depth circuits satisfy these properties.

**Lemma 2** Given oracle access to \(f: \{-1,1\}^n \to \{-1,1\}\) such that

\[
\sum_{S:|S| \leq t} |\hat{f}_S| \leq m
\]

and

\[
\sum_{S:|S| > t} |\hat{f}_S|^2 \leq \alpha
\]

and given parameters \(\epsilon, \delta > 0\), it is possible to construct in time polynomial in \(n, m, 1/\epsilon, \log 1/\delta\), a circuit \(C\) that, with probability at least \(1 - \delta\), is \(\epsilon + \alpha\)-close to \(f\).

**Lemma 3** If \(f: \{-1,1\}^n \to \{-1,1\}\) is computed by a circuit of size \(S\) and depth \(d\), then

\[
\sum_{S:|S| > t} |\hat{f}_S|^2 \leq 2\alpha
\]

for \(t := 28 \cdot (14 \log(S/\alpha))^{d-1}\).

Given the two Lemmas, we just need to observe that \(\sum_{|S| \leq t} |\hat{f}_S|^{n_t} < n^t\) for every boolean function \(f\), and the proof of Theorem 1 follows.
1 More on Learning and Fourier Analysis

In this section we prove Lemma 2. First recall the version of the Goldreich-Levin algorithm we presented in Lecture 15.

Lemma 4 There is a probabilistic algorithm that given oracle access to a function $f : \{-1,1\}^n \to \{-1,1\}$, a threshold parameter $\tau$, a confidence parameter $\delta$ and an accuracy parameter $\gamma$, runs in time polynomial in $n$, $\gamma^{-1}$, $\tau^{-1}$ and $\log \delta^{-1}$ and outputs a list $L$ of $O(\tau^{-2} \log \tau^{-1} \delta^{-1})$ sets, and a value $\hat{f}_S$ for every set $S \in L$, such that, with probability at least $1 - \delta$ the following conditions hold:

- Every set $S$ such that $|\hat{f}_S| \geq \tau$ is in the list;
- For every set $S$ in the list, $|\hat{f}_S - \hat{f}| \leq \gamma$.

We fix $\tau = \epsilon/2m$. If $\ell = O(\tau^{-2} \log \tau^{-1} \delta^{-1})$ is an upper bound to the size of the list returned by the Goldreich-Levin algorithm with threshold $\tau$ and confidence $\delta$, then we fix $\gamma = \sqrt{\epsilon/2\ell}$ and we run the Goldreich-Levin algorithm with threshold $\tau$, confidence $\delta$ and accuracy $\gamma$. We find a list $L$ of sets and values $\hat{f}_S$ for each set in the list such that, with probability $\geq 1 - \delta$ over the internal coin tosses of the algorithm, we have:

- Every set $S$ such that $|\hat{f}_S| \geq \tau$ is in the list;
- For every set $S$ in the list, $|\hat{f}_S - \hat{f}| \leq \gamma$.

Then we define the function $h(x) = \sum_{S \in L, |S| \leq t} \hat{f}_S u_S$. The Fourier coefficients of the difference $d(x) := f(x) - h(x)$ are as follows.

- If $|S| > t$, then $\hat{h}_S = 0$ and so $\hat{d}_S = \hat{f}_S$.
- If $|S| \leq t$ and $S \not\in L$, then $\hat{h}_S = 0$ and so $\hat{d}_S = \hat{f}_S$, and also $|d_S| = |\hat{f}_S| \leq \tau$.
- If $|S| \leq t$ and $S \in L$, then $|\hat{h}_S| \leq \gamma$.

We now want to estimate $E[(f(x) - h(x))^2]$. We have

$$E[(f(x) - h(x))^2] = E[d^2(x)]$$
$$= \sum_S d^2_S$$
$$= \sum_{|S| > t} d^2_S + \sum_{|S| \leq t, S \not\in L} d^2_S + \sum_{|S| \leq t, S \in L} d^2_S$$
$$\leq \sum_{|S| > t} f^2_S + \tau \sum_{|S| \leq t, S \not\in L} |d_S| + \sum_{|S| \leq t, S \in L} \gamma^2$$
$$\leq \alpha + \tau m + |L|\gamma^2$$
$$\leq \alpha + \epsilon$$

Define $g : \{-1,1\}^n \to \{-1,1\}$ such that $g(x) = 1$ if $h(x) \geq 0$ and $g(x) = -1$ if $h(x) < 0$. We see that

$$\Pr[g(x) \neq f(x)] \leq E[(f(x) - h(x))^2] \leq \alpha + \epsilon$$

We output the circuit that computes $g$. 

2
2 The Fourier Spectrum of Functions Computed by Bounded-Depth Circuits

In this section we prove Lemma 3.

We state without proof the following result of Linial, Mansour and Nisan. Denote by $DTD(f)$ the depth of the shallowest decision tree that computes $f$.

Lemma 5 (Linial-Mansour-Nisan) Let $R_p$ be the distribution over random restrictions with parameter $p$, and let $f : \{-1, 1\}^n \to \{-1, 1\}$ be a function.

$$
\sum_{|S|>t} \hat{f}_S^2 \leq 2\Pr_{\rho \sim R_p}[DTD(f_\rho) \geq tp/2]
$$

In other words, if $f$ is a function that is very likely to be computable by a shallow decision tree after a random restriction, then $f$ has low weight on large coefficient.

Lemma 6 If $f$ can be computed by a circuit of size $C$ and depth $d$, then for $p = 1/196 \cdot (\log S/\alpha)^{d-2}$

$$
\Pr_{\rho \sim R_p}[DTD(f_\rho) \geq \log(S/\alpha)] \leq \alpha
$$

Proof: Define $k := (\log S/\alpha)$, and let $s_1, \ldots, s_d$ be the number of gates at depth 1, $\ldots, d$ in $C$. First apply a random restriction with parameter 1/14, and consider the $s_1$ top gates, that we can think of as being 1-DNF (respectively, 1-CNF). Then, after the random restriction, each gate has probability at least $1 - 2^{-k}$ of being expressible as a decision tree of depth $k$, and so as a $k$-CNF (respectively, $k$-DNF). Substitute such an expression in each gate, and obtain a circuit of depth $d+1$ with top gates of fan-in $k$ and two consecutive layers of AND gates (respectively, OR gates). By using associativity, we get a circuit of depth $d$, with top gates of fan-in $k$, and with $s_2, \ldots, s_d$ gates at levels 2, $\ldots, d$ (With probability $\leq s_1/2^k$, this first step fails.) The circuit has $n/14$ inputs.

Then we apply a random restriction with parameter $1/(14k)$, and consider the $s_2$ level-2 gates. Each of them computed a $k$-CNF (respectively, $k$-DNF) before the restriction. After the restriction, each such gate computes a function computable by a depth-$k$ decision tree, except with probability at most $2^{-k}$. By replacing each gate with a $k$-DNF (respectively, $k$-CNF) and collapsing level 2 with level 3, we get a circuit of depth $d-1$, with top gates of fan-in $k$, and with $s_3, \ldots, s_d$ gates at levels 2, $\ldots, d_1$. (With probability $\leq s_2/2^k$, this second step fails.) The circuit has now $n/(14 \cdot (14k))$ inputs.

After $d-3$ steps like the one above, we get a circuit of depth 2, with top gates having fan-in $\leq k$. The circuits has $n/(14 \cdot (14k)^{d-2})$ inputs. There is a probability at most $((s_1 + \cdots + s_{d-1})/2^k)$ that the construction failed at any point. We apply one more random restriction with $p = 1/14$, and the circuit finally becomes a depth-$k$ decision tree, except with probability at most $2^{-k}$. Putting every together, and recalling $S = s_1 + \cdots + s_{d-1} + 1$ and $2^k = S/\alpha$, we have that for $p = 1/196 \cdot (14k)^{d-2}$

$$
\Pr_{\rho \sim R_p}[DTD(f_\rho) \geq k] \leq \alpha
$$

as desired. □
In order to prove Lemma 3, it now remains to fix \( t = 28 \cdot (14 \log(S/\alpha)^{d-1}) \) and \( p = 1/(196(14 \log(S/\alpha))^{d-2}) \), and deduce

\[
\sum_{|S| > t} \hat{f}_S^2 \leq 2 \Pr_{\rho \sim R_p}[DTD(f_\rho) \geq tp/2] = 2 \Pr_{\rho \sim R_p}[DTD(f_\rho) \geq \log S/\alpha] \leq 2\alpha
\]

3 References

These results are due to Linial, Mansour and Nisan [LMN93].

References

Notes for Lecture 25

Circuit Lower Bounds for Parity Using Polynomials

In this lecture we prove a lower bound on the size of a constant depth circuit which computes the XOR of \( n \) bits.

Before we talk about bounds on the size of a circuit, let us first clarify what we mean by circuit depth and circuit size. The depth of a circuit is defined as the length of the longest path from the input to output. The size of a circuit is the number of AND and OR gates in the circuit. Note that, for our purpose, we assume all the gates have unlimited fan-in and fan-out. We define \( \text{AC}^0 \) to be the class of decision problems solvable by circuits of polynomial size and constant depth. We want to prove the result that PARITY is not in \( \text{AC}^0 \).

There are two known techniques to prove this result. In this class, we will talk about a proof which uses polynomials; in the next class we will look at a different proof which uses random restrictions.

1 Circuit Upper Bounds for PARITY

Before we go into our proof, let us first look at a circuit of constant depth \( d \) that computes PARITY.

**Theorem 1** For every constant \( d \geq 2 \), there are circuits of size \( 2^{O(n^{1\over d-1})} \) that compute parity.

In the next lecture, we will prove a \( 2^{n^{1\over 2d-1}} \) size lower bound, establishing the tightness of Theorem 1. Today we will prove a weaker \( 2^{n^{1\over 4d}} \) lower bound.

**Proof:** [Of Theorem 1] Consider the circuit \( C \) shown in Figure 1, which computes the PARITY of \( n \) variables. The circuit \( C \) is a tree of XOR gates, each of which has fan-in \( n^{1\over d-1} \); the tree has depth \( d - 1 \).

Now, since each XOR gate is a function of \( n^{1\over d-1} \) variables, it can be implemented by a CNF or a DNF of size \( 2^{n^{1\over d-1}} \). Let us replace alternating layers of XOR gates in the tree by CNF’s and DNF’s - for example we replace gates in the first layer by their CNF implementation, gates in the second layer by their DNF implementation, and so on. This gives us a circuit of depth \( 2(d - 1) \). Now we can use the associativity of OR to collapse consecutive layers of OR gates into a single layer. The same thing can be done for AND to get a circuit of depth \( d \).

This gives us a circuit of depth \( d \) and size \( O(2^{n^{1\over d-1}}) \) which computes PARITY. \( \square \)
Figure 1: Circuit for Computing XOR of n variables; each small circuit in the tree computes the XOR of $k = n^\frac{1}{d-1}$ variables

## 2 Overview of the Lower Bound Proof

For our proof, we will utilise a property which is common to all circuits of small size and constant depth, which PARITY does not have.\(^1\) The property is that circuits of small size and constant depth can be represented by low degree polynomials, with high probability. More formally, we show that if a function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ is computed by a circuit of size $s$ and depth $d$, then there exists a function $g : \{0, 1\}^n \rightarrow \mathbb{R}$ such that $\Pr_x[f(x) = g(x)] \geq \frac{3}{4}$ and $\hat{g}_\alpha \neq 0$ only for $|\alpha| \leq O((\log S)^{2d})$, where $\hat{g}$ is the Fourier transform of $g$.

Then we will show that if a function $g : \{0, 1\}^n \rightarrow \mathbb{R}$ agrees with PARITY on more than a fraction $\frac{3}{4}$ of its inputs, then there is a coefficient $\alpha$ such that $\hat{g}_\alpha \neq 0$ and $|\alpha| = \Omega(\sqrt{n})$. That is, a function which agrees with PARITY on a large fraction of its inputs, has to have high degree. From these two results, it is easy to see that PARITY cannot be computed by circuits of constant depth and small size.

We give a formal definition of *degree* of a function and then formally state the two results that give our lower bound.

**Definition 1** We say that a function $g : \{0, 1\}^n \rightarrow \mathbb{R}$ has degree at most $d$ if there is a polynomial over the reals of degree at most $d$ such that $g$ and the polynomial agree on $\{0, 1\}^n$.

\(^1\)Incidentally, the property is false with high probability for random functions and it is computable in time $2^{O(n)}$ given the truth-table of a function. You may remember that this implies that our lower bound will be a *natural proof*. 

2
An equivalent way of looking at the definition of degree is to consider the size of the largest non-zero coefficient of the Fourier transform of the function.

**Fact 2** A function \( g : \{0,1\}^n \to \mathbb{R} \) has degree at most \( d \) if and only if \( \hat{g}_\alpha = 0 \) for all \( \alpha \) such that \( |\alpha| > d \).

The following two lemmas are the main results of this lecture.

**Lemma 3** For every circuit \( C \) of size \( S \) and depth \( d \), there is a function \( g : \{0,1\}^n \to \mathbb{R} \) of degree \( O((\log S)^{2d}) \) such that \( g \) and \( C \) agree on at least a \( 3/4 \) fraction of \( \{0,1\}^n \).

**Lemma 4** Let \( g : \{0,1\}^n \to \mathbb{R} \) be a function that agrees with \( \text{PARITY} \) on at least a \( 3/4 \) fraction of \( \{0,1\}^n \). Then the degree of \( g \) is \( \Omega(\sqrt{n}) \).

From Lemma 3 and Lemma 4 it is immediate to derive the following lower bound.

**Theorem 5** For every constant \( d \geq 2 \), if \( C \) is a circuit of depth \( d \) and size \( S \) that computes \( \text{parity} \), then \( S \geq 2^{\Omega(n^{1/4d})} \).

**Proof:** From Lemma 3 we have that there is a function \( g : \{0,1\}^n \to \mathbb{R} \) that agrees with \( \text{PARITY} \) on a \( 3/4 \) fraction of \( \{0,1\}^n \), and whose degree is at most \( O((\log S)^{2d}) \). From Lemma 4 we deduce that the degree of \( g \) must be at least \( \Omega(\sqrt{n}) \), so that

\[
(\log S)^{2d} = \Omega(\sqrt{n})
\]

which is equivalent to

\[
S = 2^{\Omega(n^{1/4d})}
\]

\( \square \)

## 3 Proof of Lemma 3

Most of the work in the proof of Lemma 3 will be in showing how to give a “probabilistic approximation” of a single gate using low-degree functions.

### 3.1 Approximating OR

The following lemma says that we can approximately represent OR with a polynomial of degree exponentially small in the the fan-in of the OR gate. We’ll use the notation that \( x \) is a vector of \( k \) bits, \( x_i \) is the \( i \)th bit of \( x \), and \( 0 \) is the vector of zeros (of the appropriate size based on context).

**Lemma 6** For all \( k \) and \( \epsilon \), there exists a distribution \( G \) over functions \( g : \{0,1\}^k \to \mathbb{R} \) such that

1. \( g \) is of degree \( O((\log \frac{1}{\epsilon})(\log k)) \), and
2. for all \( x \in \{0,1\}^k \),
\[
\Pr_{g \sim G} [g(x) = x_1 \lor \ldots \lor x_k] \geq 1 - \epsilon. \tag{1}
\]

**Proof Idea:** We want a random polynomial \( p : \{0,1\}^k \to \mathbb{R} \) that computes OR. An obvious choice is
\[
p_{\text{bad}}(x_1, \ldots, x_k) = 1 - \prod_{i \in \{1, \ldots, k\}} (1 - x_i), \tag{2}
\]
which computes OR with no error. But it has degree \( k \), whereas we’d like it to have logarithmic degree. To accomplish this amazing feat, we’ll replace the tests of all \( k \) variables with just a few tests of random batches of variables. This gives us a random polynomial which computes OR with one-sided error: when \( x = \mathbf{0} \), we’ll have \( p(x) = 0 \); and when some \( x_i = 1 \), we’ll almost always (over our choice of \( p \)) have \( p(x) = 1 \).

**Proof:** We pick a random collection \( S \) of subsets of the bits of \( x \). (That is, for each \( S \in S \) we have \( S \subseteq \{1, \ldots, k\} \)). We’ll soon see how to pick \( S \), but once the choice has been made, we define our polynomial as
\[
p(x_1, \ldots, x_k) = 1 - \prod_{S \in S} \left( 1 - \sum_{i \in S} x_i \right). \tag{3}
\]

Why does \( p \) successfully approximate OR? First, suppose \( x_1 \lor \ldots \lor x_k = 0 \). Then we have \( x = \mathbf{0} \), and:
\[
p(0, \ldots, 0) = 1 - \prod_{S \in S} \left( 1 - \sum_{i \in S} 0 \right) = 0. \tag{4}
\]
So, regardless of the distribution from which we pick \( S \), we have
\[
\Pr_S [p(\mathbf{0}) = 0] = 1. \tag{5}
\]

Next, suppose \( x_1 \lor \ldots \lor x_k = 1 \). We have \( p(x) = 1 \) if and only if the product term is zero. The product term is zero if and only if the sum in some factor is 1. And that, in turn, happens if and only if there is some \( S \in S \) which includes exactly one \( x_i \) which is 1. Formally, for any \( x \in \{0,1\}^k \), \( x \neq \mathbf{0} \), we want the following to be true with high probability.
\[
\exists S \in S. (|\{i \in S : x_i = 1\}| = 1) \tag{6}
\]
Given that we do not want \( S \) to be very large (so that the degree of the polynomial is small), we’ll have to pick \( S \) very carefully. In order to accomplish this, we turn to the Valiant-Vazirani reduction, which you may recall from an earlier class.

**Lemma 7 (Valiant-Vazirani)** Let \( A \subseteq \{1, \ldots, k\} \), let \( a \) be such that \( 2^a \leq |A| \leq 2^{a+1} \), and let \( H \) be a family of pairwise independent hash functions of the form \( h : \{1, \ldots, k\} \to \{0,1\}^{a+2} \). Then if we pick \( h \) at random from \( H \), there is probability at least \( 1/8 \) that there is a unique element \( i \in A \) such that \( h(i) = \mathbf{0} \). Precisely,
\[
\Pr_{h \sim H} [|\{i \in A : h(i) = \mathbf{0}\}| = 1] \geq \frac{1}{8} \tag{7}
\]
With this as a guide, we will define our collection $S$ in terms of pairwise independent hash functions. Let $t > 0$ be a value that we will set later in terms of the approximation parameter $\epsilon$. Then we let $S = \{S_{a,j}\}_{a \in \{0, \ldots, \log k\}, j \in \{1, \ldots, t\}}$ where the sets $S_{a,j}$ are defined as follows.

- For $a \in \{0, \ldots, \log k\}$:
  - For $j \in \{1, \ldots, t\}$:
    * Pick random pairwise independent hash function $h_{a,j} : \{1, \ldots, k\} \rightarrow \{0, 1\}^{a+2}$
    * Define $S_{a,j} = h^{-1}(0)$. That is, $S_{a,j} = \{i : h(i) = 0\}$.

Now consider any $x \neq 0$ which we are feeding to our OR-polynomial $p$. Let $A$ be the set of bits of $x$ which are $1$, i.e., $A = \{i : x_i = 1\}$, and let $a$ be such that $2^a \leq |A| \leq 2^{a+1}$. Then we have $a \in \{0, \ldots, \log k\}$, so $S$ includes $t$ sets $S_{a,1}, \ldots, S_{a,t}$. Consider any one such $S_{a,j}$.

By Valiant-Vazirani, we have

$$\Pr_{h_{a,j} \sim H}[|\{i \in A : h_{a,j}(i) = 0\}| = 1] \geq \frac{1}{8} \quad (8)$$

which implies that

$$\Pr_{h_{a,j} \sim H}[|\{i \in A : i \in S_{a,j}\}| = 1] \geq \frac{1}{8} \quad (9)$$

so the probability that there is some $j$ for which $|S_{a,j} \cap A| = 1$ is at least $1 - \left(\frac{7}{8}\right)^t$, which by the reasoning above tells us that

$$\Pr_p[p(x) = x_1 \lor \ldots \lor x_k] \geq 1 - \left(\frac{7}{8}\right)^t. \quad (10)$$

Now, to get a success probability of $1 - \epsilon$ as required by the lemma, we just pick $t = O(\log \frac{1}{\epsilon})$. The degree of $p$ will then be $|S| = t(\log k) = O((\log \frac{1}{\epsilon})(\log k))$, which satisfies the degree requirement of the lemma. □

Note that given this lemma, we can also approximate AND with an exponentially low degree polynomial. Suppose we have some $G$ which approximates OR within $\epsilon$ as above. Then we can construct $G'$ which approximates AND by drawing $g$ from $G$ and returning $g'$ such that

$$g'(x_1, \ldots, x_k) = 1 - g(1 - x_1, \ldots, 1 - x_k). \quad (11)$$

Any such $g'$ has the same degree as $g$. Also, for a particular $x \in \{0, 1\}^k$, $g'$ clearly computes AND if $g$ computes OR, which happens with probability at least $1 - \epsilon$ over our choice of $g$.

### 3.2 Proof of Lemma 3

Given a circuit $C$ of size $S$ and depth $d$, for every gate we pick independently an approximating function $g_i$ with parameter $\epsilon = \frac{1}{8S}$, and replace the gate by $g_i$. Then, for a given input, the probability that the new function so defined computes $C(x)$ correctly is at least the probability that the results of all the gates are correctly computed, which is at least $\frac{3}{4}$. In particular, there is a function among those generated this way that agrees with $C()$ on at least a $3/4$ fraction of inputs. Each $g_i$ has degree at most $O((\log S)^2)$, because the fan-in of each gate is at most $S$, and the degree of the function defined in the construction is at most $O((\log S)^{2d})$.  

5
4 Proof of Lemma 4

Let $g : \{0, 1\}^n \to \mathbb{R}$ be a function of degree at most $t$ that agrees with PARITY on at least a $3/4$ fraction of inputs. Let $G : \{-1, 1\}^n \to \mathbb{R}$ be defined as

$$G(x) := 1 - 2g\left(\frac{1}{2}, \frac{1}{2}x_1, \ldots, \frac{1}{2}x_n\right)$$

(12)

Note that:

- $G$ is still of degree at most $t$,
- $G$ agrees with the function $\Pi(x_1, \ldots, x_n) = x_1 \cdot x_2 \cdots x_n$ on at least a $3/4$ fraction of $\{-1, 1\}^n$.

Define $A$ to be the set of $x \in \{-1, 1\}^n$ such that $G(x) = \Pi(x)$.

$$A := \left\{ x : G(x) = \prod_{i=1}^{n} x_i \right\}.$$  

(13)

Then $|A| \geq \frac{3}{4} 2^n$, by our initial assumption. Now consider the set $F$ of all functions $f : A \to \mathbb{R}$. These form a vector space of dimension $|A|$ over the reals. We know that any function $f$ in this set can be written as

$$f(x) = \sum_{\alpha} \hat{f}_\alpha \prod_{i \in \alpha} x_i$$

(14)

Over $A$, $G(x) = \prod_{i=1}^{n} x_i$, and so for $x \in A$,

$$\prod_{i \in \alpha} x_i = G(x) \prod_{i \notin \alpha} x_i$$

(15)

By our initial assumption, $G(x)$ is a polynomial of degree at most $t$. Therefore, for every $\alpha$, such that $|\alpha| \geq t$, we can replace $\prod_{i \in \alpha} x_i$ by a polynomial of degree less than or equal to $t + \frac{n}{2}$. Every such function $f$ which belong to $F$ can be written as a polynomial of degree at most $t + \frac{n}{2}$. Hence the set $\{\prod_{i \in \alpha} x_i : |\alpha| \leq t + \frac{n}{2}\}$ forms a basis for the set $S$. As there must be at least $|A|$ such monomials, this means that

$$\sum_{k=0}^{t + \frac{n}{2}} \binom{n}{k} \geq \frac{3}{4} \cdot 2^n$$

(16)

and, in particular,

$$\sum_{k=\frac{n}{2}}^{t + \frac{n}{2}} \binom{n}{k} \geq \frac{1}{4} \cdot 2^n$$

(17)

We know from Stirling’s approximation that every binomial coefficient $\binom{n}{k}$ is at most $O(2^n/\sqrt{n})$, so we get

$$O\left(\frac{t}{\sqrt{n}} \cdot 2^n\right) \geq \frac{1}{4} \cdot 2^n$$

(18)

And so $t = \Omega(\sqrt{n})$. 

6
5 References

The proof of the PARITY lower bound using polynomials is due to Razborov [Raz87] and Smolensky [Smo87].

The first proof that PARITY is not in $\text{AC}^0$ used a different argument, and was due to Furst, Saxe and Sipser [FSS84]. The lower bound was improved to exponential by Yao [Yao85], and the optimal lower bound is due to Håstad [Hås86].

References


Notes for Lecture 23

Preliminary version

Notes on Reingold’s Theorem, Part II

Today we continue the proof that the undirected \((s,t)\)-connectivity problem can be solved in deterministic logarithmic space [Rei04]. We introduce the zig-zag graph product, an operation that can be used to reduce the degree of a graph without affecting much the eigenvalue gap. A sequence of zig-zag and powering operations, starting from an arbitrary connected graph, leads to a graph with a constant eigenvalue gap. The connectivity algorithm relies on such a construction.

1 The Zig-Zag Graph Product

Recall that if \(G\) is an undirected graph and \(A\) is its normalized adjacency matrix (its “random walk matrix”), and if we denote by \(\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n\) the eigenvalues of \(A\), in sorted order, then \(\lambda_1 = 1\) and we use the notation

\[
\lambda(G) := \max_{i=2,\ldots,n} |\lambda_i|
\]

Furthermore, if \(G\) is \(d\)-regular, connected, and not bipartite, then \(\lambda(G) \leq 1 - 1/dn^2\). Finally, we use the notation

\[
\gamma(G) := 1 - \lambda(G)
\]

and we call \(\gamma(G)\) the eigenvalue gap of \(G\). Recall that we proved that the diameter of \(G\) is at most \(\log_{1/\lambda_1} n\), which is \(O(\frac{1}{\gamma} \log n)\), where \(n\) is the number of vertices. In particular, if \(G\) is a graph with eigenvalue gap at least \(1/2\), then the diameter is at most \(\log_2 n\).

The problem of constructing graphs with a large eigenvalue gap is very difficult and it has been the subject of a lot of work. A recent construction of such graphs [RVW02] introduces a graph product called the zig-zag.

If \(G\) and \(H\) are graphs with large eigenvalue gap, the zig-zag product define a new graph \(G \square H\) that has still a large eigenvalue gap and is bigger than \(G\) and \(H\). The point is that one can use the product to start from very small graphs with large gap, that are easier to construct, and then “bootstrap” them to get bigger and bigger ones.

For the product to work, \(G\) must be a regular graph, say, with \(N\) vertices and degree \(D\), and \(H\) must have \(D\) vertices, as many as the degree of \(G\). If \(H\) is a \(d\)-regular graph, then \(G \square H\) is a \(d^2\)-regular graph with \(ND\) vertices. (It may help to think of \(N = 100,000\), \(D = 1,000\) and \(d = 10\) to get a sense of the parameters.)

In \(G \square H\) there is a vertex \([v,a]\) for every vertex \(v\) of \(G\) and vertex \(a\) of \(H\). There is an edge between \([v,a]\) and \([w,b]\) in \(G \square H\) provided that there are vertices \(a', b'\) in \(H\) such that
the edges \((a, a')\) and \((b, b')\) are in \(H\), \(w\) is the \(a'-\text{th}\) neighbor of \(v\) in \(G\), and \(v\) is the \(b'-\text{th}\) neighbor of \(w\) in \(G\). (This makes more sense given the picture I drew in class.)

In [RVW02] there are various formulas relating \(\gamma(G \odot H)\) to \(\gamma(G)\) and \(\lambda(H)\). If we restrict ourselves to special case \(\gamma(H) \geq 1/2\), then we have the following theorem:

If \(\gamma(H) \geq 1/2\), then \(\gamma(G \odot H) \geq 3/8 \gamma(G)\) \hspace{1cm} (1)

2 Combining Zig-Zag and Powering

Suppose now that \(H\) is a \(d\)-regular graph with \(d^{16}\) nodes (this choice will be clear later) and with \(\gamma(H) \geq 1/2\). Let \(G\) be a \(d^{16}\)-regular graph. Consider the graph \(G' := (G \odot H)^8\). (By this notation, we mean the graph whose adjacency matrix is the 8-th power of the adjacency matrix of \((G \odot H)\).) Then, we claim that

\[
\gamma(G') \geq \min \left\{ \frac{1}{2}, 2 \gamma(G) \right\}
\]

(2)

and, furthermore, \(G'\) is a graph with \(Nd^{16}\) vertices and degree \(d^{16}\). In other words, the new graph \(G'\) has a constant factor more vertices than \(G\), the same degree as \(G\), and an eigenvalue gap than is either twice that of \(G\) or at least 1/2.

To prove that Equation 2 holds, first remember that if \(G\) is a graph, then \(\lambda(G^k) = (\lambda(G))^k\) and so \(\gamma(G^k) = 1 - (1 - \gamma(G))^k\). We prove Equation 2 by considering two cases. If \(\gamma(G) \geq 1/4\), then \(\gamma(G \odot H) \geq 3/32\) by Equation 1 and

\[
\gamma((G \odot H)^8) = 1 - (1 - \gamma(G \odot H))^8 \geq 1 - \left( \frac{29}{32} \right)^8 = .545\ldots > \frac{1}{2}
\]

If \(\gamma(G) \leq 1/4\), then

\[
\gamma((G \odot H)^8) = 1 - (1 - \gamma(G \odot H))^8 \geq 8 \cdot \frac{3}{8} \gamma(G) - 28 \left( \frac{3}{8} \gamma(G) \right)^2 \geq 2 \gamma(G)
\]

where we used the fact that \((1 - \epsilon)^8 > 8 \epsilon - \binom{8}{2} \epsilon^2\) for \(0 < \epsilon < 1\). This completes the proof of Equation 2.

3 Reingold’s Algorithm

As before, let \(H\) be a \(d\)-regular graph with \(d^{16}\) nodes and with \(\gamma(H) \geq 1/2\). Such a graph exists for small values of \(d\). In particular, \(d = 8\) can be proved to be enough using the probabilistic method. An explicit construction is known for \(d = 14\).

Let \(G\) be a \(d^{16}\)-regular graph \(G\) with \(n\) vertices that is connected and not bipartite. Consider the following recursively defined family of graphs

\[
G_0 := G
\]

\[
G_i := (G_{i-1} \odot H)^8.
\]
Then $\gamma(G_t) \geq \min\{1/2, \gamma(G) \cdot 2^i\}$. Since $\gamma(G) \geq 1/d_{16}^2 n^2$, we have that for $t = \log_2 d_{16}^2 n^2 = O(\log n)$ we have $\lambda(G_t) \geq 1/2$.

The number of vertices of $G_t$ is $n \cdot (d_{16}^2)^t$, which is polynomial in $n$.

Finally, $G_t$ can be constructed in log-space given $G$ and $t$. More specifically, there is an algorithm that, given a graph $G$ with $n$ vertices and degree $d_{16}$, a graph $H$ with $d_{16}$ vertices and degree $d$, an integer $t$, and the names of two vertices $v$ and $w$ of $G_t$, uses $O(\log n + t)$ space and decides whether $v$ and $w$ are adjacent in $G_t$.

Suppose now that $G$ is $d_{16}$-regular graph that is not necessarily connected, but such that each connected component is not bipartite. Then each connected component of $G$ is a connected, non-bipartite, regular graph. If we apply the above construction to a connected component of $G$ we obtain a graph with eigenvalue gap at least 1/2. Indeed, if we apply the above construction to all of $G$, we can see that the result is the same as if we had applied the construction to each connected component of $G$ separately.

In conclusion, if $G$ is a $d_{16}$-regular graph with $n$ vertices that has $k$ connected components, each non-bipartite, then $G_t$, where $t = \log_2 d_{16}^2 n^2$, is a $d_{16}$-regular graph with $k$ connected components, and each connected component of $G_t$ is a connected graph with eigenvalue gap at least 1/2.

If $s$ and $v$ are vertices in $G$, then let $s_i$ and $v_i$ be vertices of $G_t$ that “correspond” to $s$ and $v$. Such vertices can be defined recursively, by letting $s_0 := s$, $v_0 = v$, and then letting $s_i$ be one of the vertices of $G_i$ in the block of vertices that replace $s_{i-1}$ (similarly for $v_i$). By induction, one can see that $s$ and $v$ are connected in $G$ if and only if $s_t$ and $v_t$ are connected in $G_t$. Finally, deciding whether $s_t$ and $v_t$ are connected in $G_t$ is easy because we know that the diameter of each connected component of $G_t$ is at most $\log_2 n \cdot d_{16}^t = O(\log n)$, where $n \cdot d_{16}^t$ is the number of vertices of $G_t$. Since each connected component of $G_t$ has degree $d_{16} = O(1)$ and diameter $\log_2 n \cdot d_{16}^t = O(\log n)$, it follows that one can explore the entire connected component of $s_t$, and verify whether $v_t$ is an element of it, using $O(\log n)$ space.

What we have described so far gives an $O(\log n)$ space algorithm to decide $(s, t)$-connectivity in undirected graphs that are: (i) regular of degree $d_{16}$ and (ii) such that each connected component is not bipartite.

It is easy to reduce the general case to this special case. First, one can reduce $(s, t)$-connectivity in general graph to $(s, t)$-connectivity in 3-regular graphs. This can be done by replacing each vertex $v$ of $G$ of degree $d_v$ by a cycle of length $d_v$, and then placing the edges of $G$ as a matching in the new graph. That is, if, in $G$, $w$ is the $i$-th neighbor of $v$ and $v$ is the $j$-th neighbor of $w$, then we put an edge between the $i$-th vertex in the cycle of $v$ and the $j$-th vertex in the cycle of $w$. Then we can add $d_{16} - 3$ self-loops to each vertex. This way the graph cannot be bipartite, and it has become regular of degree $d_{16}$.

References


Notes for Lecture 22

Notes on Reingold’s Theorem, Part I

Today we begin a proof that the undirected \((s,t)\)-connectivity problem can be solved in deterministic logarithmic space. We give a number of preliminary definitions and results. We will get to the actual algorithm next time.

1 Review of Linear Algebra

We think of vectors as row vectors, or \(1 \times n\) matrices. If \(v = (v_1, \ldots, v_n) \in \mathbb{R}^n\) is a real vector, then its length is defined as \(||v|| = \sqrt{v_1^2 + \cdots + v_n^2}\). If \(v_1, \ldots, v_n \in \mathbb{R}^n\) are mutually orthogonal vectors of length one, then they are called an orthonormal basis of \(\mathbb{R}^n\). Every vector \(v\) can then be written in a unique way as \(v = \alpha_1 v_1 + \cdots + \alpha_n v_n\), and we have \(||v|| = \sqrt{\alpha_1^2 + \cdots + \alpha_n^2}\).

Let \(A \in \mathbb{R}^{n \times n}\) be a matrix. A (left) eigenvalue of \(A\) is a value \(\lambda \in \mathbb{C}\) such that, for some non-zero vector \(v \in \mathbb{C}^n\),

\[ v \cdot A = \lambda v \]

and, if so, the vector \(v\) is called an eigenvector. If \(\lambda\) is an eigenvalue of \(A\), then we have

\[ v \cdot (A - \lambda I) = 0 \]

where \(I\) is the identity matrix and \(\mathbf{0}\) is the all-zero vector. This implies that the columns of the matrix \((A - \lambda I)\) are not linearly independent, and so

\[ \det(A - \lambda I) = 0 \]

where \(\det\) denotes the determinant. The value \(\det(A - \lambda I)\), as a function of \(\lambda\), is a polynomial of degree \(n\), and so \(A\) can have at most \(n\) eigenvalues. Note also that if \(\det(A - \lambda I) = 0\), then the columns of \(A\) cannot all be linearly independent, and so there must be a nonzero vector \(v\) such that \(v \cdot (A - \lambda I) = 0\). It follows that \(\lambda\) is an eigenvalue of \(A\) if and only if it is a root of the polynomial \(\det(A - \lambda I)\). If we count multiplicities, then there are precisely \(n\) eigenvalues for \(A\).

If \(A \in \mathbb{R}^{n \times n}\) is a symmetric matrix, that is, a matrix such that \(A_{ij} = A_{ji}\) for every \(i, j\), then there are precisely \(n\) real eigenvalues of \(A\), counting multiplicities. Furthermore, several additional results hold (all the results from now on assume that \(A\) is symmetric).

Let \(\lambda\) be an eigenvalue of \(A\), and let \(v\) and \(w\) be eigenvectors of \(\lambda\). Then if \(a, b \in \mathbb{R}\) are reals, we have that

\[ (av + bw) \cdot A = avA + bwA = a\lambda v + b\lambda w = \lambda (av + bw) \]

and so \((av + bw)\) is an eigenvector. This means that set of eigenvectors of \(\lambda\) (plus the all-zero vector) form a linear subspace of \(\mathbb{R}^n\). We have the following theorems:
1. The dimension of the space of eigenvectors of $\lambda$ is equal to the multiplicity of $\lambda$ as a root of $\det(A - \lambda I)$;

2. If $\lambda \neq \lambda'$ are two eigenvalues, then the space of eigenvectors of $\lambda$ is orthogonal to the space of eigenvectors of $\lambda'$.

This means that if $\lambda_1, \ldots, \lambda_n$ are the eigenvalues of $A$, with multiplicities, then we can find mutually orthogonal unit vectors $v_1, \ldots, v_n$ such that each $v_i$ is an eigenvector of $\lambda_i$. Such eigenvectors form a basis for $\mathbb{R}^n$.

We finally come to an observation that will be important later. Consider the matrix $A^2 = A \cdot A$. If $\lambda$ is an eigenvalue of $A$ with eigenvector $v$, then

$$v \cdot A^2 = (v \cdot A) \cdot A = \lambda v \cdot A = \lambda^2 v$$

so that $\lambda^2$ is an eigenvalue of $A^2$ with eigenvector $v$.

If $\lambda_1, \ldots, \lambda_n$ are eigenvalues of $A$ and $v_1, \ldots, v_n$ are the respective eigenvectors, then $\lambda_1', \ldots, \lambda_n'$ are eigenvalues of $A'$ and $v_1', \ldots, v_n'$ are the respective eigenvectors.

If $A$ is a symmetric matrix, $\lambda_1, \ldots, \lambda_n$ are its eigenvalues, and $v_1, \ldots, v_n$ are their respective eigenvectors, then for every vector $v$ we can write in a unique way $v = \alpha_1 v_1 + \ldots + \alpha_n v_n$. Now, note that

$$v \cdot A^t = (\alpha_1 v_1 + \ldots + \alpha_n v_n) = \alpha_1 \lambda_1' (v_1 + \ldots + \alpha_n \lambda_n' v_n)$$

that is, we can compute $v \cdot A^t$ very easily without having to actually compute the matrix $A^t$.

## 2 Graphs and Eigenvalues

Let $G = (V, E)$ be an undirected regular graph of degree $d$. We allow $G$ to have multiple edges and self-loops. We identify $V$ with $\{1, \ldots, n\}$.

We define the “random walk” matrix (normally it is called the normalized adjacency matrix) of $G$ to be the matrix $A \in \mathbb{R}^V \times V$ obtained by dividing each entry of the adjacency matrix of $G$ by $d$. Another way to look at it is as follows: let $i$ and $j$ be vertices of $G$, and consider the probabilistic process of picking at random one of the edges that are incident on $i$; then $A_{i,j}$ is the probability that $j$ is the other endpoint of the selected edge.

Consider now the matrix $A^2$. It should be clear that $A^2_{i,j}$ is the probability that, starting from $i$, a two-step random walk will land on $j$. In general, $A^t_{i,j}$ is the probability of going from $i$ to $j$ in a $t$-step random walk.

Let $p = (p_1, \ldots, p_n) \in \mathbb{R}^V$ be a probability distribution, that is $p_i \geq 0$ for all $i$, and $\sum_i p_i = 1$. Then the vector $p \cdot A$ is also a probability distribution, and it is the distribution that we get in the following way: pick a vertex $i$ at random according to probability $p$, then move to a random neighbor. Similarly, $p \cdot A^t$ is the probability distribution that we get by picking a start vertex according to distribution $p$ and then performing $t$ steps of a random walk.

Since, as we saw above, it is easy to compute a product of the form $p \cdot A^t$ given the eigenvalues and eigenvectors of $A$, it is no wonder that eigenvalues and eigenvectors play an important role in the study of random walks.
First, note that $A$ is a symmetric matrix (because we assumed that $G$ was an undirected graph), and so all its eigenvalues are real numbers.

It is easy to see that 1 is an eigenvalue of $A$, with eigenvector the uniform distribution $u := (1/n, \ldots, 1/n)$. (Recall that we assumed that $G$ was a regular graph, otherwise this would not be true.) It is also a theorem that for every other eigenvalue $\lambda$ we have $|\lambda| \leq 1$.

If $G$ is not connected, then the multiplicity of 1 as an eigenvalue is more than 1, that is, there are at least two linearly independent eigenvectors of 1. (For example, think of the distribution that is uniform in one connected component, and zero elsewhere.) If $G$ is connected, then the eigenvalue 1 has multiplicity 1. It is however possible that $-1$ could be an eigenvalue, and, for example, this happens if the graph is bipartite: if $(S, V - S)$ a bipartition of the vertices such that all the edges go between $S$ and $V - S$, then we can construct an eigenvector of $-1$ by defining a vector $v$ such that $v_i = 1$ if $i \in S$ and $v_i = -1$ otherwise. Fortunately, it is a theorem that these are the only cases.

**Theorem 1** Let $G$ be an undirected, regular, connected, non-bipartite, graph of degree $d$ with $n$ vertices, and let $A$ be the random walk matrix of $G$. Then if $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$ are the eigenvalues of $A$, with multiplicities, in sorted order, we have $\lambda_1 = 1$ and $|\lambda_i| \leq 1 - 1/dn^2$ for $i = 2, \ldots, n$.

Finally, we are ready to study random walks in $G$, where $G$ is, as in the theorem above, an undirected, regular, connected, non-bipartite, graph of degree $d$ with $n$ vertices.

Let $A$ be the random walk matrix of $G$, let $1 > \lambda_2 \geq \cdots \geq \lambda_n$ be the eigenvalues of $A$ and $v_1, \ldots, v_n$ the respective eigenvectors, chosen to be orthogonal and of length 1. Note that $v_1$ is a multiple of the uniform distribution. Let $\lambda(G) = \max_{j=2}^{n} |\lambda_j|$ be the second largest eigenvalue in absolute value. Recall that $\lambda < 1 - 1/dn^2$.

Let $p \in \mathbb{R}^n$ be a distribution. We can write $p = u + (p - u)$, where $u := (1/n, \cdots, 1/n)$ is the uniform distribution and where $(p - u)$ is orthogonal to the uniform distribution. (It is easy to verify this last claim.) The vectors $v_2, \cdots, v_n$ are a basis for the set of vectors orthogonal to $u$, and so we can then write, for some coefficients $\alpha_2, \cdots, \alpha_n$,

$$p = u + \alpha_2 v_2 + \cdots + \alpha_n v_n.$$ 

Consider now $p \cdot A$, that is the distribution obtained by picking a vertex according to $p$ and then doing a one-step random walk. We have

$$p \cdot A = u \cdot A + \sum_{j=2}^{n} \alpha_j v_j A = u + \sum_{j=2}^{n} \alpha_j \lambda_j v_j.$$ 

in other words, the distribution that we obtain is uniform, up to an error term, and the
error term is $\sum_{j=2}^{n} \alpha_j \lambda_j v_j$. As a vector, how long is this error term? We can compute

$$\|p \cdot A - u\| = \left\| \sum_{j=2}^{n} \alpha_j \lambda_j v_j \right\|$$

$$= \sqrt{\sum_{j=2}^{n} \alpha_j^2 \lambda_j^2}$$

$$\leq \max_{j=2}^{n} |\lambda_j| \cdot \sqrt{\sum_{j=2}^{n} \alpha_j^2}$$

$$= \lambda \cdot ||p - u||$$

That is, the new distribution is closer by a factor of $\lambda$ to the uniform distribution. After $t$ steps we clearly have $||pA^t - u|| \leq \lambda^t ||p - u||$.

For a vertex $i \in V$, define $p_i = 1$ and $p_j = 0$ for $j \neq 1$. This corresponds to the distribution that gives probability one to vertex $i$. As before, we have $||pA^t - u|| \leq \lambda^t ||p - u||$ and it can be seen that $||p - u|| = \sqrt{1 - 1/n} < 1$. So we get

$$||pA^t - u|| \leq \lambda^t$$

Pick a value of $t$ such that $\lambda^t \leq 1/2n$, for example $t = O(\frac{1}{1-\lambda} \log n)$ is enough. Now we have $||pA^t - u|| < 1/2n$, from which it follows that every entry of $pA^t$ must be at least $1/2n$. This also implies that every vertex of $G$ is reachable from $i$ in $O(\frac{1}{1-\lambda} \log n)$ steps, and since $i$ was arbitrary we have actually proved that the diameter of $G$ is at most $O(\frac{1}{1-\lambda} \log n)$.

For general graphs, we only know $\lambda \leq 1 - 1/dn^2$, so the above proof only shows that the diameter is at most $O(dn^2 \log n)$, which is not very interesting. The argument does, however, show that if $\lambda$ is a constant, for example .9, then the diameter is logarithmic.

An expander is a graph such that $\lambda(G)$ is a constant bounded away from 1. An expander is a graph such that $\lambda(G)$ is a constant bounded away from 1.1 Expanders have a lot of applications in computer science in general and in complexity theory in particular. For this lecture, the only property we use is that the diameter is logarithmic.

It is known that there is a constant $d$ such that for every $n$ there is a $d$-regular expander with $n$ vertices and $\lambda(G) \leq 1/2$. In fact, it is known how to get $\lambda(G) \leq \sqrt{2d - 1/d}$ for certain values of $d$.

---

1This, of course, does not make sense as a definition because for fixed graph $\lambda$ is always a “constant bounded away from one.” In the formal definition, we think of an infinite family of graphs $\{G_n\}$, where $G_n$ has $n$ vertices, such that there is a fixed constant $\lambda$ such that $\lambda(G_n) \leq \lambda$ for every $n$. 

4
Notes for Lecture 19

In these notes we introduce Levin's theory of average-case complexity. This theory is still in its infancy: in these notes we will introduce the notion of “distributional problems,” discuss various formalizations of the notion of “algorithms that are efficient on average,” introduce a reducibility that preserves efficient average-case solvability, and finally prove that there is a problem that is complete for the distributional version of \( \text{NP} \) under such reductions. It is still an open question how to apply this theory to the study of natural distributional problems that are believed to be hard on average.

1 Distributional Problems

**Definition 1 (Distributional Problem)** A distributional problem is a pair \( \langle L, \mu \rangle \), where \( L \) is a decision problem and \( \mu \) is a distribution over the set \( \{0, 1\}^* \) of possible inputs.

In other settings where average-case hardness is considered (e.g. in the study of one-way functions) we normally describe the distribution of inputs as a collection of distributions \( \mu_1, \ldots, \mu_n, \ldots \) where \( \mu_n \) is a distribution over the set \( \{0, 1\}^n \) of inputs of a given input length. There are various reasons why this single-distribution approach is convenient for the purposes of this chapter. We will discuss it again later, but for now the basic intuition is that we will discuss reductions where the length of the output is not a function of the length of the input, so that sampling inputs from a fixed-length distribution and passing them to a reduction does not produce a fixed-length distribution (unlike the case of cryptographic reductions).

We will restrict to the study of distributional problems where \( \mu \) is “polynomial-time computable.” What do we mean by that? For all \( x \in \{0, 1\}^* \), let

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

where ‘\( \leq \)’ denotes lexicographic ordering. Then \( \mu \) must be computable in \( \text{poly}(|x|) \) time. Clearly this notion is at least as strong as the requirement that \( \Pr[x] \) be computable in polynomial time, because

\[
\Pr[x] = \mu^*(x) = \mu(x) - \mu(x - 1),
\]  

(2)

\( x - 1 \) being the lexicographic predecessor of \( x \). Indeed one can show that, under reasonable assumptions, 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

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

(3)

\footnote{One can always reduce the approach of distinct distributions to the approach of this chapter by assuming that \( \mu \) first picks at random a certain input length \( n \), and then it samples from \( \mu_n \).}
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.

2 DistNP

We define the complexity class

\[ \text{DistNP} = \{ \langle L, \mu \rangle : L \in NP, \mu \text{ polynomial-time computable} \} \]  

(4)

There are at least two good reasons for looking only at polynomial-time computable distributions.

1. One can show that there exists a distribution \( \mu \) such that every problem is as hard on average under \( \mu \) as it is in the worst case. Therefore, unless we place some computational restriction on \( \mu \), 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.’

3 Reductions

Definition 2 (Reduction) We say that a distributional problem \( \langle L_1, \mu_1 \rangle \) reduces to a distributional problem \( \langle L_2, \mu_2 \rangle \) (in symbols, \( \langle L_1, \mu_1 \rangle \leq \langle L_2, \mu_2 \rangle \)) if there exists a polynomial-time computable function \( f \) such that:

1. \( x \in L_1 \) \iff \( f(x) \in L_2 \).

2. There is a \( \epsilon > 0 \) such that, for every \( x \), \( |f(x)| = \Omega(|x|^\epsilon) \).

3. For all \( y \in \{0,1\}^* \),

\[
\sum_{x : f(x) = y} \mu_1(x) \leq \text{poly } (|y|) \mu_2(y). \tag{5}
\]

The first condition is the standard condition of many-to-one reductions in complexity theory: it ensures that an algorithm that is always correct for \( L_2 \) can be converted into an algorithm that is always correct for \( L_1 \). The second condition is a technical one, that is necessary to show that the reduction preserves efficient-on-average algorithms. All known reductions satisfies this condition naturally.

The third condition is called domination. To motivate this condition, consider that we want reductions to preserve the existence of algorithms that are efficient on average.
Suppose that we have an algorithm $A_2$ for problem $L_2$ such that, when we pick $y$ according to distribution $\mu_2$, $A(y)$ is efficient on average; if we want to solve $L_1$ under distribution $\mu_1$, then, starting from an input $x$ distributed according to $\mu_1(x)$, we compute $f(x)$ and then apply algorithm $A_2$ to $f(x)$. This will certainly be correct, but what about the running time? Intuitively, it could be the case that $A_2$ is very slow on some inputs, but such inputs are unlikely to be sampled according to distribution $\mu_2$; the domination condition ensures us that such inputs are also unlikely to be sampled when we sample $x$ according to $\mu_1$ and then consider $f(x)$.

4 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 some flawed definitions before settling on the best one and on an alternate one.

A first difficulty comes with the fact that we are dealing with a single distribution on all possible inputs. The most intuitive choice of saying that $A$ is efficient if $E[t(x)]$ is small is problematic because the expectation could be infinite even if $A$ runs in worst-case polynomial time.

One can work around this difficulty by defining $A$ to be polynomial provided that for some constant $c$ and for every sufficiently large $n$,

$$E[t(x) \mid |x| = n] \leq n^c$$

However we chose to define distributional problems and reducibility without separating problems by input size, and we would run into several difficulties in separating them now. Besides, it seems reasonable that there could be input lengths on which $A$ takes a long time, but that are generated with very low probability under $\mu$; in such cases $A$ may still be regarded as efficient, but this is not taken into account in the above definition.

Our next attempt folds the polynomial running time into the single distribution by defining $A$ to be polynomial on average if there is a polynomial $p$ such that

$$E \left[ \frac{t(x)}{p(|x|)} \right] = O(1)$$

This definition is quite appealing, but is still subject to the fatal flaw of not being robust, in that: (1) reductions do not preserve this definition of polynomial solvability on average and (2) the definition is sensitive to trivial representation changes such as replacing a matrix representation of a graph by an adjacency list.

To see why these problems arise, let $\mu$ be the uniform distribution, and let

$$t(x) = 2^n \text{ if } x = \overline{0}, \ t(x) = n^2 \text{ otherwise.} \quad (6)$$

The average running time is about $n^2$. But suppose now that $n$ is replaced by $2n$ (because of a change in representation, or because of the application of a reduction), then

$$t(x) = 2^{2n} \text{ if } x = \overline{0}, \ t(x) = 4 \cdot n^2 \text{ otherwise.} \quad (7)$$
Similarly, if \( t(x) \) is replaced by \( t^2(x) \), the average running time becomes exponential.

We now come to a satisfying definition.

**Definition 3 (Polynomial on average)** Suppose \( A \) is an algorithm for a distributional problem \( \langle L, \mu \rangle \) that runs in time \( t(x) \) on input \( x \). We say that \( A \) has polynomial running time on average is there is a constant \( c \) such that

\[
E \left[ \frac{t(x)^{1/c}}{|x|} \right] = O(1)
\]

Notice, first, that this definition is satisfied by any algorithm that runs in worst-case polynomial time. If \( t(x) = O(|x|^c) \), then \( t(x)^{1/c} = O(|x|) \) and the sum converges. More interestingly, suppose \( t() \) is a time bound for which the above definition is satisfied; then an algorithm whose running time is \( t'(x) = t(x)^2 \) also satisfies the definition, unlike the case of the previous definition. In fact we have the following result, whose non-trivial proof we omit.

**Theorem 1** If \( \langle L_1, \mu_1 \rangle \leq \langle L_2, \mu_2 \rangle \) and \( \langle L_2, \mu_2 \rangle \) admits an algorithm that is polynomial on average, then \( \langle L_1, \mu_1 \rangle \) also admits an algorithm that is polynomial on average.

There is an additional interesting property of the definition of polynomial of average: there is a high probability that the algorithm runs in polynomial time. Suppose that

\[
E \left[ \frac{t(x)^{1/c}}{|x|} \right] = O(1) \tag{8}
\]

and that we wish to compute \( \Pr[t(x) \geq k \cdot |x|^c] \). Such a probability is clearly the same as

\[
\Pr[t(x)^{1/c} \geq k^{1/c}|x|]
\]

and by Markov’s inequality this is at most \( O(1/k^{1/c}) \), which can be made fairly small by picking \( k \) large enough. Since the algorithm runs in time at most \( kn^c \) for a subset of inputs having probability \( 1 - k^{-1/c} \), we see that our definition gives a smooth quantitative tradeoff for how much time we need to solve an increasing fraction of inputs.

In the setting of one-way functions and in the study of the average-case complexity of the permanent and of problems in \( \text{EXP} \) (with applications to pseudorandomness), we normally interpret “average case hardness” in the following way: that an algorithm of limited running time will fail to solve the problem on a noticeable fraction of the input. Conversely, we would interpret average-case tractability as the existence of an algorithm that solves the problem in polynomial time, except on a negligible fraction of inputs. This leads to the following formal definition.

**Definition 4 (Heuristic polynomial time)** We say that an algorithm \( A \) is a heuristic polynomial time algorithm for a distributional problem \( \langle L, \mu \rangle \) if \( A \) always runs in polynomial time and for every polynomial \( p \)

\[
\sum_{x: A(x) \neq \chi_L(x)} \mu'(x)p(|x|) = O(1)
\]
In other words, a polynomial time algorithm for a distributional problem is a heuristic if the algorithm fails on a negligible fraction of inputs, that is, a subset of inputs whose probability mass is bounded even if multiplied by a polynomial in the input length. It might also make sense to consider a definition in which $A$ is always correct, although it does not necessarily work in polynomial time, and that $A$ is heuristic polynomial time if there is a polynomial $q$ such that for every input length $n$, time $O(n^{1/2})$ steps. Our definition is only more general, because from an algorithm $A$ as before one can obtain an algorithm $A$ satisfying Definition 4 by adding a clock that stops the computation after $q(|x|)$ steps.

The definition of heuristic polynomial time is incomparable with the definition of average polynomial time. For example, an algorithm could take time $2^n$ on a fraction $1/n^{log n}$ of the inputs of length $n$, and time $n^2$ on the remaining inputs, and thus be a heuristic polynomial time algorithm with respect to the uniform distribution, while not being average polynomial time algorithm with respect to the uniform distribution. On the other hand, consider an algorithm such that for every input length $n$, and for $1 \leq k \leq 2^{n/2}$, there is a fraction about $1/k^2$ of the inputs of length $n$ on which the algorithm takes time $\Theta(kn)$. Then this algorithm satisfies the definition of average polynomial time under the uniform distribution, but if we impose a polynomial clock there will be an inverse polynomial fraction of inputs of each length on which the algorithm fails, and so the definition of heuristic polynomial time cannot be met.

It is easy to see that heuristic polynomial time is preserved under reductions.

**Theorem 2** If $\langle L_1, \mu_1 \rangle \leq \langle L_2, \mu_2 \rangle$ and $\langle L_2, \mu_2 \rangle$ admits a heuristic polynomial time algorithm, then $\langle L_1, \mu_1 \rangle$ also admits a heuristic polynomial time algorithm.

**Proof:** Let $A_2$ be the algorithm for $\langle L_2, \mu_2 \rangle$, let $f$ be the function realizing the reduction, and let $p$ be the polynomial witnessing the domination property of the reduction. Let $c$ and $\epsilon$ be such that for every $x$ we have $|x| \leq c|f(x)|^{1/\epsilon}$.

Then we define the algorithm $A_1$ than on input $x$ outputs $A_2(f(x))$. Clearly this is a polynomial time algorithm, and whenever $A_2$ is correct on $f(x)$, then $A_1$ is correct on $x$. We need to show that for every polynomial $q$

$$\sum_{x:A_2(f(x)) \neq \chi_{L_2}(f(x))} \mu_1'(x)q(|x|) = O(1)$$

and the left-hand side can be rewritten as

$$\sum_{y:A_2(y) \neq \chi_{L_2}(y)} \sum_{x:f(x)=y} \mu_1'(x)q(|x|)$$

$$\leq \sum_{y:A_2(y) \neq \chi_{L_2}(y)} \sum_{x:f(x)=y} \mu_1'(x)q(c \cdot |y|^{1/\epsilon})$$

$$\leq \sum_{y:A_2(y) \neq \chi_{L_2}(y)} \mu_2'(y)p(|y|)q'(|y|)$$

$$= O(1)$$

where the last step uses the fact that $A_2$ is a polynomial heuristic for $\langle L_2, \mu_2 \rangle$ and in the second-to-last step we introduce the polynomial $q'(n)$ defined as $q(c \cdot n^{1/\epsilon})$. 

5
5 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 \(\langle M, x, 1^t, 1^l \rangle\), where \(M\) is an encoding of a Turing machine and \(1^t\) is a sequence of \(t\) ones. Then we define the following “universal” problem \(U\). 

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

That \(U\) is \text{NP}-complete follows directly from the definition. Recall the definition of \text{NP}: we say that \(L \in \text{NP}\) if 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 \(U\) we need only map \(x\) onto \(R(x) = \langle M, x, 1^t, 1^l \rangle\) where \(t\) and \(l\) are sufficiently large bounds.

To give a reduction that satisfies the domination condition is indeed harder. Let \(\langle L, \mu \rangle \in \text{DistNP}\). Define a uniform distribution over the \(\langle M, x, 1^t, 1^l \rangle\) as follows:

\[
\mu'(\langle M, x, 1^t, 1^l \rangle) = \frac{1}{|M|(|M| + 1)2^{|M|}} \cdot \frac{1}{|x|(|x| + 1)2^{|x|}} \cdot \frac{1}{(t + l)(t + l + 1)}. \tag{9}
\]

The trouble is that, because of the domination condition, we can’t map \(x\) onto \(R(x)\) if \(\mu'(x) > \text{poly}(|x|)2^{-|x|}\). We work around this problem by compressing \(x\) to a shorter string if \(\mu'(x)\) 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 3** Suppose \(\mu\) 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. \(|C(x)| \leq 1 + \min \left\{|x|, \log \frac{1}{\mu(x)} \right\}\).

**Proof:** If \(\mu'(x) \leq 2^{-|x|}\) then simply let \(C(x) = 0x\), that is, \(0\) concatenated with \(x\). If, on the other hand, \(\mu'(x) > 2^{-|x|}\), 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 \(\mu\) 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 \(\mu'(x) \leq 2^{-|x|}, |C(x)| \leq 1 + \log \frac{1}{\mu(x)}\). \(\Box\)

Now the reduction is to map \(x\) onto \(R_2(x) = \langle \overline{M}, C(x), 1^t, 1^{1+|x|} \rangle\). 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 \( \mu_2 (x) = \Pr [R_2 (x)] \). Then, since the map is one-to-one, we need only show that \( \mu (x) \leq \text{poly} (|x|) \mu_2 (x) \). Since \( t = O (\text{poly} (t)) \),

\[
\mu_2 (x) = \frac{1}{O \left( \frac{|M|^2}{2^{|M|}} \right)} \cdot \frac{1}{O \left( \frac{|C(x)|^2}{2^{|C(x)|}} \right)} \cdot \frac{1}{O \left( t + l + |x| \right)}
\]

\[
\geq \text{poly} (|x|) \max \left( 2^{-|x|}, \mu (x) \right)
\]

\[
\geq \text{poly} (|x|) \mu (x)
\]

and we are done.

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

6 Polynomial-Time Samplability

**Definition 5 (Samplable distributions)** We say that a distribution \( \mu \) is polynomial-time samplable if there exists a probabilistic algorithm \( A \), taking no input, that outputs \( x \) with probability \( \mu (x) \) and runs in \( \text{poly} (|x|) \) time.

Any polynomial-time computable distribution is also polynomial-time samplable, provided that for all \( x \),

\[
\mu (x) \geq 2^{-\text{poly}(|x|)} \text{ or } \mu (x) = 0. \tag{10}
\]

For a polynomial-time computable \( \mu \) satisfying the above property, we can indeed construct a sampler \( A \) that first chooses a real number \( r \) uniformly at random from \([0, 1]\), to \( \text{poly} (|x|) \) bits of precision, and then uses binary search to find the first \( x \) such that \( \mu (x) \geq r \).

On the other hand, under reasonable assumptions, there are efficiently samplable distributions \( \mu \) that are not efficiently computable.

In addition to \( \text{DistNP} \), we can look at the class

\[
\langle \text{NP}, \text{P-samplable} \rangle = \{ \langle L, \mu \rangle : L \in \text{NP}, \mu \text{ polynomial-time samplable} \}. \tag{11}
\]

A result due to Impagliazzo and Levin states that if \( \langle L, \mu \rangle \) is \( \text{DistNP} \)-complete, then \( \langle L, \mu \rangle \) is also complete for the class \( \langle \text{NP}, \text{P-samplable} \rangle \).

This means that the completeness result established in the previous section extends to the class of NP problems with samplable distributions. The completeness, however, is proved under a different notion of reducibility, that preserves heuristic but not average polynomial time algorithms.

7 References

Levin’s theory of average-case complexity was introduced in [Lev86]. Ben-David et al. [BDCGL92] prove several basic results about the theory. Impagliazzo and Levin [IL90] show that the theory can be generalized to samplable distributions. Impagliazzo [Imp95] wrote a very clear survey on the subject. Another good reference is a survey paper by Goldreich [Gol97].
References


Exercises

1. For a parameter $c$, consider the distribution $D_{n,cn}$ over instances of 3SAT with $n$ variables generated by picking $cn$ times independently a random clause out of the $8\binom{n}{3}$ possible clauses that can be constructed from $n$ variables. (Note that the same clause could be picked more than once.) Let $D_c$ be the distribution obtained by first picking a number $n$ with probability $1/n(n + 1)$ and then sampling from $D_{n,cn}$.

(a) Show that an instance from $D_{n,cn}$ is satisfiable with probability at least $(7/8)^{cn}$ and at most $2^n \cdot (7/8)^{cn}$.

(b) Argue that, using the definition given in this lecture, $D_{15}$ cannot be reduced to $D_{30}$.

[Hint: take a sufficiently large $n$, and then look at the probability of satisfiable instances of length $n$ under $D_{15}$ and the probability that their image is generated by $D_{30}$.]
Notes for Lecture 15

Notes written 12/07/04

Learning Decision Trees

In these notes it will be convenient to denote bits using the set \{-1,1\} instead of \{0,1\}.

1 Decision Trees

A decision tree accepts or rejects input strings \(x_1, \ldots, x_n\) and can be described as a directed tree where every internal node has two children. The computational path begins at the root. Each non-final vertex is labeled with a variable \(x_i\) and has two outgoing edges, one labeled 1 and the other \(-1\). The path follows edge \(b\) such that \(x_i = b\), and ends at a leaf. Each leaf is labeled \(-1\) or 1. Such a label is called the label of the leaf. The output of a decision tree \(T\) on input \(x\) is the value of the leaf reached in the computational path of \(T\) given \(x\).

The size of a decision tree is the number of nodes. Note that in a tree where every internal node has exactly two children it is always true that the number of internal vertices is equal to the number of leaves minus 1. Therefore measuring size in terms of number of internal vertices, number of leaves, or total number of vertices are all equivalent measures to within a constant factor.

2 Fourier Analysis

For every \(S \subseteq \{1, \ldots, n\}\), define the function \(u_S : \{-1,1\}^n \to \mathbb{R}\) as

\[ u_S(x_1, \ldots, x_n) = \prod_{i \in S} x_i \]

If \(S = \emptyset\), then we define \(u_\emptyset(x) = 1\) for every \(x\).

Given two functions \(f, g : \{-1,1\}^n \to \mathbb{R}\), define their inner product as

\[ f \cdot g = \frac{1}{2^n} f(x)g(x) = \mathbb{E}_x[f(x)g(x)] \]

Then we have that for every set \(S\)

\[ u_S \cdot u_S = 1 \]

In fact, more generally, \(f \cdot f = 1\) for every function \(f : \{-1,1\}^n \to \{-1,1\}\).

Furthermore, if \(S \neq T\), we have

\[ u_S \cdot u_T = 0 \]
Which implies that the functions $u_S$ are linearly independent over the reals. Indeed, if there were a linear combination

$$u_S = \sum_{T \neq S} \alpha_T u_T$$

then we would also have

$$1 = u_S \cdot u_S = (\sum_{T \neq S} \alpha_T u_T) \cdot u_S = \sum_{T \neq S} \alpha_T u_T \cdot u_S = 0$$

The set of functions $f : \{-1, 1\}^n \rightarrow \mathbb{R}$ is a $2^n$-dimensional vector space over the reals, and since there are $2^n$ functions $u_S$, all linearly independent, they must form a basis. Indeed, with respect to the inner product we have defined, the $u_S$ form an orthonormal basis.

Every function $f : \{-1, 1\}^n \rightarrow \{-1, 1\}$ can then be written as a linear combination

$$f(x) = \sum_S \hat{f}_S u_S(x)$$

of the functions $u_S$. The coefficients $\hat{f}_S$ are called the Fourier coefficients of $f$.

We will need a few properties of the coefficients. First, we note that

$$\hat{f}_S = f \cdot u_S$$

which is a standard property of inner-product spaces with an orthonormal basis.

Next, we have

**Lemma 1 (Parseval’s Equality)** For every function $f : \{-1, 1\}^n \rightarrow \{-1, 1\}$,

$$\sum_{S} \hat{f}_S^2 = \mathbb{E}_x[f^2(x)].$$

In particular, if $f : \{-1, 1\}^n \rightarrow \{-1, 1\}$ is boolean, then $\sum_S \hat{f}_S^2 = 1$.

Finally, if $f : \{-1, 1\}^n \rightarrow \{-1, 1\}$ is a boolean function, then

$$\hat{f}_S = \mathbb{E}[f(x)u_S(x)] = \mathbb{P}[f(x) = u_S(x)] - \mathbb{P}[f(x) \neq u_S(x)] = 2\mathbb{P}[f(x) = u_S(x)] - 1 \quad (2)$$

Note also that $u_S(x_1, \ldots, x_n) = \bigoplus_{i \in S} x_i$ if we adopt the convention that 1 is False and −1 is True. Therefore, $\hat{f}_S \geq \epsilon$ if and only if $f$ has agreement at least $1/2 + \epsilon/2$ with the “linear” function $\bigoplus_{i \in S} x_i$ (where we mean linear with respect to operations over bits, not linear over the reals).

This observation, together with Parseval’s equality, implies that there are at most $1/\epsilon^2$ linear functions that have agreement at least $1/2 + \epsilon/2$ with a given boolean function.

We can then formulate the following version of the Goldreich-Levin theorem [GL89].

**Lemma 2 (Goldreich-Levin, revised form)** There is a probabilistic algorithm $GL$ that given oracle access to a function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ and given a threshold parameter $\tau > 0$, an accuracy parameter $\gamma > 0$ and a confidence parameter $\delta > 0$, runs in time polynomial in $n$, $1/\tau$, $1/\gamma$ and $\log 1/\delta$, and outputs a list of sets $S_1, \ldots, S_t$ and of numbers $\hat{f}_{S_1}, \ldots, \hat{f}_{S_t}$, such that, with probability at least $1 - \delta$ the following conditions hold:
• Every set $S$ such that $|\hat{f}_S| \geq \tau$ is in the list;

• For every set $S$ in the list, $|\hat{f}_S - \bar{f}_S| \leq \gamma$.

**Proof:** In the standard version, the Goldreich-Levin algorithm is given a parameter $\epsilon$ and it produces a list of size $O(1/\epsilon^2)$ such that, for each $S$ such that $\hat{f}_S \geq 2\epsilon$, there is a probability at least $3/4$ that $S$ is in the list.

We first run the Goldreich-Levin algorithm with parameter $\tau/2$ independently $O(\log 1/\tau\delta)$ times, and take the union of the lists. Now, for each of the $\leq 1/\tau^2$ sets $S$ such that $\hat{f}_S \geq \tau$, the set $S$ has a probability at most $1/4$ of being missed in each iteration, and a probability at most, say, $\tau^2\delta/4$ of being missed every time. By a union bound, there is a probability at least $1 - \delta/4$ that the final list contains every set $S$ such that $\hat{f}_S \geq \tau$.

We then repeat the same operations, but using $-f$ as an oracle. This gives another list of size $O(\tau^{-2}\log \tau^{-1}\delta^{-1})$ that contains all the sets $S$ such that $\hat{f}_S \leq -\tau$, except with probability at most $\delta/4$.

We take the union of the two lists, and define it to be $L$. Except with probability at most $\delta/2$, the list satisfies the first condition.

For every set $S \in L$, we pick $t = O(\gamma^{-2}\log L/\delta)$ sample points $x^1, \ldots, x^t$ in $\{-1,1\}^n$, and define

$$\bar{f}_S = \frac{1}{t} \sum_i f(x^i)u_S(x^i)$$

(3)

By Chernoff bounds, there is a probability at most $\delta/2L$ that $\bar{f}_S$ differs from $\hat{f}_S$ by more than $\gamma$. Taking a union bound over the sets, we see that the numbers $\bar{f}_S$ satisfy the second condition, except with probability at most $\delta/2$. \(\square\)

### 3 Overview of the Proof

Our main result is the following.

**Theorem 3** There is a probabilistic learning algorithm $A$ that given oracle access to a function $f : \{0,1\}^n \rightarrow \{0,1\}$ that can be computed by a decision tree of size $S$, and given $S$ and parameters $\epsilon, \delta > 0$, runs in time polynomial in $n, S, 1/\epsilon, \log 1/\delta$ and outputs a circuit $C$ that, with probability at least $1 - \delta$, is $\epsilon$-close to $f$.

We prove it in two steps. We show that every function whose Fourier coefficients satisfy a certain condition can be efficiently learned, and then we show that functions computed by small decision trees satisfy the condition.

**Lemma 4** There is a probabilistic learning algorithm $A$ that given oracle access to a function $f : \{0,1\}^n \rightarrow \{0,1\}$, given a number $m$ such that

$$\sum_S |\hat{f}_S| \leq m$$

and given $\epsilon, \delta > 0$, runs in time polynomial in $n, m, 1/\epsilon, \log 1/\delta$ and outputs a circuit $C$ that, with probability at least $1 - \delta$, is $\epsilon$-close to $f$. 

3
The number $\sum S |\hat{f}_S|$ is called the $\ell_1$-norm of the function $f$, and is also denoted by $||f||_1$. Lemma 4 says that functions of polynomial $\ell_1$-norm can be learned in polynomial time.

**Lemma 5** If $f$ can be computed by a decision tree of size $S$, then the $\ell_1$ norm of $f$ is at most $S$.

### 4 Proof of Lemma 4

We fix $\tau = \epsilon / 2L$. If $\ell = O(\tau^{-2} \log(\tau^{-1} \delta^{-1})$ is an upper bound to the size of the list returned by the Goldreich-Levin algorithm with threshold $\tau$ and confidence $\delta$, then we fix $\gamma = \sqrt{\epsilon / 2\ell}$ and we run the Goldreich-Levin algorithm with threshold $\tau$, confidence $\delta$ and accuracy $\gamma$. We find a list $L$ of sets and values $\bar{f}_S$ for each set in the list such that, with probability $\geq 1 - \delta$ over the internal coin tosses of the algorithm, we have:

- Every set $S$ such that $|\hat{f}_S| \geq \tau$ is in the list;
- For every set $S$ in the list, $|\hat{f}_S - \bar{f}_S| \leq \gamma$.

Then we define the function $h(x) = \sum_{S \in L} \bar{f}_S u_S$. The Fourier coefficients of the difference $d(x) := f(x) - h(x)$ are as follows.

- If $S \not\in L$, then $\hat{d}_S = 0$ and so $\hat{d}_S = \hat{f}_S$, and also $|\hat{d}_S| = |\hat{f}_S| \leq \tau$.
- If $S \in L$, then $|\hat{h}_S| \leq \gamma$.

We now want to estimate $E[(f(x) - h(x))^2]$, which is a measure of how good is $h$ as an approximation to $f$. We have

\[
E[(f(x) - h(x))^2] = E[d^2(x)] = \sum_S \hat{d}_S^2 = \sum_{S \not\in L} \hat{d}_S^2 + \sum_{S \in L} \hat{d}_S^2 \\
\leq \tau \sum_{S \not\in L} |\hat{d}_S| + \sum_{S \in L} \gamma^2 \\
\leq \tau m + |L| \gamma^2 \leq \epsilon
\]

Define $g : \{-1, 1\}^n \to \{-1, 1\}$ such that $g(x) = 1$ if $h(x) \geq 0$ and $g(x) = -1$ if $h(x) < 0$. We see that

\[
\Pr[g(x) \neq f(x)] \leq E_x[(f(x) - h(x))^2] \leq \alpha + \epsilon
\]

because every $x$ such that $g(x) \neq f(x)$ must also be such that $|f(x) - h(x)| \geq 1$ and so $(f(x) - h(x))^2 \geq 1$.

We output the circuit that computes $g$. 


5 Proof of Lemma 5

First note that if \( f, g, h : \{-1, 1\}^n \to \mathbb{R} \) are functions such that \( f = g + h \), then \( \hat{f}_S = \hat{g}_S + \hat{h}_S \) for every set \( S \), and we have
\[
||f||_1 \leq ||g||_1 + ||h||_1 \tag{4}
\]

Let \( f \) be a function computed by a decision tree \( T \) with \( m \) leaves. We assume that tests are never repeated in a computational path of \( T \), otherwise we can get a smaller tree \( T \) for \( f \) satisfying such a property.

For every leaf \( v \) of the decision tree, let \( val(v) \) be the output associated with that leaf, \( d(v) \) the depth of \( v \) and \( I_v \) the set of inputs \( x \) such that computation of \( T \) on input \( x \) ends at \( v \). The set \( I_v \) clearly contains a \( 1/2^{d(v)} \) fraction of \( \{-1, 1\}^n \) (this is where we are using the assumption that in each computational path we test distinct inputs). We also define \( var(v) \subseteq \{1, \ldots, n\} \) the be the set of indices \( i \) such that a test for \( x_i \) is in the path from the root to \( v \). Note that \( d(v) = |var(v)| \). Finally, we define \( f^v(x) \) to be \( val(v) \) if the computation of \( T \) on input \( x \) ends at \( v \), and zero otherwise.

By definition:
\[
f(x) = \sum_{v \text{ leaf}} f^v(x)
\]

For a set \( S \) and a leave \( v \),
\[
\hat{f}^v_S = \mathbb{E}_x f^v(x) u_S(x) = \frac{1}{2^{d(v)}} \mathbb{E}_{x \in I_v} f^v(x) u_S(x) = \frac{1}{2^{d(v)}} \mathbb{E}_{x \in I_v} u_S(x)
\]

From which we see that \( \hat{f}^v_S = 0 \) if \( S \not\subseteq var(v) \), and \( |\hat{f}^v_S| = 1 \) if \( S \subseteq var(v) \). Since there are \( 2^{d(v)} \) subsets of \( var(v) \), we get
\[
||f^v||_1 = \sum_S |\hat{f}^v_S| = \sum_{S \subseteq var(v)} \frac{1}{2^{d(v)}} = 1
\]

And so
\[
||f||_1 \leq \sum_{v \text{ leaf}} ||f^v||_1 \leq m
\]

6 References

The results of these notes are due to Kushilevitz and Mansour [KM93].

References


Notes for Lecture 14 v0.9

These notes are in a draft version. Please give me any comments you may have, because this will help me revise them.

Today we will prove the Goldreich-Levin theorem:

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

The asymptotic version of the Theorem follows from the following finite version, that we state directly in the counterpositive direction.

**Theorem 2** Let $f : \{0, 1\}^n \to \{0, 1\}^n$ be a bijection computable by a circuit of size $t$, and suppose that there is a circuit $C$ of size $S$ such that

$$\Pr_{x, r}[C(f(x), r) = x \cdot r] \geq \frac{1}{2} + \epsilon$$

Then there is a circuit $C'$ of size $O((S + t) \cdot \text{poly}(n, 1/\epsilon))$ such that

$$\Pr_x[C'(f(x)) = x] \geq \frac{\epsilon}{4}$$

In turn, Theorem 2 follows from the existence of the following algorithm.

**Lemma 3** There is a probabilistic algorithm that, given a parameter $\epsilon$ and oracle access to a function $g : \{0, 1\}^n \to \{0, 1\}$, runs in time $O(n^2 \epsilon^{-2} \log n)$, makes $O(n \epsilon^{-4} \log n)$ oracle accesses, and outputs a list of $O(1/\epsilon^2)$ elements of $\{0, 1\}^n$.

If $x$ is such that $Pr_r[B(r) = x \cdot r] \geq \frac{1}{2} + \epsilon$, then there is a probability at least $1/2$ that $x$ is in the output list.

1 Proof of Theorem 2 Using Lemma 3

From the assumption that

$$\Pr_{x, r}[C(f(x), r) = x \cdot r] \geq \frac{1}{2} + \epsilon$$

it follows that

$$\Pr_x\left[\Pr_r[C(f(x), r) = x \cdot r] \geq \frac{1}{2} + \frac{\epsilon}{2}\right] \geq \frac{\epsilon}{2}$$

Let us call an $x$ such that $Pr_r[C(f(x), r) = x \cdot r] \geq 1/2 + \epsilon/2$ a good $x$, and, for a given $x$, let us denote by $B_x$ the function defined as $B_x(r) = C(f(x), r)$. 

1
For a good $x$, if we apply the algorithm of Lemma 3 to the function $B_x$ with parameter $\epsilon/2$, we obtain a list that, with probability at least $1/2$, contains $x$.

Consider now the following algorithm: given a string $y$, define $B(r) := C(y, r)$ and run the algorithm of Lemma 3 with function $B()$ and parameter $\epsilon/2$. Once the algorithm outputs a list $x^1, x^2, \ldots, x^{O(1/\epsilon^2)}$, compute $f(x^i)$ for each $i$, and output the $x^i$ such that $f(x^i) = y$, if any.

If we pick $x$ at random and give $f(x)$ to the above algorithm, there is a probability at least $\epsilon/2$ that $x$ is good and, if so, there is a probability at least $1/2$ that $x$ is in the list. Therefore, there is a probability at least $\epsilon/4$ that the algorithm inverts $f()$, where the probability is over the choices of $x$ and over the internal randomness of the algorithm. In particular, there is a fixed choice of the internal randomness of the algorithm that results in inverting $f()$ on an $\epsilon/4$ fraction of the inputs. Finally, we convert the algorithm into a circuit, and the resulting circuit is $C'$.

2 Proof of Lemma 3

2.1 A Special Case First

We start by considering the simpler case in which we are given oracle access to a function $B()$ such that $\Pr_r[B(r) = x \cdot r] \geq 7/8$ and we want to find $x$.

If we denote by $e_i$ the vector that has a 1 in the $i$-th coordinate and 0s in other coordinate, we see that $x_i = x \cdot e_i$. Furthermore, for every $r$, we have $x_i = x \cdot (r \oplus e_i) \oplus x \cdot r$ because of the linearity of the $\cdot$ operator and of the fact that $r \oplus r$ is the all zero vector.

Consider now the process of picking a random $r$ and computing $B(r \oplus e_i) \oplus B(r)$. Except with probability at most $1/8$, $B(r \oplus e_i) = x \cdot (r \oplus e_i)$ and, except with probability at most $1/8$, $B(r) = x \cdot r$. Therefore, with probability at least $3/4$,

$$B(r \oplus e_i) \oplus B(r) = x \cdot (r \oplus e_i) \oplus x \cdot r = x_i$$

This suggests the following algorithm:

Algorithm $A_7$:

for $i := 1$ to $n$
    pick $k = O(\log n)$ random elements $r^1, \ldots, r^k \in \{0, 1\}^n$
    compute:
    \[B(r^1 \oplus e_1) \oplus B(r^1)\]
    \[B(r^2 \oplus e_2) \oplus B(r^2)\]
    \[\vdots\]
    \[B(r^k \oplus e_n) \oplus g(r^n)\]
    assign to $x_i$ the value occurring in the majority of these computations

return $x$

To analyze the algorithm, note that, for a particular value of $i$, we expect to get the right value of $x_i$ in a fraction $3/4$ of the $k$ trials, and the algorithm derives the correct value of
provided that more than half of the $k$ trials are correct. Then, by a Chernoff bound, the probability of estimating $x_i$ incorrectly is $e^{-\Omega(k)}$. We can then choose $k = O(\log n)$ and make sure that the error probability is at most, say, $1/100n$, and hence we conclude that the output of the algorithm is correct with probability at least $99/100$.

We note that the running time of this program is $O(n^2k) = O(n^2\log n)$ and that it makes $O(nk) = O(n\log n)$ oracle accesses.

### 2.2 The General Case

Consider now the general case. We are given an oracle $B()$ such that $B(r) = x \cdot r$ for an $1/2 + \epsilon$ fraction of the $r$. Our goal will be to use $B()$ to simulate an oracle that has agreement $7/8$ with $x \cdot r$, so that we can use the algorithm of the previous section to find $x$. We perform this “reduction” by “guessing” the value of $x \cdot r$ at a few points.

We first choose $t$ random points $r^1, \ldots, r^t \in \{0, 1\}^n$ where $t = O(1/\epsilon^2)$. For the moment, let us suppose that we have “magically” obtained the values $x \cdot r^1, \ldots, x \cdot r^k$. Then define $B'(r)$ as the majority value of:

$$x \cdot r^j \oplus B(r \oplus r^j) \quad j = 1, 2, \ldots, t \quad (1)$$

For each $j$, the above expression equals $x \cdot r$ with probability at least $1/2 + \epsilon$ (over the choices of $r^j$) and by choosing $t = O(1/\epsilon^2)$ we can ensure that

$$\Pr_{r, r^1, \ldots, r^k} [B'(r) = x \cdot r] \geq \frac{31}{32} \quad (2)$$

from which it follows that

$$\Pr_{r^1, \ldots, r^k} \left[ \Pr_r [B'(r) = x \cdot r] \geq \frac{7}{8} \right] \geq \frac{3}{4} \quad (3)$$

Consider the following algorithm.

**Algorithm GL-First-Attempt:**

pick $r^1, \ldots, r^t \in \{0, 1\}^k$ where $t = O(1/\epsilon^2)$

for all $b_1, \ldots, b_t \in \{0, 1\}$

- define $B'_{b_1 \ldots b_t}(r)$ as majority of: $b_j \oplus B(r \oplus r^j)$

- apply Algorithm $A_{\frac{7}{8}}$ to $B'_{b_1 \ldots b_t}$

- add result to list

The idea behind this program is that we do not in fact know the values $x \cdot r^j$, but we can “guess” them by considering all choices for the bits $b_j$. If $B(r)$ agrees with $x \cdot r$ for at least a $1/2 + \epsilon$ fraction of the $r$s, then there is a probability at least $3/4$ that in one of the iteration we invoke algorithm $A_{\frac{7}{8}}$ with a simulated oracle that has agreement $7/8$ with $x \cdot r$. Therefore, the final list contains $x$ with probability at least $3/4 - 1/100 > 1/2$.

The obvious problem with this algorithm is that its running time is exponential in $t = O(1/\epsilon^2)$ and the resulting list may also be exponentially larger than the $O(1/\epsilon^2)$ bound promised by the Lemma.

To overcome these problems, consider the following similar algorithm.
Algorithm GL:

pick $r^1, \ldots, r^l \in \{0, 1\}^k$ where $l = \log O(1/\epsilon^2)$
define $r_S := \bigoplus_{j \in S} r^j$ for each non-empty $S \subseteq \{1, \ldots, l\}$
for all $b_1, \ldots, b_l \in \{0, 1\}$
define $b_S := \bigoplus_{j \in S} b_j$ for each non-empty $S \subseteq \{1, \ldots, l\}$
define $B'_{b_1 \ldots b_l}(r)$ as majority over non-empty $S \subseteq \{1, \ldots, l\}$ of $b_S \oplus B(r \oplus r_S)$
run Algorithm $A^*_l$ with oracle $B'_{b_1 \ldots b_l}$
add result to list

Let us now see why this algorithm works. First we define, for any nonempty $S \subseteq \{1, \ldots, l\}$, $r_S = \bigoplus_{j \in S} r^j$. Then, since $r^1, \ldots, r^l \in \{0, 1\}^k$ are random, it follows that for any $S \neq T$, $r_S$ and $r_T$ are independent and uniformly distributed. Now consider an $x$ such that $x \cdot r$ and $B(r)$ agree on a $\frac{1}{2} + \epsilon$ fraction of the values of $r$. Then for the choice of $\{b_j\}$ where $b_j = x \cdot r^j$ for all $j$, we have that

$$b_S = x \cdot r_S$$

for every non-empty $S$. In such a case, for every $S$ and every $r$, there is a probability at least $\frac{1}{2} + \epsilon$, over the choices of the $r^j$ that

$$b_S \oplus B(r \oplus r_S) = x \cdot r,$$

and these events are pair-wise independent. Note the following simple lemma.

**Lemma 4** Let $R_1, \ldots, R_t$ be a set of pairwise independent 0−1 random variables, each of which is 1 with probability at least $\frac{1}{2} + \epsilon$. Then $\Pr[\sum_i R_i \geq t/2] \geq 1 - \frac{1}{4\epsilon^2 t}$.

**Proof:** Let $R = R_1 + \cdots + R_t$. The variance of a 0/1 random variable is at most 1/4, and, because of pairwise independence, $\text{Var}[R] = \text{Var}[R_1 + \cdots + R_t] = \sum_i \text{Var}[R_i] \leq t/4$.

We then have

$$\Pr[R \leq t/2] \leq \Pr[\|R - E[R]\| \geq \epsilon t] \leq \frac{\text{Var}[R]}{\epsilon^2 t^2} \leq \frac{1}{4\epsilon^2 t}$$

$\square$

Lemma 4 allows us to upper-bound the probability that the majority operation used to compute $B'$ gives the wrong answer. Combining this with our earlier observation that the $\{r_S\}$ are pairwise independent, we see that choosing $l = 2 \log 1/\epsilon + O(1)$ suffices to ensure that $B'_{b_1 \ldots b_l}(r)$ and $x \cdot r$ have agreement at least $7/8$ with probability at least $3/4$. Thus we can use Algorithm $A^*_l$ to obtain $x$ with high probability. Choosing $l$ as above ensures that the list generated is of length at most $2^l = O(1/\epsilon^2)$ and the running time is then $O(\epsilon^{-4} \cdot n^2 \log n)$ with $O(\epsilon^{-4} \cdot n \log n)$ oracle accesses, due to the $O(1/\epsilon^2)$ iterations of Algorithm $A^*_l$, that makes $O(n \log n)$ oracle accesses, and to the fact that one evaluation of $B'(\cdot)$ requires $O(1/\epsilon^2)$ evaluations of $B(\cdot)$.
3 References

The results of this lecture are from [GL89]. Goldreich and Levin initially presented a different proof. They credit the proof with pairwise independence to Rackoff. Algorithm $A_7$ is due to Blum, Luby and Rubinfeld [BLR93]. The use of Algorithm GL-First-Attempt as a motivating example might be new to these notes. (Or, actually, to the Fall 2001 notes for this class.)

References


Notes for Lecture 12 v0.91

These notes are in a draft version. Please give me any comments you may have, because this will help me revise them.

Last Revision 11/30/04

1 Pseudorandom Generators

Definition 1 (Indistinguishability - finite definition) Two random variables $X$ and $Y$ taking value over $\{0,1\}^n$ are $(S,\epsilon)$-indistinguishable if for every circuit $C$ of size at most $S$ we have
\[
|\Pr[C(X) = 1] - \Pr[C(Y) = 1]| \leq \epsilon.
\]
We say that a random variable $X$ is $(S,\epsilon)$-pseudorandom if it is $(S,\epsilon)$-indistinguishable from the uniform distribution.

Definition 2 (Pseudorandom Generator) A function $G : \{0,1\}^* \rightarrow \{0,1\}^*$ is a pseudorandom generator of stretch $\ell(n)$, where $\ell : \mathbb{N} \rightarrow \mathbb{N}$ and $\ell(n) \geq n + 1$, if
\begin{itemize}
  \item $G$ is computable in polynomial time, in the length of the input;
  \item $G$ maps inputs of length $n$ into outputs of length $\ell(n)$;
  \item For every two polynomials $p()$ and $q()$ and for every sufficiently large $n$, the random variable $G(U_n)$ is $(p(n),1/q(n))$-pseudorandom, where $U_n$ denotes a random variable uniformly distributed in $\{0,1\}^n$.
\end{itemize}

Remark 1 The definition of pseudorandom generators is typically given in the following equivalent form. First, say that a function $\nu : \mathbb{N} \rightarrow \mathbb{R}$ is negligible if for every polynomial $p$ and for every sufficiently large $n$ we have $\nu(n) \leq 1/p(n)$. Then $G$ is said to be a pseudorandom generator if it satisfies to the first two properties of the above definition and if, in addition, for every family of polynomial size circuits $\{C_n\}$ there is a negligible function $\nu()$ such that
\[
|\Pr[C_{\ell(n)}(U_{\ell(n)}) = 1] - \Pr[C_{\ell(n)}(G(U_n)) = 1]| \leq \nu(n).
\]
One can easily verify that this definition implies Definition 2 (note that $\ell(n)$ must be upper bounded by a polynomial). For the other direction, let $G$ be a pseudorandom generator according to Definition 2 and fix a family of polynomial circuits. Define
\[
\nu(n) := |\Pr[C_{\ell(n)}(U_{\ell(n)}) = 1] - \Pr[C_{\ell(n)}(G(U_n)) = 1]|
\]
and note that for every polynomial $q()$ it must be that $\nu(n) \leq 1/q(\ell(n))$ for every sufficiently large $n$, and so it follows that $\nu()$ must be negligible.
One-way Functions and Hard-Core Bits

Definition 3 (One-Way Function – finite definition) A function $f : \{0, 1\}^n \rightarrow \{0, 1\}^n$ is $(S, \epsilon)$-one-way if for every circuit $A$ of size at most $S$,

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

For the next definition, we adopt the following convention: if $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ is a function, then we denote by $f_n : \{0, 1\}^n \rightarrow \{0, 1\}^*$ the restriction of $f$ to inputs of length $n$.

Definition 4 (One-Way Function - asymptotic definition) A family $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ is a one-way function if:

- $f$ is computable in polynomial time (in the length of the input) and
- for every two polynomials $p()$ and $q()$ and for every sufficiently large $n$, $f_n$ is $(p(n), 1/q(n))$-one-way.

In words, a one-way function is easy to compute but intractable to invert.

Theorem 1 If pseudorandom generators exist, then one-way functions exist.

Proof: See Exercises. □

The converse is also true, but it has an extremely difficulty proof.

Theorem 2 ([HILL99]) If one-way functions exist, then pseudorandom generators exist.

In these notes we will prove the simpler, but still remarkable, result that if one-way permutations exist then pseudorandom generators exist. A one-way permutation is a one-way function $f$ such that, for every $n$, $f_n : \{0, 1\}^n \rightarrow \{0, 1\}^n$ is a bijection. In general, we call a function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ a permutation if, for every $n$, $f_n : \{0, 1\}^n \rightarrow \{0, 1\}^n$ is a bijection.

We begin by introducing the notion of a hard-core predicate of a one-way permutation.

Definition 5 (Hard-Core Predicate – finite definition) A function $B : \{0, 1\}^n \rightarrow \{0, 1\}$ is a $(S, \epsilon)$ hard-core predicate for a permutation $f : \{0, 1\}^n \rightarrow \{0, 1\}^n$ if for every circuit $A$ of size at most $S$ we have

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

Definition 6 (Hard-Core Predicate – asymptotic definition) A function $B : \{0, 1\}^* \rightarrow \{0, 1\}$ is a hard-core predicate for a permutation $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ if

- $B$ is computable in polynomial time;
- for every two polynomials $p$ and $q$, and for every sufficiently large $n$, $B_n$ is $(p(n), 1/q(n))$ hard-core for $f_n$. 

2
In words, $B(x)$ is an efficiently computable property of $x$. Given $f(x)$, however, it is intractable to even guess with probability much better than $1/2$ whether $B(x)$ is zero or one.

For standard conjectured one-way permutations, such as RSA and exponentiation, very simple functions, such as the value of the last bit of the input, are hard-core predicates. The following result shows that every one-way permutation can be modified to have a hard-core predicate.

**Theorem 3 (Goldreich-Levin [GL89])** Let $f$ be a one-way permutation and define $f'$ such that $f'_{2n}(x, r) = f_n(x), r$. Define $B_{2n}(x, r) = x \cdot r$, where $x \cdot r = \sum_i x_i r_i \pmod{2}$. Then $f'$ is a one-way permutation and $B$ is a hard-core predicate for $f'$.

In words, the theorem says that if $f$ is a one-way permutation, and we pick at random $x \in \{0, 1\}^n$ and a subset $S \subseteq \{1, \ldots, n\}$, and we give to an adversary the value $f(x)$ and set $S$, it is intractable for the adversary to compute $\bigoplus_{i \in S} x_i$, or even to guess such value with probability much better than $1/2$. We defer the proof of the Goldreich-Levin Theorem to a later section.

### 3 One-way Permutations Imply Pseudorandom Generators

The main result of this section is the following.

**Theorem 4 (Blum-Micali-Yao [BM84, Yao82])** Suppose that one-way permutations exist, and let $\ell(n)$ be a polynomial. Then there are pseudorandom generators of stretch $\ell(n)$.

We will prove the Theorem in the finite setting, which gives important information about the security of concrete pseudorandom generators based on concrete finite permutations. We begin with the case of stretch $n + 1$.

**Lemma 5** Let $f : \{0, 1\}^n \to \{0, 1\}^n$ be a permutation and $B : \{0, 1\}^n \to \{0, 1\}$ be a $(S, \epsilon)$ hard-core predicate for $f$. Define

$$G(x) := f(x), B(x).$$

Then $G(U_n)$ is $(S - O(1), \epsilon)$-pseudorandom.

**Proof:** We prove that if $A$ is a circuit of size $S$ such that

$$|\Pr_{x \sim \{0, 1\}^n, r \sim \{0, 1\}}[A(f(x), r) = 1] - \Pr_{x \sim \{0, 1\}^n}[A(f(x), B(x)) = 1]| \geq \epsilon$$

then we can construct a circuit $C$ of size $S + O(1)$ such that

$$\Pr[C(f(x)) = B(x)] \geq \frac{1}{2} + \epsilon$$

An annoying technicality is that the new function is only defined for inputs of even length. One can get around this by saying that when $f''$ gets an input of odd length $2k + 1$ it discards the last input bit and then it computes $f'_{2k}$ of the first $2k$ input bits, as defined above.
and this clearly implies that the Lemma is true.

We start by noting that Equation (1) can be rewritten as

\[
\Pr_{x \sim \{0,1\}^n, r \sim \{0,1\}}[A'(f(x), r) = 1] - \Pr_{x \sim \{0,1\}^n}[A'(f(x), B(x)) = 1] \geq \epsilon
\]

where \( A' \) is either \( A \) or the complement of \( A \).

Equation (2) means that \( A'(f(x), b) \) is more likely to output 1 if \( b = B(x) \) than if \( b = \neg B(x) \). This suggests the following algorithm.

```
Input: y
    // the algorithm receives in input y = f(x) and tries to guess B(x) = B(f^{-1}(y))
begin
    pick random \( b \in \{0,1\} \)
    if \( A'(f(x), b) = 1 \) then return \( b \)
    else return \( \neg b \)
end
```

We will prove that, over the choices of \( x \) and \( b \), the algorithm, on input \( f(x) \) correctly computes \( B(x) \) with probability \( 1/2 + \epsilon \). Let us denote by \( C_b(y) \) the output of the algorithm given the input \( y \) and the random choice \( b \). That is, \( C_b(y) = (\neg b) \oplus A'(y, b) \).

\[
\Pr[C_b(f(x)) = B(x)] = \Pr[b = B(x)] \cdot \Pr[C_b(f(x)) = B(x) | b = B(x)]
+ \Pr[b \neq B(x)] \cdot \Pr[C_b(f(x)) = B(x) | b \neq B(x)]
= \frac{1}{2} \Pr[A'(f(x), B(x)) = 1] + \frac{1}{2} \Pr[A'(f(x), \neg B(x)) = 0]
= \frac{1}{2} + \frac{1}{2} \Pr[A'(f(x), B(x)) = 1] - \frac{1}{2} \Pr[A'(f(x), \neg B(x)) = 1]
\]

Let us now study the last expression. We can think of the probability of the event that \( A'(f(x), r) = 1 \) as the average of the probabilities that \( A'(f(x), B(x)) = 1 \) and \( A'(f(x), \neg B(x)) = 1 \). Equation (2) tells us that there is a difference of \( \epsilon \) between the probability of the event \( A'(f(x), B(x)) = 1 \) and the event \( A'(f(x), r) = 1 \). Then, it must follow that there is a difference of \( 2\epsilon \) between the probability of \( A'(f(x), B(x)) = 1 \) and of \( A'(f(x), \neg B(x)) = 1 \), so that the last expression in the above derivation is at least \( 1/2 + \epsilon \).

More formally:

\[
\frac{1}{2} \Pr[A'(f(x), B(x)) = 1] - \frac{1}{2} \Pr[A'(f(x), \neg B(x)) = 1]
= \Pr[A'(f(x), B(x)) = 1] - \left( \frac{1}{2} \Pr[A'(f(x), B(x)) = 1] + \frac{1}{2} \Pr[A'(f(x), \neg B(x)) = 1] \right)
= \Pr[A'(f(x), B(x)) = 1] - \Pr[A'(f(x), r) = 1]
\]

Combining everything together, we have

\[
\Pr[C_b(f(x)) = B(x)] = \frac{1}{2} + \Pr[A'(f(x), B(x)) = 1] - \Pr[A'(f(x), r) = 1] \geq \frac{1}{2} + \epsilon
\]
Finally, there exists a specific value \( b^* \in B \) such that
\[
\Pr_x[C_{b^*}(f(x))] = B(x) \geq \frac{1}{2} + \epsilon
\]
and we define \( C \) to be \( C_{b^*} \). Note that the size of \( C \) is \( S + O(1) \).

**Lemma 6** Let \( f : \{0,1\}^n \to \{0,1\}^n \) be a permutation, \( B : \{0,1\}^n \to \{0,1\} \) be a \((S, \epsilon)\) hard-core predicate for \( f \) and suppose that both \( B \) and \( f \) are computable by circuits of size at most \( t \). Define
\[
G(x) := B(x), B(f(x)), \ldots, B(f^{(k-1)}(x)), f^{(k)}(x)
\]
Then \( G(U_n) \) is \((S - O(tk), \epsilon/k)\)-pseudorandom.

**Proof:** We again proceed by contradiction. We assume that there is a circuit \( A \) of size \( S \) such that
\[
|\Pr[A(G(x)) = 1] - \Pr[A(r) = 1]| > \epsilon \tag{3}
\]
where \( x \) is uniform in \( \{0,1\}^n \) and \( r \) is uniform in \( \{0,1\}^{n+k} \), and we show there is a circuit \( C \) of size \( \leq S + O(tk) \) such that
\[
\Pr[C(f(x)) = B(x)] > \frac{1}{2} + \frac{\epsilon}{k}
\]
As a first step, we note that there is a circuit \( A' \) (which is either equal to \( A \) or to its complement) such that Expression (3) can be written as
\[
\Pr[A'(G(x)) = 1] - \Pr[A'(r) = 1] > \epsilon \tag{4}
\]
We now do a hybrid argument. We define \( k+1 \) distributions \( X_0, \ldots, X_L \). The distribution \( X_i \) is defined by computing \( g = G(x) \) for a random \( x \in \{0,1\}^n \) and picking \( r \in \{0,1\}^k \) at random; then the first \( i \) bits of \( r \) are concatenated with the last \( k-i \) bits of \( g \). We have by definition that \( X_0 \) is distributed like \( G(x) \) and \( X_k \) is uniform. (Indeed, since \( x \) is uniformly random and \( f \) is a permutation, then \( f(x) \) is uniformly distributed. By induction, \( f^{(i)}(x) \) is uniform for every \( i \).

We can rewrite Expression (4) as
\[
\Pr[A'(X_0) = 1] - \Pr[A'(X_k) = 1] > \epsilon
\]
and we note that we can write
\[
\epsilon < \sum_{j=0}^{k-1} \Pr[A'(X_j) = 1] - \Pr[A'(X_{j+1}) = 1]
\]
and so there exists one \( j \) for which
\[ \Pr[A'(X_{j-1}) = 1] - \Pr[A'(X_j) = 1] > \frac{\epsilon}{k} \]

which means that \( A' \) can distinguish

\[ X_{j-1} = b_1, \ldots, b_{j-1}, B(f^{j-1}(x)), \ldots, B(f^{k-1}(x)), f^k(x) \]

from

\[ X_j = b_1, \ldots, b_j, B(f^j(x)), \ldots, B(f^{k-1}(x)), f^k(x) \]

(where \( b_i \) are random bits)

Recall that, for every \( i \), the distribution \( f^i(x) \) is uniform in \( \{0, 1\}^n \). This means that the two distributions above can be equivalently redefined if we substitute \( f^i(x) \) with a uniformly random element \( y \). All this is giving us that \( C \) can distinguish

\[ b_1, \ldots, b_{j-1}, B(f^{j-1}(y)), B(y), \ldots, B(f^{k-j-1}(y)), f^{k-j}(y) \]

from

\[ b_1, \ldots, b_{j-1}, b_j, B(y), \ldots, B(f^{k-j-1}(y)), f^{k-j}(y) \]

On input \( y \) we can compute \( f(y), \ldots, f^{k-j}(y) \) and also \( B(y), \ldots, B(f^{k-j-1}(y)) \).

Consider now the following algorithm.

```
Input: y
// the algorithm receives in input y = f(z) and tries to guess B(z) = B(f^{i}(y))
begin
    pick random \( b_1, \ldots, b_j \in \{0, 1\} \)
    if \( A'(b_1, \ldots, b_j, B(y), \ldots, B(f^{k-j-1}(y)), f^{k-j}(y)) = 1 \) then return \( b_j \)
    else return \( \neg b_j \)
end
```

Denote by \( C_{b_1,\ldots,b_j}(y) \) the output of the algorithm given input \( y \) and random choices \( b_1, \ldots, b_j \). Then, as in the proof of a previous lemma, it is possible to show that

\[
\Pr[C_{b_1,\ldots,b_j}(f(x)) = B(x)] = \frac{1}{2} + \Pr[A'(X_{j-1}) = 1] - \Pr[A'(X_j) = 1] > \frac{1}{2} + \frac{\epsilon}{k}
\]

Finally, we observe that there must exist a fixed choice of \( b_1^*, \ldots, b_j^* \) such that

\[
\Pr[C_{b_1^*,\ldots,b_j^*}(f(x)) = B(x)] > \frac{1}{2} + \frac{\epsilon}{k}
\]

and we define \( C \) to be equal to \( C_{b_1^*,\ldots,b_j^*} \). \( \square \)
4 References

The notion of indistinguishability is due to Goldwasser and Micali [GM84], who also introduced the hybrid argument. Blum and Micali [BM84] were the first to give a formal definition of pseudorandom generator, but their definition was not based on indistinguishability. The indistinguishability-based definition is due to Yao [Yao82], who showed the equivalence of his definition to the definition of Blum and Micali. Yao also treated in greater generality the notion of hard-core predicate, that had been used in an ad hoc way by Goldwasser and Micali and by Blum and Micali.

References


Exercises

1. Let $\{X_n\}_{n \geq 1}$ and $\{Y_n\}_{n \geq 1}$ be ensembles (sets) of random variables, where $X_n$ and $Y_n$ take values over $\{0, 1\}^n$. Say that $\{X_n\}$ and $\{Y_n\}$ are indistinguishable if for every two polynomials $p$ and $q$ and for every large enough $n$ we have that $X_n$ and $Y_n$ are $(p(n), 1/q(n))$-indistinguishable.

   Prove that if $\{X_n\}$ and $\{Y_n\}$ are computationally indistinguishable, and $f$ is a length-preserving (meaning that the length of the output is always equal to the length of the input) polynomial time computable function, then $\{f_n(X_n)\}$ and $\{f_n(Y_n)\}$ are also computationally indistinguishable.

   [Hint: start by proving that if $f_n()$ is computable by a circuit of size $t$, and $X_n$ and $Y_n$ are $(S, \epsilon)$-indistinguishable, then $f_n(X_n)$ and $f_n(Y_n)$ are $(S - t, \epsilon)$-indistinguishable.]

2. Prove that there is an ensemble $\{X_n\}$ that is computationally indistinguishable from the ensemble of uniform distributions $\{U_n\}$, even though only $n \log n$ elements of $\{0, 1\}^n$ have non-zero probability in $X_n$.

   [Hint: use the probabilistic method and Chernoff bounds to argue that there exists a random variable $X_n$ that ranges over only $n \log n$ elements of $\{0, 1\}^n$ and that is $(n^{\Omega(\log n)}, 1/n^{\Omega(\log n)})$ pseudorandom.]

3. Prove that if pseudorandom generators of stretch $2n$ exist, then one-way functions exist.

   [Hint: prove that the generator itself is a one-way function.]

4. Prove that if a permutation $f$ has a hard-core predicate $B$, then $f$ is a one-way permutation.

5. Prove that if $P = NP$ then there cannot be any pseudorandom generators, even of stretch $n + 1$. 
Notes for Lecture 10

Counting Problems

1 Counting Classes

Definition 1. $R$ is an \textbf{NP}-relation, if there is a polynomial time algorithm $A$ such that $(x, y) \in R \iff A(x, y) = 1$ 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 2. #$P$ is the class of all problems of the form #$R$, where $R$ is an \textbf{NP}-relation.

Observe that an \textbf{NP}-relation $R$ naturally defines an \textbf{NP} language $L_R$, where $L_R = \{x : x \in R(y, x)\}$, and every \textbf{NP} language can be defined in this way. Therefore problems in #$P$ can always be seen as the problem of counting the number of witnesses for a given instance of an \textbf{NP} problem.

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

Definition 3. We say there is a parsimonious reduction from #$A$ to #$B$ (written #$A \leq_{\text{par}} #$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 4. #$A \leq #$B if there is a polynomial time algorithm for #$A$ given an oracle that solves #$B$.

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

Theorem 1. #$\text{CIRCUITSAT}$ is #$P$-complete under parsimonious reductions.

Proof: Let #$R$ be in #$P$ 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 $C(y) = \hat{C}(x, y)$. \hfill \Box

Theorem 2. #$\text{3SAT}$ 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 \( \varphi \) such that the number of satisfying assignments for \( \varphi \) 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 \( \varphi \) 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 \( \varphi \) with \( n + m \) variables and \( 4m \) clauses. So there is a parsimonious reduction from \#CIRCUITSAT to \#3SAT. \( \square \)

Notice that if a counting problem \( \# \mathcal{R} \) is \#P-complete under parsimonious reductions, then the associated language \( L_R \) is NP-complete, because \#3SAT \( \leq_{\text{par}} \# \mathcal{R} \) implies \( 3SAT \leq L_R \). On the other hand, with the less restrictive definition of reducibility, even some counting problems whose decision version is in P are \#P-complete. For example, the problem of counting the number of satisfying assignments for a given 2CNF formula and the problem of counting the number of perfect matchings in a given bipartite graphs are both \#P-complete.

2 Complexity of counting problems

We will prove the following theorem:

**Theorem 3** For every counting problem \#A in \#P, there is a probabilistic algorithm \( C \) that on input \( x \), computes 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 \( 1 \), using an oracle for NP.

The theorem says that \#P can be approximate in BPP\^{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 concerns the following result.

**Theorem 4 (Toda)** For every \( k \), \( \Sigma_k \subseteq \mathcal{P} \).

This implies that \#3SAT is \( \Sigma_k \)-hard for every \( k \), i.e., \#3SAT lies outside PH, unless the hierarchy collapses. Recall that BPP lies inside \( \Sigma_2 \), and hence approximating \#3SAT can be done in \( \Sigma_3 \). 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.

\(^1\)The above discussion was not very rigorous but it can be correctly formalized.
• It is enough to give a polynomial time $O(1)$-approximation for $\#3\text{SAT}$.

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

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

Given $\varphi$, we can construct $\varphi^k = \varphi_1 \land \varphi_2 \land \cdots \land \varphi_k$ where each $\varphi_i$ is a copy of $\varphi$ constructed using fresh variables. If $\varphi$ has $t$ satisfying assignments, $\varphi^k$ has $t^k$ satisfying assignments. Then, giving $\varphi^k$ to the algorithm we get

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

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

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

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 $\text{NP}$ questions, because the algorithm can guess these $k$ assignments and check them.

3 An approximate comparison procedure

Suppose that we had available an approximate comparison procedure $a\text{-}\text{comp}$ with the following properties:

- If $\#3\text{SAT}(\varphi) \geq 2^{k+1}$ then $a\text{-}\text{comp}(\varphi, k) = \text{YES}$ with high probability;
- If $\#3\text{SAT}(\varphi) < 2^k$ then $a\text{-}\text{comp}(\varphi, k) = \text{NO}$ with high probability.

Given $a\text{-}\text{comp}$, we can construct an algorithm that $2$-approximates $\#3\text{SAT}$ as described in Figure 1.

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

$$\frac{1}{2} a \leq \#3\text{SAT}(\varphi) < 2 \cdot a$$

and the algorithm outputs the correct answer with in a factor of $2$.

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

3
Input: $\varphi$

compute:
\[
\text{a-comp}(\varphi, 0)
\]
\[
\text{a-comp}(\varphi, 1)
\]
\[
\text{a-comp}(\varphi, 2)
\]
\[
\vdots
\]
\[
\text{a-comp}(\varphi, n + 1)
\]

if \text{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 $t = 1, \ldots, i - 1$ and NO for $t = i$
Output $2^i$

Figure 1: How to use \text{a-comp} to approximate \#3SAT.

4 Constructing \text{a-comp}

The procedure and its analysis is similar to the Valiant-Vazirani reduction: for a given formula $\varphi$ we pick a hash function $h$ from a pairwise independent family, and look at the number of assignments $x$ that satisfy $h$ and such that $h(x) = 0$.

In the Valiant-Vazirani reduction, we proved that if $S$ is a set of size approximately equal to the size of the range of $h()$, then, with constant probability, exactly one element of $S$ is mapped by $h()$ into 0. Now we use a different result, a simplified version of the “Leftover Hash Lemma” proved by Impagliazzo, Levin, and Luby in 1989, that says that if $S$ is sufficiently larger than the range of $h()$ then the number of elements of $S$ mapped into 0 is concentrated around its expectation.

Lemma 5 Let $H$ be a family of pairwise independent hash functions $h : \{0,1\}^n \rightarrow \{0,1\}^m$. Let $S \subseteq \{0,1\}^n$, $|S| \geq 4 \frac{2^m}{\epsilon^2}$. Then,

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

From this, \text{a-comp} can be constructed as in Figure 2.

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

- If $|S| \geq 2^{k+1}$, by Lemma 5 we have:

\[
\Pr_{h \in H} \left[ \left| \{a \in S : h(a) = 0\} \right| - \frac{|S|}{2^m} \right] \leq \frac{1}{4} \leq \frac{3}{4}.
\]

- If $|S| \leq 2^{k+1}$, by Lemma 5 we have:

\[
\Pr_{h \in H} \left[ \left| \{a \in S : h(a) = 0\} \right| - \frac{|S|}{2^m} \right] \leq \frac{1}{4} \leq \frac{3}{4}.
\]
input: \( \varphi, k \)

if \( k \leq 5 \) then check exactly whether \( \#3\text{SAT}(\varphi) \geq 2^k \).
if \( k \geq 6 \),
    pick \( h \) from a set of pairwise independent hash functions \( h : \{0,1\}^n \to \{0,1\}^m \),
    where \( m = k - 5 \)
    answer YES iff there are more then 48 assignments \( a \) to \( \varphi \) such that
    \( a \) satisfies \( \varphi \) and \( h(a) = 0 \).

Figure 2: The approximate algorithm for \( \#3\text{SAT} \).

(set \( \epsilon = \frac{1}{4} \), and \( |S| \geq \frac{4 \cdot 2^m}{\epsilon^2} = 64 \cdot 2^m \), because \( |S| \geq 2^{k+1} = 2^{m+6} \))

\[
\Pr_{h \in H} \left[ |\{a \in S : h(a) = 0\}| \leq \frac{3}{4} \cdot \frac{|S|}{2^m} \right] \leq \frac{1}{4},
\]

\[
\Pr_{h \in H} \left[ |\{a \in S : h(a) = 0\}| \geq 48 \right] \geq \frac{3}{4},
\]

which is the success probability of the algorithm.

- If \( |S| < 2^k \):
  Let \( S' \) be a superset of \( S \) of size \( 2^k \). We have

\[
\Pr_{h \in H}[\text{answer YES}] = \Pr_{h \in H}[\{a \in S : h(a) = 0\} \geq 48] \\
\leq \Pr_{h \in H}[\{a \in S' : h(s) = 0\} \geq 48] \\
\leq \Pr_{h \in H} \left[ \left| \{a \in S' : h(s) = 0\} \right| - \frac{|S'|}{2^m} \right] \geq \frac{|S'|}{2^m} \\
\leq \frac{1}{4}
\]

(by Lemma 5 with \( \epsilon = 1/2, |S'| = 32 \cdot 2^m \).)

Therefore, the algorithm will give the correct answer with probability at least 3/4, which can then be amplified to, say, 1 - 1/4n (so that all n invocations of \( a\)-\text{comp} are likely to be correct) by repeating the procedure \( O(\log n) \) times and taking the majority answer.

5 The proof of Lemma 5

We finish the lecture by proving Lemma 5.

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 \(i\), \(\Pr[X_i = 1] = \frac{1}{2^m}\) and \(\E[X_i] = \frac{1}{2^m}\).

Hence,
\[
\E\left[\sum_i X_i\right] = \frac{|S|}{2^m}.
\]

Also we calculate the variance
\[
\Var[X_i] = \E[X_i^2] - \E[X_i]^2 \\
\leq \E[X_i^2] \\
= \E[X_i] = \frac{1}{2^m}.
\]

Because \(X_1, \ldots, X_k\) are pairwise independent,
\[
\Var\left[\sum_i X_i\right] = \sum_i \Var[X_i] \leq \frac{|S|}{2^m}.
\]

Using Chebyshev’s Inequality, we get
\[
\Pr\left[\left|\{a \in S : h(a) = 0\}\right| - \frac{|S|}{2^m} \geq \epsilon \frac{|S|}{2^m}\right] = \Pr\left[\left|\sum_i X_i - \E[\sum_i X_i]\right| \geq \epsilon \E[\sum_i X_i]\right] \\
\leq \frac{\Var[\sum_i X_i]}{\epsilon^2 \E[\sum_i X_i]^2} \leq \frac{|S|}{2^m} \frac{\epsilon^2 |S|^2}{(2^m)^2} \\
= \frac{2^m}{\epsilon^2 |S|} \leq \frac{1}{4}.
\]

\[\square\]

6 Approximate Sampling

So far we have considered the following question: for an \(\textbf{NP}\)-relation \(R\), given an input \(x\), what is the size of the set \(R_x = \{y : (x, y) \in R\}\)? A related question is to be able to sample from the uniform distribution over \(R_x\).

Whenever the relation \(R\) is “downward self reducible” (a technical condition that we won’t define formally), it is possible to prove that there is a probabilistic algorithm running in time polynomial in \(|x|\) and \(1/\epsilon\) to approximate within \(1 + \epsilon\) the value \(|R_x|\) if and only if there is a probabilistic algorithm running in time polynomial in \(|x|\) and \(1/\epsilon\) that samples a distribution \(\epsilon\)-close to the uniform distribution over \(R_x\).

We show how the above result applies to 3SAT (the general result uses the same proof idea). For a formula \(\varphi\), a variable \(x\) and a bit \(b\), let us define by \(\varphi_{x \leftarrow b}\) the formula obtained by substituting the value \(b\) in place of \(x\).\(^2\)

\(^2\)Specifically, \(\varphi_{x \leftarrow 1}\) is obtained by removing each occurrence of \(\neg x\) from the clauses where it occurs, and removing all the clauses that contain an occurrence of \(x\); the formula \(\varphi_{x \leftarrow 0}\) is similarly obtained.
If \( \varphi \) is defined over variables \( x_1, \ldots, x_n \), it is easy to see that

\[
\#\varphi = \#\varphi_{x=0} + \#\varphi_{x=1}
\]

Also, if \( S \) is the uniform distribution over satisfying assignments for \( \varphi \), we note that

\[
\Pr_{(x_1, \ldots, x_n) \sim S}[x_1 = b] = \frac{\#\varphi_{x=b}}{\#\varphi}
\]

Suppose then that we have an efficient sampling algorithm that given \( \varphi \) and \( \epsilon \) generates a distribution \( \epsilon \)-close to uniform over the satisfying assignments of \( \varphi \).

Let us then run the sampling algorithm with approximation parameter \( \epsilon/2n \) and use it to sample about \( \tilde{O}(n^2/\epsilon^2) \) assignments. By computing the fraction of such assignments having \( x_1 = 0 \) and \( x_1 = 1 \), we get approximate values \( p_0, p_1 \), such that \( |p_b - \Pr_{(x_1, \ldots, x_n) \sim S}[x_1 = b]| \leq \epsilon/n \). Let \( b \) be such that \( p_b \geq 1/2 \), then \( \#\varphi_{x=b}/p_b \) is a good approximation, to within a multiplicative factor \( (1 + 2\epsilon/n) \) to \( \#\varphi \), and we can recurse to compute \( \#\varphi_{x=b} \) to within a \( (1 + 2\epsilon/n)^{n-1} \) factor.

Conversely, suppose we have an approximate counting procedure. Then we can approximately compute \( p_b = \frac{\#\varphi_{x=b}}{\#\varphi} \), generate a value \( b \) for \( x_1 \) with probability approximately \( p_b \), and then recurse to generate a random assignment for \( \#\varphi_{x=b} \).

The same equivalence holds, clearly, for 2SAT and, among other problems, for the problem of counting the number of perfect matchings in a bipartite graph. It is known that it is \( \text{NP} \)-hard to perform approximate counting for 2SAT and this result, with the above reduction, implies that approximate sampling is also hard for 2SAT. The problem of approximately sampling a perfect matching has a probabilistic polynomial solution, and the reduction implies that approximately counting the number of perfect matchings in a graph can also be done in probabilistic polynomial time.

The reduction and the results from last section also imply that 3SAT (and any other \( \text{NP} \) relation) has an approximate sampling algorithm that runs in probabilistic polynomial time with an \( \text{NP} \) oracle. With a careful use of the techniques from last week it is indeed possible to get an \textit{exact} sampling algorithm for 3SAT (and any other \( \text{NP} \) relation) running in probabilistic polynomial time with an \( \text{NP} \) oracle. This is essentially best possible, because the approximate sampling requires randomness by its very definition, and generating satisfying assignments for a 3SAT formula requires at least an \( \text{NP} \) oracle.

7 References

The class \( \#\text{P} \) was defined by Valiant [Val79]. An algorithm for approximate counting within the polynomial hierarchy was developed by Stockmeyer [Sto83]. The algorithm presented in these notes is taken from lecture notes by Oded Goldreich. The left-over hash lemma is from [HILL99]. The problem of approximate sampling and its relation to approximate counting is studied in [JVV86].
References


Notes for Lecture 8

In this lecture we will define the probabilistic complexity classes $\text{BPP}$, $\text{RP}$, $\text{ZPP}$ and we will see how they are related to each other, as well as to other deterministic or circuit complexity classes. Then we will present a randomized reduction showing that the SAT problem remains as hard when restricted to inputs that have either zero or one satisfying assignments as in the general case.

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 1** 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 $\Pr[A(x, r) = \text{correct answer for } x] \geq 2/3$. 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 $\text{BPP}$.

**Definition 2** A decision problem $L$ is in $\text{BPP}$ if there is a polynomial time algorithm $A$ and a polynomial $p()$ such that:

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

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

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 $\text{P}$ as:

**Definition 3** A decision problem $L$ is in $\text{P}$ if there is a polynomial time algorithm $A$ and a polynomial $p()$ such that:

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

$\forall x \notin L \quad \Pr_{r \in \{0,1\}^{p(|x|)}}[A(x, r) = 1] = 0$

Similarly, we define the classes $\text{RP}$ and $\text{ZPP}$.

**Definition 4** A decision problem $L$ is in $\text{RP}$ if there is a polynomial time algorithm $A$ and a polynomial $p()$ such that:

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

$\forall x \notin L \quad \Pr_{r \in \{0,1\}^{p(|x|)}}[A(x, r) = 1] \leq 0$
Definition 5 A decision problem \( L \) is in \( \text{ZPP} \) if there is a polynomial time algorithm \( A \) whose output can be 0, 1, ? and a polynomial \( p() \) such that:

\[
\forall x \Pr_{r \in \{0,1\}^{p(|x|)}}[A(x, r) = ?] \leq 1/2 \\
\forall x, \forall r \text{ such that } A(x, r) \neq ? \text{ then } A(x, r) = 1 \text{ if and only if } x \in L
\]

2 Relations between complexity classes

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

Theorem 1 \( \text{RP} \subseteq \text{NP} \).

Proof: Suppose we have a \( \text{RP} \) algorithm for a language \( L \). Then this algorithm is can be seen as a “verifier” showing that \( L \) is in \( \text{NP} \). If \( x \in L \) then there is a random sequence \( r \), for which the algorithm answers yes, and we think of such sequences \( r \) as witnesses that \( x \in L \). If \( x \notin L \) then there is no witness. □

We can also show that the class \( \text{ZPP} \) is no larger than \( \text{RP} \).

Theorem 2 \( \text{ZPP} \subseteq \text{RP} \).

Proof: We are going to convert a \( \text{ZPP} \) algorithm into an \( \text{RP} \) algorithm. The construction consists of running the \( \text{ZPP} \) 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 \( \text{ZPP} \) algorithm giving the output ? is less than 1/2. □

Another interesting property of the class \( \text{ZPP} \) is that it’s equivalent to the class of languages for which there is an average polynomial time algorithm that always gives the right answer. More formally,

Theorem 3 A language \( L \) is in the class \( \text{ZPP} \) 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 \( \text{ZPP} \) algorithm and if it outputs a ?, then we run it again. Suppose that the running time of the \( \text{ZPP} \) algorithm is \( T \), then the average running time of the new algorithm is:

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

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 \( \text{ZPP} \). We run the algorithm for time \( 2t(|x|) \) and output a ? if the algorithm has 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. □

Let us now prove the fact that $\text{RP}$ is contained in $\text{BPP}$.

**Theorem 4** $\text{RP} \subseteq \text{BPP}$

**Proof:** We will convert an $\text{RP}$ algorithm into a $\text{BPP}$ algorithm. In the case that the input $x$ does not belong to the language then the $\text{RP}$ algorithm always gives the right answer, so it certainly satisfies that $\text{BPP}$ requirement of giving the right answer with probability at least $2/3$. In the case that the input $x$ does belong to the language then we need to improve the probability of a correct answer from at least $1/2$ to at least $2/3$.

Let $A$ be an $\text{RP}$ algorithm for a decision problem $L$. We fix some number $k$ and define the following algorithm:

\[
A^{(k)}(x) = \begin{cases} 0 & \text{if } A(x, r_1) = A(x, r_2) = \ldots = A(x, r_k) = 0 \\ 1 & \text{else} \end{cases}
\]

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

\[
\begin{align*}
\Pr_{r_1, \ldots, r_k}[A^k(x, r_1, \ldots, r_k) = 1] &= 0 \\
\Pr_{r_1, \ldots, r_k}[A^k(x, r_1, \ldots, r_k) = 1] &\geq 1 - \left(\frac{1}{2}\right)^k
\end{align*}
\]

It is easy to see that by choosing an appropriate $k$ the second probability can go arbitrarily close to 1. In particular, choosing $k = 2$ suffices to have a probability larger than $2/3$, which is what is required by the definition of $\text{BPP}$. In fact, by choosing $k$ to be a polynomial in $|x|$, we can make the probability \textit{exponentially} close to 1. This means that the definition of $\text{RP}$ that we gave above would have been equivalent to a definition in which, instead of the bound of $1/2$ for the probability of a correct answer when the input is in the language $L$, we had have a bound of $1 - \left(\frac{1}{2}\right)^{q(|x|)}$, for a fixed polynomial $q$. □

Let, now, $A$ be a $\text{BPP}$ algorithm for a decision problem $L$. Then, we fix $k$ and define the following algorithm:
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 5 (Chernoff Bound)**

Suppose $X_1, \ldots, X_k$ are independent random variables with values in $\{0, 1\}$ and for every $i$, $\Pr[X_i = 1] = p$. Then:

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

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

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 $p = \Pr_r[A(x, r) = 1]$ 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.

$$
\Pr[A^{(k)} outputs the wrong answer on \ x] = \Pr\left[\frac{1}{k}\sum_{i=1}^{k} X_i < \frac{1}{2}\right]
$$

$$
\leq \Pr\left[\frac{1}{k}\sum_{i=1}^{k} X_i - p \leq -\frac{1}{6}\right]
$$

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

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

The probability is exponentially small in $k$. The same reasoning applies also for the case where $x \not\in L$. 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 6 (Adleman) 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 $x$

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

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 $n$.

**Claim 7** There is a random sequence $r \in \{0,1\}^{p(n)}$ such that for every $x \in \{0,1\}^n$ $A(x, r)$ 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.

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

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

$$\geq 1 - \sum_{x \in \{0,1\}^n} \Pr_r[A(x, r)\text{is wrong}]$$

$$\geq 1 - 2^n \cdot 2^{-(n+1)}$$

$$\geq \frac{1}{2}$$

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 \( \text{BPP} \subseteq \text{NP} \) or not is still an open question. What we know is that it's unlikely that \( \text{NP} \) is contained in \( \text{BPP} \), since then by the previous result \( \text{NP} \) would have polynomial size circuits and hence by the result of Karp and Lipton the polynomial hierarchy would collapse.

3 \( \text{BPP} \subseteq \Sigma_2 \)

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

**Lemma 8** If \( L \) is in \( \text{BPP} \) then there is an algorithm \( A \) such that for every \( x \),

\[
\Pr_r(A(x, r) = \text{right answer}) \geq 1 - \frac{1}{3^m},
\]

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 \( \text{BPP} \) algorithm for \( L \). Then for every \( x \), \( \Pr_r(\hat{A}(x, r) = \text{wrong answer}) \leq \frac{1}{3} \), 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 \( \frac{k(n)}{2} \) executions of \( \hat{A} \) accept. Call the new algorithm \( A \). Then \( A \) uses \( k(n)\hat{m}(n) \) random bits and \( \Pr_r(A(x, r) = \text{wrong answer}) \leq 2^{-ck(n)} \). We can then find \( k(n) \) with \( k(n) = \Theta(\log \hat{m}(n)) \) such that

\[
\frac{1}{2^{ck(n)}} \leq \frac{1}{3k(n)^{\hat{m}(n)}}. \quad \square
\]

**Theorem 9** \( \text{BPP} \subseteq \Sigma_2 \).

**Proof:** Let \( L \) be in \( \text{BPP} \) and \( A \) as in the claim. Then we want to show that

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

where \( m \) is the number of random bits used by \( A \) on input \( x \).

Suppose \( x \in L \). Then

\[
\Pr_{y_1, \ldots, y_m}(\exists z A(x, y_1 \oplus z) = \cdots = A(x, y_m \oplus z) = 0) \\
\leq \sum_{z \in \{0, 1\}^m} \Pr_{y_1, \ldots, y_m}(A(x, y_1 \oplus z) = \cdots = A(x, y_m \oplus z) = 0) \\
\leq 2^m \frac{1}{(3m)^m} \\
< 1.
\]

So

\[
\Pr_{y_1, \ldots, y_m}(\exists z \bigvee_i A(x, y_i \oplus z)) = 1 - \Pr_{y_1, \ldots, y_m}(\exists z A(x, y_1 \oplus z) = \cdots = A(x, y_m \oplus z) = 0) \\
> 0.
\]
So \((y_1, \ldots, y_m)\) exists. 
Conversely suppose \(x \notin L\). Then

\[
\Pr_z \left( \bigvee_i A(x, y_i + z) \right) \leq \sum_i \Pr_z (A(x, y_i + z) = 1)
\]

\[
\leq m \cdot \frac{1}{3m}
\]

\[
= \frac{1}{3}.
\]

So

\[
\Pr_z(A(x, y_1 + z) = \cdots = A(x, y_m + z) = 0) = \Pr_z \left( \bigvee_i A(x, y_i + z) \right)
\]

\[
\geq \frac{2}{3}
\]

\[
> 0.
\]

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

4 The Valiant-Vazirani Reduction

In this section we show the following: suppose there is an algorithm for the satisfiability 
problem that always find a satisfying assignment for formulae that have exactly one sat-
sifiable assignment (and behaves arbitrarily on other instances): then we can get an RP 
algorithm for the general satisfiability problem, and so \(\text{NP} = \text{RP}\).

We prove the result by presenting a randomized reduction that given in input a CNF 
formula \(\phi\) produces in output a polynomial number of formulae \(\psi_0, \ldots, \psi_n\). If \(\phi\) is satisfiable, 
then (with high probability) at least one of the \(\psi_i\) is satisfiable and has exactly one satisfying 
assignment; if \(\phi\) is not satisfiable, then (with probability one) all \(\psi_i\) are unsatisfiable.

The idea for the reduction is the following. Suppose \(\phi\) is a satisfiable formula with \(n\) 
variables that has about 2\(^k\) satisfying assignments, and let \(h : \{0, 1\}^n \rightarrow \{0, 1\}^k\) be a hash 
function picked from a family of pairwise independent hash functions: then the average 
number of assignments \(x\) such that \(\phi(x)\) is true and \(h(x) = (0, \ldots, 0)\) is about one. Indeed, 
we can prove formally that with constant probability there is exactly one such assignment,\(^1\) 
and that there is CNF formula \(\psi\) (easily constructed from \(\phi\) and \(h\)) that is satisfied precisely 
by that assignment. By doing the above construction for values of \(k\) ranging from 0 to \(n\), 
we obtain the desired reduction. Details follow.

Definition 6 Let \(H\) be a family of functions of the form \(h : \{0, 1\}^n \rightarrow \{0, 1\}^m\). We say 
that \(H\) is a family of pair-wise independent hash functions if for every two different inputs 
\(x, y \in \{0, 1\}^n\) and for every two possible outputs \(a, b \in \{0, 1\}^m\) we have

\[
\Pr_{h \in H}[h(x) = a \land h(y) = b] = \frac{1}{2^m}
\]

\(^1\)For technical reasons, it will be easier to prove that this is the case when picking a hash function 
\(h : \{0, 1\}^n \rightarrow \{0, 1\}^{k+2}\).
Another way to look at the definition is that for every \( x \neq y \), when we pick \( h \) at random then the random variables \( h(x) \) and \( h(y) \) are independent and uniformly distributed. In particular, for every \( x \neq y \) and for every \( a, b \) we have \( \Pr[h(x) = a | h(y) = b] = \Pr[h(x) = a] \).

For \( m \) vectors \( a_1, \ldots, a_m \in \{0,1\}^m \) and \( m \) bits \( b_1, \ldots, b_m \), define \( h_{a_1,\ldots,a_m,b_1,\ldots,b_m} : \{0,1\}^n \to \{0,1\}^m \) as \( h_{a,b}(x) = (a_1 \cdot x + b_1, \ldots, a_m \cdot x + b_m) \), and let \( H_{\text{AFF}} \) be the family of functions defined this way. Then it is not hard to see that \( H_{\text{AFF}} \) is a family of pairwise independent hash functions.

**Lemma 10** Let \( T \subseteq \{0,1\}^n \) be a set such that \( 2^k \leq |T| < 2^{k+1} \) and let \( H \) be a family of pairwise independent hash functions of the form \( h : \{0,1\}^n \to \{0,1\}^{k+2} \). Then if we pick \( h \) at random from \( H \), there is a constant probability that there is a unique element \( x \in T \) such that \( h(x) = 0 \). Precisely,

\[
\Pr_{h \in H}[|\{x \in T : h(x) = 0\}| = 1] \geq \frac{1}{8}
\]

**Proof:** Let us fix an element \( x \in T \). We want to compute the probability that \( x \) is the unique element of \( T \) mapped into 0 by \( h \). Clearly,

\[
\Pr[h(x) = 0 \land \forall y \in T - \{x\}. h(y) \neq 0] = \Pr[h(x) = 0]. \Pr[\forall y \in T - \{x\}. h(y) \neq 0|h(x) = 0]
\]

and we know that

\[
\Pr[h(x) = 0] = \frac{1}{2^{k+2}}
\]

The difficult part is to estimate the other probability. First, we write

\[
\Pr[\forall y \in T - \{x\}. h(y) \neq 0|h(x) = 0] = 1 - \Pr[h\exists y \in T - \{x\}. h(y) = 0|h(x) = 0]
\]

And then observe that

\[
\Pr[\exists y \in T - \{x\}. h(y) = 0|h(x) = 0] \leq \sum_{y \in |T| - \{x\}} \Pr[h(y) = 0|h(x) = 0] = \sum_{y \in |T| - \{x\}} \Pr[h(y) = 0] = \frac{|T| - 1}{2^{k+2}} \leq \frac{1}{2}
\]

Notice how we used the fact that the value of \( h(y) \) is independent of the value of \( h(x) \) when \( x \neq y \).

Putting everything together, we have

\[
\Pr[h(\forall y \in T - \{x\}. h(y) \neq 0|h(x) = 0] \geq \frac{1}{2}
\]
and so 
\[
\Pr_h[h(x) = 0 \land \forall y \in T - \{x\}.h(y) \neq 0] \geq \frac{1}{2k+3}
\]

To conclude the argument, we observe that the probability that there is a unique element of \( T \) mapped into 0 is given by the sum over \( x \in T \) of the probability that \( x \) is the unique element mapped into 0 (all these events are disjoint, so the probability of their union is the sum of the probabilities). The probability of a unique element mapped into 0 is then at least \(|T|/2k+3 > 1/8\). \( \square \)

**Lemma 11** There is a probabilistic polynomial time algorithm that on input a CNF formula \( \phi \) and an integer \( k \) outputs a formula \( \psi \) such that

- If \( \phi \) is unsatisfiable then \( \psi \) is unsatisfiable.
- If \( \phi \) has at least \( 2^k \) and less than \( 2^{k+1} \) satisfying assignments, then there is a probability at least \( 1/8 \) that \( \psi \) has exactly one satisfying assignment.

**Proof**: Say that \( \phi \) is a formula over \( n \) variables. The algorithm picks at random vectors \( a_1, \ldots, a_{k+2} \in \{0,1\}^n \) and bits \( b_1, \ldots, b_{k+2} \) and produces a formula \( \psi \) that is equivalent to the expression \( \phi(x) \land (a_1 \cdot x + b_1 = 0) \land \ldots \land (a_{k+2} \cdot x + b_{k+2} = 0) \). Indeed, there is no compact CNF expression to compute \( a \cdot x \) if \( a \) has a lot of ones, but we can proceed as follows: for each \( i \) we add auxiliary variables \( y^i_1, \ldots, y^i_n \) and then write a CNF condition equivalent to \((y^i_1 = x_1 \land a_i[1]) \land \ldots \land (y^i_n = y^i_{n-1} \oplus (x_n \land a_i[n] \oplus b_i))) \). Then \( \psi \) is the AND of the clauses in \( \phi \) plus all the above expressions for \( i = 1, 2, \ldots, k + 2 \).

By construction, the number of satisfying assignments of \( \psi \) is equal to the number of satisfying assignments \( x \) of \( \phi \) such that \( h_{a_1,\ldots,a_{k+2},b_1,\ldots,b_{k+2}}(x) = 0 \). If \( \phi \) is unsatisfiable, then, for every possible choice of the \( a_i \), \( \psi \) is also unsatisfiable.

If \( \phi \) has between \( 2^k \) and \( 2^{k+1} \) assignments, then Lemma 10 implies that with probability at least \( 1/8 \) there is exactly one satisfying assignment for \( \psi \). \( \square \)

**Theorem 12 (Valiant-Vazirani)** Suppose there is a polynomial time algorithm that on input a CNF formula having exactly one satisfying assignment finds that assignment. (We make no assumption on the behaviour of the algorithm on other inputs.) Then \( \text{NP} = \text{RP} \).

**Proof**: It is enough to show that, under the assumption of the Theorem, 3SAT has an \( \text{RP} \) algorithm.

On input a formula \( \phi \), we construct formulae \( \psi_0, \ldots, \psi_n \) by using the algorithm of Lemma 11 with parameters \( k = 0, \ldots, n \). We submit all formulae \( \psi_0, \ldots, \psi_n \) to the algorithm in the assumption of the Theorem, and accept if the algorithm can find a satisfying assignment for at least one of the formulae. If \( \phi \) is unsatisfiable, then all the formulae are always unsatisfiable, and so the algorithm has a probability zero of accepting. If \( \phi \) is satisfiable, then for some \( k \) it has between \( 2^k \) and \( 2^{k+1} \) satisfying assignments, and there is a probability at least \( 1/8 \) that \( \psi_k \) has exactly one satisfying assignment and that the algorithm accepts. If we repeat the above procedure \( t \) times, and accept if at least one iteration accepts, then if \( \phi \) is unsatisfiable we still have probability zero of accepting, otherwise we have probability at least \( 1 - (7/8)^t \) of accepting, which is more than \( 1/2 \) already for \( t = 6 \). \( \square \)
5 References

Probabilistic complexity classes were defined in [Gil77]. Adleman’s proof that $\text{BPP} \subseteq \text{SIZE}(n^{O(1)})$ appears in [Adl78]. Sipser’s proof that $\text{BPP} \subseteq \Sigma_2$ appears in [Sip83], and Lautemann’s proof is in [Lau83]. The Valiant-Vazirani result is from [VV86].

References


Exercises

1. Prove that \( \text{ZPP} = \text{RP} \cap \text{coRP} \).

2. Show that if \( \text{NP} \subseteq \text{BPP} \) then \( \text{NP} = \text{RP} \).

3. Prove that \( \text{SPACE}(O(n^{\log n})) \not\subseteq \text{BPP} \).

4. Change the assumption of Theorem 12 to having a \emph{probabilistic} polynomial time algorithm that on input a formula with exactly one satisfying assignment finds that assignment with probability at least \( 1/2 \). Prove that it still follows that \( \text{NP} = \text{RP} \).
Notes for Lecture 7

In this lecture we introduce the polynomial hierarchy and prove the Karp-Lipton-Sipser theorem.

1 Alternating Quantifiers

One way to look at the difference between \( \text{NP} \) and \( \text{coNP} \) is that a decision problem in \( \text{NP} \) is asking a of “does there exist” question, where the existence of the answer can by definition be efficiently proved. On the other hand, \( \text{coNP} \) asks “is it true for all” questions, which do not seem to have simple, efficient proofs.

Formally, a decision problem \( A \) is in \( \text{NP} \) if and only if there is a polynomial time procedure \( V(\cdot, \cdot) \) and a polynomial time bound \( p() \) such that

\[
x \in A \text{ if and only if } \exists y. |y| \leq p(|x|) \land V(x,y) = 1
\]

and a problem \( A \) is in \( \text{coNP} \) if and only if there is a polynomial time procedure \( V(\cdot, \cdot) \) and a polynomial bound \( p() \) such that

\[
x \in A \text{ if and only if } \forall y : |y| \leq p(|x|), V(x,y) = 1
\]

Now suppose you had a decision problem \( A \) defined in the following form:

\[
x \in A \iff \exists y_1 \text{ s.t. } |y_1| \leq p(|x|) \land \forall y_2 \text{ s.t. } |y_2| \leq p(|x|) V(x,y_1,y_2)
\]

(where \( p() \) is a polynomial time bound and \( V(\cdot, \cdot, \cdot) \) is a polynomial time procedure.)

In other words, an algorithm solving problem \( A \) should return \text{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 \( \phi \) over variables \( x_1, \ldots, x_n \), is there a formula \( \phi' \) which is equivalent to \( \phi \) and is of size at most \( k \)? In this case, \( y_1 \) is the formula \( \phi' \), \( y_2 \) 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 \( y_2 \), \( \phi \) and \( \phi' \) agree. Notice that \( \phi' \) is equivalent to \( \phi \) if and only if it agrees with \( \phi \) under all assignments of Boolean values to the variables.

As we will see, the problem \( A \) is a member of the class \( \Sigma_2 \) in the second level of the polynomial hierarchy.

2 The hierarchy

The polynomial hierarchy starts with familiar classes on level one: \( \Sigma_1 = \text{NP} \) and \( \Pi_1 = \text{coNP} \). For all \( i \geq 1 \), it includes two classes, \( \Sigma_i \) and \( \Pi_i \), which are defined as follows:

\[
A \in \Sigma_i \iff \exists y_1. \forall y_2. \ldots. Q y_i. V_A(x,y_1,\ldots,y_i)
\]
and
\[ B \in \Pi_i \iff \forall y_1. \exists y_2. \ldots. Q'y_i. V_B(x, y_1, \ldots, y_i) \]
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 \( \Sigma_{10} \) is \( \forall \), and the quantifier \( Q' \) for \( \Pi_{10} \) is \( \exists \)). For clarity, we have also omitted the conditions that each string \( y_i \) must be of polynomial length, but such conditions must be added for a completely formal definition of \( \Sigma_i \) and \( \Pi_i \).

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 \( \Pi_k \) 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.

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 \( M \) be a class of Turing machines. Then \( M^A \) is defined to be the class of machines obtained from \( M \) by allowing instances of \( A \) to be solved in one step. Similarly, if \( M \) is a class of Turing machines and \( C \) is a complexity class, then \( M^C = \bigcup_{A \in C} M^A \). If \( L \) is a complete problem for \( C \), and the machines in \( M \) are powerful enough to compute polynomial-time computations, then \( M^C = M^L \).

**Theorem 1** \( \Sigma_2 = \text{NP}^{3\text{SAT}} \).

**Proof:** Let \( A \in \Sigma_2 \), then for some polynomial \( p() \) and polynomial-time computable \( V() \) we have
\[ x \in A \text{ if and only if } \exists y_1 \text{ s.t. } |y_1| \leq p(|x|)\forall y_2 \text{ s.t. } |y_2| \leq p(|x|) \forall V(x, y_1, y_2) = 1 \]

Then we define a non-deterministic machine with an \( \text{NP} \)-oracle as follows: on input \( x \), the machine guesses a string \( y_1 \) of length at most \( p(|x|) \), and then asks the oracle whether \( \exists y_2. |y_2| \leq p(|x|) \forall V(x, y_1, y_2) = 0 \). The above question is an existential question about a polynomial-time computation, so, by Cook’s theorem, it is possible to construct in polynomial time a 3SAT instance that is satisfiable if and only if the answer to the above question is YES. The machine accepts if and only if the answer from the oracle is NO. It is immediate that the machine has an accepting computation if and only if
\[ \exists y_1. |y_1| \leq p(|x|) \forall V(x, y_1, y_2) = 1 \]

that is, the machine accepts if and only if \( x \in A \).

Notice that, in the above computation, only one oracle query is made, even though the definition of \( \text{NP}^{3\text{SAT}} \) allows us to make an arbitrary number of oracle queries.

Let now \( A \in \text{NP}^{3\text{SAT}} \), and let \( M \) be the oracle machine that solves \( A \). We first show that there is a machine \( M' \) that also solves \( A \), only makes one oracle query, and accepts
if and only if the answer to the oracle query is NO. On input $x$, $M'$ guesses an accepting computation of $M(x)$, that is, $M'$ guesses all the non-deterministic choices of $M(x)$, all the oracle questions, and all the answers. Then, for each question that was answered with a YES, $M'$ guesses a satisfying assignment to verify that the guess was correct. Finally, $M'$ is left with a certain set of oracle questions, say, the formulae $\phi_1, \ldots, \phi_k$, for which it has guessed that the correct oracle answer is NO. Then $M'$ asks its oracle whether (a formula equivalent to) $\phi_1 \lor \cdots \lor \phi_k$ is satisfiable, and it accepts if and only if the answer is NO.

Consider the computation of $M'(x)$ when $x \in A$: there is a valid accepting computation of $M(x)$, and $M'(x)$ can guess that computation along with the valid oracle answers; it can also guess valid assignments for all the queries for which the answer is YES; finally, it is left with unsatisfiable formulae $\phi_1, \ldots, \phi_k$, the answer to the single oracle query of $M'$ is NO, and $M'$ accepts.

Conversely, if $M'(x)$ has an accepting computation, then there must be a valid accepting computation of $M(x)$, and so $x \in A$. □

In fact, a more general result is known, whose proof works along similar lines.

**Theorem 2** For every $i \geq 2$, $\Sigma_i = NP^{\Sigma_{i-1}}$.

### 4 Additional Properties

Here are some more facts about the polynomial hierarchy, which we will not prove:

1. $\Pi_i$ and $\Sigma_i$ have complete problems for all $i$.
2. A $\Sigma_i$-complete problem is not in $\Pi_j$, $j \leq i - 1$, unless $\Pi_j = \Sigma_i$, and it is not in $\Sigma_j$ 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 $j \geq i$.

We will just prove the following special case of part (3).

**Theorem 3** Suppose $NP = coNP$. Then, for every $i \geq 2$, $\Sigma_i = NP$.

**Proof:** Let us first prove that, under the assumption of the theorem, $\Sigma_2 = NP$. Let $A \in \Sigma_2$ and let $M$ be the non-deterministic oracle machine that decides $A$ using oracle access to 3SAT. Let also $M'$ be the non-deterministic polynomial time Turing machine that decides the complement of the 3SAT problem. We now describe a non-deterministic polynomial time Turing machine $M''$ to decide $A$: on input $x$, $M''$ guesses an accepting computation of $M(x)$, along with oracle queries and answers; for each oracle question $\phi$ for which a YES answer has been guessed, $M''$ guesses a satisfying assignment; for each oracle question $\psi$ for which a NO answer has been guessed, $M''$ guesses an accepting computation of $M'(\psi)$. It is easy to verify that $M''(x)$ has an accepting computation if and only if $M^{3SAT}(x)$ has an accepting computation.
We can prove by induction on $i$ that $\Sigma_i = NP$. We have covered the base case. Let us now suppose that $\Sigma_{i-1} = NP$; then $\Sigma_i = NP^{\Sigma_{i-1}} = NP^{NP} = \Sigma_2 = NP$. □

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 The Karp-Lipton-Sipser Theorem

Theorem 4 (Karp-Lipton-Sipser) If $NP \subseteq SIZE(n^{O(1)})$ then $PH = \Sigma_2$. In other words, the polynomial hierarchy would collapse to its second level.

Before proving the above theorem, we first show a result that contains some of the ideas in the proof of the Karp-Lipton-Sipser theorem.

Lemma 5 If $NP \subseteq SIZE(n^{O(1)})$ then there is a family of polynomial-size circuits that on input a 3CNF formula $\phi$ outputs a satisfying assignment for $\phi$ if one such assignment exists and a sequence of zeroes otherwise.

Proof: We define the circuits $C^1_n, \ldots, C^n_n$ as follows:

- $C^1_n$, on input a formula $\phi$ over $n$ variables outputs 1 if and only if there is a satisfying assignment for $\phi$ where $x_1 = 1$,

- $\ldots$

- $C^i_n$, on input a formula $\phi$ over $n$ variables and bits $b_1, \ldots, b_{i-1}$, outputs 1 if and only if there is a satisfying assignment for $\phi$ where $x_1 = b_1, \ldots, x_{i-1} = b_{i-1}, x_i = 1$

- $\ldots$

- $C^n_n$, on input a formula $\phi$ over $n$ variables and bits $b_1, \ldots, b_{n-1}$, outputs 1 if and only if $\phi$ is satisfied by the assignment $x_1 = b_1, \ldots, x_{n-1} = b_{n-1}, x_n = 1$.

Also, each circuit realizes an NP computation, and so it can be built of polynomial size. Consider now the sequence $b_1 = C^1_n(\phi), b_2 = C^2_n(b_1, \phi), \ldots, b_nC^n_n(b_1, \ldots, b_{n-1}, \phi)$. The reader should be able to convince himself that this is a satisfying assignment for $\phi$ if $\phi$ is satisfiable, and a sequence of zeroes otherwise. □

We now prove the Karp-Lipton-Sipser theorem.

Proof: [Of Theorem 4] We will show that if $NP \subseteq SIZE(n^{O(1)})$ then $\Pi_2 \subseteq \Sigma_2$. By a result in a previous lecture, this implies that $PH = \Sigma_2$.

Let $L \in \Pi_2$, then there is a polynomial $p()$ and a polynomial-time computable $V()$ such that

$$x \in L \iff \forall y_1. |y_1| \leq p(|x|) \exists y_2. |y_2| \leq p(|x|). V(x, y_1, y_2) = 1$$

By adapting the proof of Lemma 5 (see Figure 1), or by using the statement of the Lemma and Cook’s theorem, we can show that, for every $n$, there is a circuit $C_n$ of size polynomial in $n$ such that for every $x$ and every $y_1$, $|y_1| \leq p(|x|)$,
There is $y_2$ starting with 1. 0 or 1

Figure 1: How to use decision problem solvers to find a witness to a search problem.

$$\exists y_2. |y_2| \leq p(|x|) \land V(x, y_1, y_2) = 1 \text{ iff } V(x, y_1, C_n(x, y_1)) = 1$$

Let $q(n)$ be a polynomial upper bound to the size of $C_n$.

So now we have that for inputs $x$ of length $n$,

$$x \in L \text{ iff } \exists C_n, |C_n| \leq q(n). \forall y_1. |y_1| \leq p(n). V(x, y_1, C_n(x, y_1)) = 1$$

which shows that $L$ is in $\Sigma_2$. $\square$

6 References

The polynomial time hierarchy was defined by Stockmeyer [Sto76]. Wrathall [Wra76] shows that every class in the polynomial hierarchy has complete problems.

The Karp-Lipton-Sipser theorem appears in [KL80].

References


Exercises

1. In the MAX SAT problem we are given a formula $\phi$ in conjunctive normal form and we want to find the assignment of values to the variables that maximizes the number of satisfied clauses. (For example, if $\phi$ is satisfiable, the optimal solution satisfies all the clauses and the MAX SAT problem reduces to finding a satisfying assignment.) Consider the following decision problem: given a formula $\phi$ in conjunctive normal form and an integer $k$, determine if $k$ is the number of clauses of $\phi$ satisfied by an optimal assignment.

   • Prove that this problem is in NP if and only if NP = coNP.  
     [Hint: prove that it is both NP-hard and coNP-hard.]
   • Prove that this problem is in $\Sigma_2$.

2. Define $\text{EXP} = \text{DTIME}(2^{n^{O(1)}})$. Prove that if $\text{EXP} \subseteq \text{SIZE}(n^{O(1)})$ then $\text{EXP} = \Sigma_2$. 


Notes for Lecture 6

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.

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 1.

Figure 1: A Boolean circuit.

A circuit $C$ with $n$ inputs and $m$ outputs computes a function $f_C : \{0,1\}^n \to \{0,1\}^m$. See Figure 2 for an example.

Define $\text{SIZE}(C) = \# \text{ of AND and OR gates of } C$. 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 1 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. $|x| = n$,

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

Definition 2 Say $L \in \text{SIZE}(s(n))$ 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 2: A circuit computing the boolean function $f_C(x_1 x_2 x_3 x_4) = x_1 \oplus x_2 \oplus x_3 \oplus x_4$.

2 Relation to other complexity classes

Unlike other complexity measures, like time and space, for which there are languages of arbitrarily high complexity, the size complexity of a problem is always at most exponential.

**Theorem 1** For every language $L$, $L \in \text{SIZE}(O(2^n))$.

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

Use the identity $f(x_1 x_2 \ldots x_n) = (x_1 \land f(1 x_2 \ldots x_n)) \lor (\overline{x_1} \land f(0 x_2 \ldots x_n))$ to recursively construct a circuit for $f$, as shown in Figure 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 $s(n) = 2 \cdot 2^n - 3 = O(2^n)$. □

The exponential bound is nearly tight.

**Theorem 2** There are languages $L$ such that $L \notin \text{SIZE}(o(2^n/n))$. In particular, for every $n \geq 11$, there exists $f : \{0,1\}^n \rightarrow \{0,1\}$ that cannot be computed by a circuit of size $2^n/4n$.

**Proof:** This is a counting argument. There are $2^{2^n}$ functions $f : \{0,1\}^n \rightarrow \{0,1\}$, 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

$$s \times (2 \log(n + s) + 3) \leq s \cdot (2 \log 2s + 3) = s \cdot (2 \log 2s + 5).$$
Figure 3: A circuit computing any function $f(x_1 x_2 \ldots x_n)$ of $n$ variables assuming circuits for two functions of $n-1$ variables.

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

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

This is satisfied if $s \leq \frac{2n}{4n}$ and $n \geq 11$. □

The following result shows that efficient computations can be simulated by small circuits.

**Theorem 3** If $L \in \text{DTIME}(t(n))$, then $L \in \text{SIZE}(O(t^2(n)))$.

**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 4.

Assume that each entry $(a, q)$ of the tableau is encoded using $k$ bits. By Proposition 1, the transition function $\{0,1\}^{3k} \rightarrow \{0,1\}^k$ used by the machine can be implemented by a “next state circuit” of size $k \cdot O(2^{3k})$, 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 5. □

**Corollary 4** $\mathbf{P} \subseteq \text{SIZE}(n^{O(1)})$.

On the other hand, it’s easy to show that $\mathbf{P} \neq \text{SIZE}(n^{O(1)})$, and, in fact, one can define languages in $\text{SIZE}(O(1))$ that are undecidable.
Figure 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 5: Circuit to simulate a Turing machine computation by constructing the tableau.
Exercises

1. Show that $\text{SIZE}(n^{O(1)}) \not\subseteq \text{P}$.

2. Show that there is a language in $\text{SPACE}(2^{o(n)})$ that does not belong to $\text{SIZE}(2^{o(n)})$. 
1 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 $L$ the set of decision problems solvable in $O(\log n)$ space. We denote by $PSPACE$ the set of decision problems solvable in polynomial space. A first observation is that a space-efficient machine is, to a certain extent, also a time-efficient one. In general we denote by $SPACE(s(n))$ the set of decision problems that can be solved using space at most $s(n)$ on inputs of length $n$.

**Theorem 1** 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. $\square$

**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. In general we denote by $NSPACE(s(n))$ the set of decision problems that can be solved by non-deterministic machines that use at most $s(n)$ bits of space on inputs of length $n$.

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 **NL** is contained in **P**. An intuitive reason
why not all problems with a log-space “verifier” can be simulated in \( \text{NL} \) is that an \( \text{NL} \) machine does not have enough memory to keep track of all the non-deterministic choices that it makes.

**Theorem 2** \( \text{NL} \subseteq \text{P} \).

**Proof:** Let \( L \) be a language in \( \text{NL} \) and let \( M \) be a non-deterministic log-space machine for \( L \). Consider a computation of \( M(x) \). As before, 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 \( \delta \)). 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. □

2 Reductions in NL

We would like to introduce a notion of completeness in \( \text{NL} \) analogous to the notion of completeness that we know for the class \( \text{NP} \). A first observation is that, in order to have a meaningful notion of completeness in \( \text{NL} \), we cannot use polynomial-time reductions; otherwise any \( \text{NL} \) problem having at least a YES instance and at least a NO instance would be trivially \( \text{NL} \)-complete. To get a more interesting notion of \( \text{NL} \)-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 \( B \), \( A \leq_{\text{log}} B \), if \( \exists \) a function \( f \) computable in log space such that \( x \in A \iff f(x) \in B \), and \( B \in L \).

**Theorem 3** If \( B \in L \), and \( A \leq_{\text{log}} B \), then \( A \in L \).

**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 give 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 \( f(x) \); 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 4** if \( A \leq_{\log} B, B \leq_{\log} C \), then \( A \leq_{\log} C \).

### 3 NL Completeness

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

**Definition 2** A decision problem \( A \) is NL-hard if for every \( B \in \text{NL} \), \( B \leq_{\log} A \). A decision problem \( A \) is NL-complete if \( A \in \text{NL} \) 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 5** STCONN is NL-complete.

**Proof:**

1. **STCONN \( \in \text{NL} \).**
   
   On input \( G(V,E), s,t \), set \( p \) to \( s \). For \( i = 1 \) to \( |V| \), nondeterministically, 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 \( A \in \text{NL} \), 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.

\[\square\]

### 4 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 6 STCONN 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 $\text{REACH}(u,v,k)$ that accepts and halts if $v$ can be reached from $u$ in $\leq k$ steps. If $k = 1$, then $\text{REACH}$ accepts if $(u,v)$ is an edge. If $k \geq 2$, $\forall w \in V - \{u,v\}$, compute $\text{REACH}(u,w,\lfloor k/2 \rfloor)$ and $\text{REACH}(w,v,\lceil k/2 \rceil)$. If both accept and halt, accept. Else, reject.

Let $S(k)$ be the worst-case space use of $\text{REACH}(\cdot,\cdot,k)$. 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 $\text{REACH}$ and for tracking $w$. So, $S(k) = O(\log k \cdot \log n)$. Since $k \leq n$, the worst-case space use of $\text{REACH}$ is $O(\log^2 n)$. □

Essentially the same proof applies to arbitrary non-deterministic space-bounded computations. This result was proved in [Sav70]

Theorem 7 (Savitch’s Theorem) For every function $s(n)$ computable in space $O(s(n))$, $\text{NSPACE}(s) = \text{SPACE}(O(s^2))$

**Proof:** We begin with a nondeterministic machine $M$, which on input $x$ uses $s(|x|)$ space. We define $\text{REACH}(c_i,c_j,k)$, as in the proof of Theorem 6, which accepts and halts if $M(x)$ can go from $c_i$ to $c_j$ in $\leq k$ steps. We compute $\text{REACH}(c_0, c_{\text{acc}}, 2O(s(|x|)))$ for all accepting configurations $c_{\text{acc}}$. If there is a call of $\text{REACH}$ which accepts and halts, then $M$ accepts. Else, $M$ rejects. If $\text{REACH}$ accepts and halts, it will do so in $\leq 2^{O(|x|)}$ steps.

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

Comparing Theorem 6 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 $\text{REACH}(\cdot,\cdot,k)$, we see $t(1) = O(n + m)$, and $t(k) = n(2 \cdot T(k/2))$, 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, although such an algorithm is known for undirected connectivity.

5 Undirected Connectivity

In the undirected $s - t$ connectivity problem (abbreviated ST-UCONN) we are given an undirected graph $G = (V,E)$ and two vertices $s,t \in V$, and the question is whether that is a path between $s$ and $t$ in $G$.

While this problem is not known to be complete for NL, and it probably is not, ST-UCONN is complete for the class SL of decision problems that are solvable by symmetric non-deterministic machines that use $O(\log n)$ space. A non-deterministic machine is symmetric if whenever it can make a transition from a global state $s$ to a global state $s'$ then the transition from $s'$ to $s$ is also possible. The proof of SL-completeness of ST-UCONN is identical to the proof of NL-completeness of ST-CONN except for the additional observation that the transition graph of a symmetric machine is undirected.
For ST-UCONN 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].

6 Randomized Log-space

We now wish to introduce randomized space-bounded Turing machine. For simplicity, we will only introduce randomized machines for solving decision problems. In addition to a read-only input tape and a read/write work tape, such machines also have a read-only random tape to which they have one-way access, meaning that the head on that tape can only move, say, left-to-right. For every fixed input and fixed content of the random tape, the machine is completely deterministic, and either accepts or rejects. For a Turing machine $M$, an input $x$ and a content $r$ of the random tape, we denote by $M(r, x)$ the outcome of the computation.

We say that a decision problem $L$ belongs to the class $RL$ (for randomized log-space) if there is a probabilistic Turing machine $M$ that uses $O(\log n)$ space on inputs of length $n$ and such that

- For every content of the random tape and for every input $x$, $M$ halts.
- For every $x \in L$, $\Pr_r[M(r, x) \text{ accepts }] \geq 1/2$
- For every $x \notin L$, $\Pr_r[M(r, x) \text{ accepts }] = 0$.

Notice that the first property implies that $M$ always runs in polynomial time. It is easy to observe that any constant bigger than 0 and smaller than 1 could be equivalently used instead of $1/2$ in the definition above. It also follows from the definition that $L \subseteq RL \subseteq NL$.

The following result shows that, indeed, $L \subseteq SL \subseteq RL \subseteq NL$.

**Theorem 8** The problem ST-UCONN is in $RL$.

We will not give a proof of the above theorem, but just describe the algorithm. Given an undirected graph $G = (V, E)$ and two vertices $s, t$, the algorithm performs a random walk of length $100 \cdot n^3$ starting from $s$. If $t$ is never reached, the algorithm rejects.

input: $G = (V, E), s, t$

$v \leftarrow s$

for $i \leftarrow 1$ to $100 \cdot n^3$

- pick at random a neighbor $w$ of $v$
  - if $w = t$ then halt and accept
  - $v \leftarrow w$ reject

The analysis of the algorithm is based on the fact that if we start a random walk from a vertex $s$ of an undirected vertex $G$, then all vertices in the connected component of $s$ are likely to be visited at least once after $\Theta(n^3)$ steps.
The best known deterministic simulation of RL uses $O((\log n)^{3/2})$ space, and is due to Saks and Zhou [SZ95].

7 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 \textsc{STCONN}. \textsc{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 \textsc{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 \textsc{STCONN}, and since \textsc{STCONN} has been shown to be in NL (by the existence of our machine), so is $A$. Also, NL $\subseteq$ coNL, since if \textsc{STCONN} $\in$ NL, by definition \textsc{STCONN} $\in$ coNL, and since \textsc{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. This result was proved independently in [Imm88] and [Sze88].

7.1 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 \textsc{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 $s$
    if guess = YES then
        non-deterministically guess the distance $k$ from $s$ to $v$
        $p ← s$
        for $i ← 1$ to $k$ do
            non-deterministically pick a neighbor $q$ of $p$
            $p ← q$
            if $p \neq v$, reject
        $c ← c + 1$
    if $c = r$ then return YES, otherwise return NO
```
It is easy to verify that this algorithm is indeed in NL. The algorithm only needs to maintain the five variables $c, k, p, q, v$, and each of these variables can be represented with $\log |V|$ bits.

Regarding correctness, notice that, in the algorithm, $c$ 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 from $s$. Hence the algorithm accepts if and only if it discovers that $t$ is not reachable from $s$.

### 7.2 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) |
| $c \gets 0$ |
| for all $v \in V$ do |
|     non-deterministically guess if $v$ is reachable from $s$ in $k$ steps |
|     if guess = YES then |
|     | $p \gets s$ |
|     | for $i \gets 1$ to $k$ do |
|     | | non-deterministically guess a neighbor $q$ of $p$ (possibly not moving at all) |
|     | | $p \gets q$ |
|     | if $p \neq v$ reject |
|     | $c \gets c + 1$ |
| 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 $r_k$ to be the set of vertices reachable from $s$ in at most $k$ steps. Then $r = r_{n-1}$, where $n$ is the number of vertices in $G$. The idea is to try to compute $r_k$ from $r_{k-1}$ and repeat the procedure $n - 1$ times, starting from $r_0 = 1$. In Figure 7.2 is another try at an algorithm.

Here is the idea behind the algorithm: for each vertex $v$, we need to determine if it is reachable from $s$ in at most $k$ steps. To do this, we can loop over all vertices which are a distance at most $k - 1$ 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 $k$ steps). The algorithm is able to force all vertices of distance at most $k - 1$ to be considered because it is given $r_{k-1}$ as an input.

Now, putting this algorithm together with the first one listed above, we have shown that STCONN $\in$ NL, implying that NL $= \text{coNL}$. 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 $O(s(n))$. 

7
input: \( G = (V, E), s, k, r_{k-1} \)
output: the number of vertices reachable from \( s \) in at most \( k \) steps (including \( s \) in this count)
assumption: \( r_{k-1} \) is the exact number of vertices reachable from \( s \) in at most \( k - 1 \) steps

\[
c \leftarrow 0
\]
for all \( v \in V \) do
  \( d \leftarrow 0 \)
  flag \leftarrow FALSE
  for all \( w \in V \) do
    \( p \leftarrow s \)
    for \( i \leftarrow 1 \) to \( k - 1 \) do
      non-deterministically pick a neighbor \( q \) of \( p \) (possibly not moving at all)
      \( p \leftarrow q \)
    if \( p = w \) then
      \( d \leftarrow d + 1 \)
      if \( v \) is a neighbor of \( w \), or if \( v = w \) then
        flag \leftarrow TRUE
    if \( d < r_{k-1} \) reject
  if flag then \( c \leftarrow c + 1 \)
return \( c \)

Figure 1: The correct algorithm that proves \( \text{NL} = \text{coNL} \).

References


Exercises

1. Define the class $\text{BPL}$ (for bounded-error probabilistic log-space) as follows. A decision problem $L$ is in $\text{BPL}$ if there is a log-space probabilistic Turing machine $M$ such that

- For every $r$ and every $x$, $M(r, x)$ halts;
- If $x \in L$ then $\Pr_r[M(r, x) \text{ accepts }] \geq 2/3$;
- If $x \notin L$ then $\Pr_r[M(r, x) \text{ accepts }] \leq 1/3$.

Then

(a) Prove that $\text{RL} \subseteq \text{BPL}$.
(b) Prove that $\text{BPL} \subseteq \text{SPACE}(O((\log n)^2))$.
(c) This last question requires a somewhat different approach: prove that $\text{BPL} \subseteq \text{P}$.
Notes for Lecture 1

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. We will occasionally refer to Turing machines, especially in this lecture.

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. We conclude by demonstrating the use of diagonalization to show some separations between complexity classes. It is unlikely that such techniques will help solving the $P$ versus $NP$ problem.

1 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 as 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. For the moment, we will discuss decision and search problem.

---

1This distinction is useful and natural, but it is also arbitrary: in fact every problem can be seen as a search problem.
In a decision problem, given an input \( x \in \{0,1\}^\ast \), 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 \( L \subseteq \{0,1\}^\ast \) of inputs for which the answer is YES. A subset of \( \{0,1\}^\ast \) 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 \( \{0,1\}^\ast \) 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 \( x \in \{0,1\}^\ast \) we want to compute some answer \( y \in \{0,1\}^\ast \) that is in some relation to \( x \), if such a \( y \) exists. Thus, a search problem is specified by a relation \( R \subseteq \{0,1\}^\ast \times \{0,1\}^\ast \), 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 \( c : V \rightarrow \{1,2,3\} \) 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 \( R_{3\text{COL}} \) that contains all the pairs \( (G,c) \) where \( G \) is a 3-colorable graph and \( c \) is a valid 3-coloring of \( G \).

## 2 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 \( n \).

We denote by \( \text{P} \) the class of decision problems that are solvable in polynomial time.

We say that a search problem defined by a relation \( R \) is a \( \text{NP} \) search problem if the relation is efficiently computable and such that solutions, if they exist, are short. Formally, \( R \) is an \( \text{NP} \) 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 \( |y| \leq p(|x|) \).

We say that a decision problem \( L \) is an \( \text{NP} \) decision problem if there is some \( \text{NP} \) 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 \( \text{NP} \) 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 \( y, |y| \leq p(|x|) \) such that \( V(x, y) \) accepts.

We denote by \( \text{NP} \) the class of \( \text{NP} \) decision problems.

Equivalently, \( \text{NP} \) can be defined as the set of decision problems that are solvable in polynomial time by a non-deterministic Turing machine. Suppose that \( L \) is solvable in polynomial time by a non-deterministic Turing machine \( M \): then we can define the relation \( R \) such that \( (x, t) \in R \) if and only if \( t \) is a transcript of an accepting computation of \( M \) on input \( x \) and it’s easy to prove that \( R \) is an \( \text{NP} \) relation and that \( L \) is in \( \text{NP} \) according to our first definition. Suppose that \( L \) is in \( \text{NP} \) according to our first definition and that \( R \) is the corresponding \( \text{NP} \) relation. Then, on input \( x \), a non-deterministic Turing machine can guess a string \( y \) of length less than \( p(|x|) \) and then accept if and only if \( (x, y) \in R \). Such a machine can be implemented to run in non-deterministic polynomial time and it decides \( L \).

For a function \( t : \mathbb{N} \rightarrow \mathbb{N} \), we define by \( \text{DTIME}(t(n)) \) the set of decision problems that are solvable by a deterministic Turing machine within time \( t(n) \) on inputs of length \( n \), and by \( \text{NTIME}(t(n)) \) the set of decision problems that are solvable by a non-deterministic Turing machine within time \( t(n) \) on inputs of length \( n \). Therefore, \( \text{P} = \bigcup_{k} \text{DTIME}(O(n^k)) \) and \( \text{NP} = \bigcup_{k} \text{DTIME}(O(n^k)) \).

3 \ NP-completeness

3.1 Reductions

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

Two immediate observations: if \( A \leq B \) and \( B \) is in \( \text{P} \), then also \( A \in \text{P} \) (conversely, if \( A \leq B \), and \( A \not\in \text{P} \) then also \( B \not\in \text{P} \)); if \( A \leq B \) and \( B \leq C \), then also \( A \leq C \).

3.2 NP-completeness

A decision problem \( A \) is \( \text{NP} \)-hard if for every problem \( L \in \text{NP} \) we have \( L \leq A \). A decision problem \( A \) is \( \text{NP} \)-complete if it is \( \text{NP} \)-hard and it belongs to \( \text{NP} \).

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

3.3 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 \( y \in \{0, 1\}^*, |y| \leq l \), such that \( M(x, y) \) accepts in \( \leq t \) steps.

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

\(^2\)The “unary” encoding of an integer \( n \) is a sequence of \( n \) ones.
Let $L$ be an $\text{NP}$ 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 $y$, $|y| \leq p_L(|x|)$ 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 $f(x) = (V_L(x), x, T_L(|x|) + p_L(|x|))$. Just by applying the definitions, we can see that $x \in L$ if and only $f(x) \in U$.

3.4 The Problem SAT

In SAT (that stands for $\text{CNF}$-satisfiability) we are given Boolean variables $x_1, x_2, \ldots, x_n$ and a Boolean formula $\phi$ 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 assign 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 $\lor$ (that stands for OR), $\land$ (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; $x \land y$ is TRUE if and only if both $x$ and $y$ are TRUE; $x \lor y$ is TRUE if and only at least one of $x$ or $y$ is TRUE.

So, for example, the expression $\neg x \land (x \lor y)$ can be satisfied by setting $x$ to FALSE and $y$ to TRUE, while the expression $x \land (\neg x \lor y) \land \neg y$ 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 $(x_2 \lor \neg x_4 \lor x_5)$ is a clause, and so is $(x_3)$. A formula in conjunctive normal form is the AND of clauses. For example

$$(x_3 \lor \neg x_4) \land (x_1) \land (\neg x_3 \lor x_2)$$

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

$$x \land (\neg x \lor y) \land \neg y$$

is not satisfiable, as it has already been observed.

**Theorem 1 (Cook)** SAT is NP-complete.

We will give a proof of Cook’s Theorem later in the course.

4 Diagonalization

Diagonalization is essentially the only way we know of proving separations between complexity classes. The basic principle is the same as in Cantor’s proof that the set of real
numbers is not countable. First note that if the set of real numbers \( r \) in the range \([0, 1)\) is countable then the set of infinite binary sequences is countable: we can identify a real number \( r \) in \([0, 1)\) with its binary expansion \( r = \sum_{j=1}^{\infty} 2^{-j} r[j] \). If we had an enumeration of real numbers, then we would also have an enumeration of infinite binary string. (The only thing to watch for is that some real numbers may have two possible binary representations, like 0.01000 ... and 0.001111 ...)

So, suppose towards a contradiction that the set of infinite binary sequences were countable, and let \( B_i[j] \) be the \( j \)-th bit of the \( i \)-th infinite binary sequence. Then define the sequence \( B \) whose \( j \)-th bits is \( 1 - B_i[j] \). This is a well-defined sequence but there can be no \( i \) such that \( B = B_i \), because \( B \) differs from \( B_i \) in the \( i \)-th bit.

Similarly, we can prove that the Halting problem is undecidable by considering the following decision problem \( D \): on input \( \langle M \rangle \), the description of a Turing machine, answer NO if \( M(\langle M \rangle) \) halts and accepts and YES otherwise. The above problem is decidable if the Halting problem is decidable. However, suppose \( D \) where decidable and let \( T \) be a Turing machine that solves \( D \), then \( T(\langle T \rangle) \) halts and accepts if and only if \( T(\langle T \rangle) \) does not halt and accept, which is a contradiction.

It is easy to do something similar with time-bounded computations. Instead of the general halting problem, we can define a “bounded” halting problem, for example we can define the problem \( BH \) where given a Turing machine \( \langle M \rangle \) and a string \( x \) we answer YES if \( M \) accepts \( x \) within \( |x|^3 \) steps and NO otherwise. Then we can define the problem \( D \) where on input \( \langle M \rangle \) and \( x \) we answer YES if \( M(\langle M \rangle, x) \) rejects within \( n^3 \) steps, where \( n \) is the length of \( \langle M \rangle, x \), and NO otherwise. Clearly, there cannot be a machine that solves \( D \) in time less than \( n^3 \) on inputs of length \( n \), and, similarly, we can deduce that \( BH \) cannot be solved in time \( o(n^3) \), because an algorithm that solves \( BH \) in time \( t(n) \) can be easily modified to solve \( D \) in time \( O(t(2n)) \). Finally, we note that \( BH \) can be solved in time polynomial in \( n \), which shows DTIME\((n^3) \neq P \). Just by replacing the bound \( n^3 \) with other bounds, one can show that, for every \( k \), DTIME\((O(n^k)) \neq P \), or that DTIME\((2^n) \neq DTIME(2^{2n}) \) and so on.

If we want to show tighter separations, however, such as DTIME\((n^3) \neq DTIME(2^{n^3}) \), we need to be more careful in the definition of our “diagonal” problem \( D \). If \( D \) is defined as the problem of recognizing pairs \( \langle (M, x) \rangle \) such that \( M(\langle M \rangle, x) \) rejects within \( n^3 \) steps, where \( n \) is the length of \( \langle M \rangle, x \), then it is possible then, in an input of length \( n \) for \( D \), \( n/2 \) bits, say, are devoted to the description of \( \langle M \rangle \). To simulate on step of the computation of \( M \), then, a universal Turing machine must scan the entire description of \( M \), and so each step of the simulation will take at least \( \Omega(n) \) time, and the total running time for deciding \( D \) will be \( \Omega(n^4) \). We can overcome this problem if we define \( D \) not in terms of the running time of \( M \), but rather in terms of the running time of a fixed universal Turing machine that simulates \( M \). We first state a result about efficient universal Turing machines.

**Lemma 2 (Efficient Universal Turing Machine)** There is a Turing machine \( U \) that, on input the description \( \langle M \rangle \) of a Turing machine \( M \) and a string \( x \), behaves like \( M \) on input \( x \), that is, if \( M(x) \) accepts then \( U(\langle M \rangle, x) \) accepts, if \( M(x) \) rejects then \( U(\langle M \rangle, x) \) rejects, and if \( M(x) \) does not halt then \( U(\langle M \rangle, x) \) does not halt. Furthermore, if \( M(x) \) halts within \( t \) steps then \( U(\langle M \rangle, x) \) halts within \( O(|\langle M \rangle|^{O(1)} \cdot t) \) steps.

We can now make an argument for a very tight “hierarchy” theorem.
Theorem 3 \( \text{DTIME}(o(n^3)) \not\subseteq \text{DTIME}(O(n^3 \log n)) \).

Proof: Consider the following decision problem \( D \): on input \( (M, x) \) answer YES if \( U \) rejects \( (M, x) \) within \( |x|^3 \) steps, and NO otherwise.

The problem can be solved in \( O(n^3(\log n)) \) time. We just need to modify \( U \) so that it initializes a counter at \( |x|^3 \), decreases its value after each step, and, if it hasn’t halted already, it halts when the counter reaches zero. With some care, maintaining and updating the counter can be done in time \( O(\log n) \) per step even on a one-tape Turing machine.

Suppose by contradiction that \( D \) is solvable in time \( t(n) = o(n^3) \) on inputs of length \( n \) by a machine \( T \). Then, for every \( x \) of length \( n \), \( U((T), x) \) also halts in \( o(n^3) \) time. Let \( x \) be sufficiently long so that \( U((T), x) \) halts in less than \( |x|^3 \) time. Then, if \( x \) is a YES instance of \( D \), it means that \( U \) rejects \( (T, x) \) within \( |x|^3 \) time, which means that \( T \) rejects \( x \), which means that \( T \) is incorrect on input \( x \). Similarly, if \( x \) is a NO instance of \( D \), then \( U \) does not reject \( (T, x) \) within \( |x|^3 \) time, but \( U \) halts within \( |x|^3 \) time, and so it follows that \( U \) accepts \( (T, x) \), and so \( T \) accepts \( x \), incorrectly. \( \Box \)

See the homeworks for generalizations of the above proof.

We would like to do the same for non-deterministic time, but we run into the problem that we cannot ask a non-deterministic machine to reject if and only if a non-deterministic machine of comparable running time accepts. If we could do so, then we would be able to prove \( \text{NP} = \text{coNP} \). A considerably subtler argument must be used instead, which uses the following simple fact.

Theorem 4 On input the description \( \langle M \rangle \) of a non-deterministic Turing machine \( M \), a string \( x \) and an integer \( t > n \), the problem of deciding whether \( M \) accepts \( x \) within \( t \) steps is solvable in deterministic \( |\langle M \rangle|O(1)2^{O(t)} \) time.

We will also need a theorem about efficient universal non-deterministic Turing machines.

Lemma 5 There is a non-deterministic Turing machine \( NU \) that, on input the description \( \langle M \rangle \) of a non-deterministic Turing machine \( M \) and a string \( x \):

- If \( M(x) \) has an accepting computation of length \( t \) then \( NU(\langle M \rangle, x) \) has an accepting computation of length \( O(|\langle M \rangle|^{O(1)} \cdot t) \)

- If no computational paths of \( M(x) \) accepts, then no computational paths of \( NU(\langle M \rangle, x) \) accepts.

Finally, we can state and prove a special case of the non-deterministic hierarchy theorem.

Theorem 6 \( \text{NTIME}(o(n^3)) \not\subseteq \text{NTIME}(O(n^3 \log n)) \).

Proof: Let \( f : \mathbb{N} \to \mathbb{N} \) be defined inductively so that \( f(1) = 2 \) and \( f(k + 1) = 2^{k - (f(k))^3} \). Consider the following decision problem \( D \): on input \( x = (\langle M \rangle, 1^t) \), where \( M \) is a non-deterministic Turing machine,

1. if \( t = f(k) \) for some \( k \), then the answer is YES if and only if the simulation of \( M(\langle M \rangle, 1^{1+k-f(k-1)}) \) as in Theorem 4 returns NO within \( t \) steps,
2. otherwise answer YES if and only if $NU(\langle M \rangle, \langle M \rangle, 1^{t+1})$ accepts within $t^3$ steps.

We first observe that $D$ is solvable by a non-deterministic Turing machine running in $O(n^3 \log n)$ time, where $n$ is the input length.

Suppose that $D$ were decided by a non-deterministic Turing machine $T$ running in time $o(n^3)$, and consider inputs of the form $\langle T \rangle, 1^t$ (which are solved by $T$ in time $o(t^3)$). Pick a sufficiently large $k$, and consider the behaviour of $T$ on inputs $\langle T \rangle, 1^t$ for $f(k-1) < t < f(k)$; since all such inputs fall in case (2), we have that $T(\langle T \rangle, 1^t) = T(\langle T \rangle, 1^{t+1})$ for all such $t$ and, in particular,

$$T(\langle T \rangle, 1^{1+f(k-1)}) = T(\langle T \rangle, 1^{f(k)}) \quad (1)$$

On the other hand, the input $\langle T \rangle, 1^{f(k)}$ falls in case (2), and since $T(\langle T \rangle, 1^{1+f(k-1)})$ can be simulated deterministically in time $2^{(k-1)-(f(k-1))^3}$, if $k$ is large enough, and so the correct answer on input $\langle T \rangle, 1^{f(k)}$ is NO if and only if $T(\langle T \rangle, 1^{1+f(k-1)})$ accepts, which is in contradiction to Equation 1. □

5 References

The time hierarchy theorem is proved in [HS65], which is also the paper that introduced the term “Computational Complexity.” The non-deterministic hierarchy theorem is due to Cook [Coo73]. The notion of NP-completeness is due to Cook [Coo71] and Levin [Lev73], and the recognition of its generality is due to Karp [Kar72].

References


Exercises

1. Show that if \( \mathbf{P} = \mathbf{NP} \) for decision problems, then every \( \mathbf{NP} \) search problem can be solved in polynomial time.

2. Generalize Theorem 3. Say that a monotone non-decreasing function \( t : \mathbb{N} \to \mathbb{N} \) is time-constructible if, given \( n \), we can compute \( t(n) \) in \( O(t(n)) \) time. Show that if \( t(n) \) and \( t'(n) \) are two time-constructible functions such that \( t'(n) > t(n) > n^3 \) and \( \lim_{n \to \infty} \frac{t(n)\log t(n)}{t'(n)} = 0 \) then \( \text{DTIME}(t'(n)) \not\subseteq \text{DTIME}(t(n)) \).
Notes for Lecture 26

Circuit Lower Bounds for Parity Using the Switching Lemma

In this lecture we give an alternate proof that parity $\notin \mathbf{AC}^0$ using the technique of random restrictions. This is the original method that was used to prove parity $\notin \mathbf{AC}^0$.

We will describe a proof of the following tight result.

**Theorem 1** If $C$ is a circuit of size $S$ and depth $d$ that computes the parity of $n$ inputs, then

$$S \geq 2^{\Omega(n^{1/d-1})}$$

1 Conventions About the Circuit

**Convention 1** We count NOT gates neither towards the size nor towards the depth of the circuit. (This only makes the lower bound stronger.)

**Lemma 2** If $C$ is a circuit of size $S$ and depth $d$, then there is a circuit $C'$ of size at most $2S$ and depth $d$ that computes the same function and such that all the not gates are applied at the input level.

**Proof**: We prove the stronger statement that, for every gate $g$ of $C$, there is a gate $g'$ in $C'$ whose output is the complement of $g$. Then we just let the output of $C'$ be the complement of the output gate of $C$.

Let us order the gates of $C$ as $g_1, \ldots, g_S$ in such a way that if the gate $g_i$ uses the output of gate $g_j$ as an input then $j < i$. We describe an inductive construction. Regarding $g_1$, if $g_1$ is a AND gate (respectively, an OR gate), then $g'_1$ is an OR gate (respectively, an AND gate), whose inputs are the complements of the inputs of $g_1$. It follows from De Morgan’s law that the output of $g'_1$ is the complement of the output of $g_1$. Now, if we have constructed gates $g'_1, \ldots, g'_i$ whose outputs are the complement of $g_1, \ldots, g_i$, then it’s easy to define $g'_{i+1}$ using the same idea. □

**Convention 2** From now on, we restrict ourselves to circuits where the NOT gates are only applied at the input level. By Lemma 2, this only affects the lower bound by a multiplicative factor of two.

**Lemma 3** If $C$ is a circuit of size $S$ and depth $d$, then there is a circuit $C'$ of size at most $dS$ and depth $d$ that computes the same function and such that

- The gates are arranged in $d$ layers, so that a gate at layer $i$ takes inputs only from gates at layer $i-1$. The gates at layer 1 take as inputs only the inputs of the circuit.
Each layer contains either only OR gates or only AND gates. Layers with OR gates and layers with AND gates alternate.

If $C$ satisfied Convention 2, then $C'$ also satisfies Convention 2.

Proof: Using the associativity of AND and OR, we can make sure that, in every input-output path in the circuit, we always see an alternation between AND gates and OR gates. Said another way, we can make sure that the inputs to each AND gates are coming only from OR gates (or from inputs to the circuit), and vice versa. Suppose that there is an AND gate $g$ in the circuit one of whose inputs is coming from another AND gate $g'$: then we can connect the inputs of $g'$ directly to $g$. This does not change the size or depth of the circuit, and it reduces the number of “violations” of the above property. By repeated applications of the same rule, we eventually get a circuit of the same size and depth of $C$ and such that each AND gate take only inputs from OR gates (and possibly from inputs of the circuit) and vice versa.

Finally, we arrange the gates in $d$ layers so that each layer is made of either all AND or all OR gates, and wires go only from lower-numbered layers to higher-numbered layers. Finally, we replace all wires that skip many layers by a path of alternating fan-in 1 AND gates and fan-in 1 OR gates. This step increases the size of the circuit by at most a factor of $d$. $\square$

Convention 3 From now on, we restrict ourselves to circuits as in the conclusion of Lemma 3.

2 Random Restrictions and Sketch of the Lower Bound Proof

A restriction fixes some inputs of a circuit to constant values, and leaves other inputs free. More formally, a restriction is a function $\rho : \{1, \ldots, n\} \to \{0, 1, \ast\}$. Applying a restriction $\rho$ to a circuit $C$ with $n$ inputs defines a restricted circuit $C_\rho$ as follows:

- For each $i \in \{1, \ldots, n\}$:
  - If $\rho(i) = 0$, set the $i$th input of $C$ to 0.
  - If $\rho(i) = 1$, set the $i$th input of $C$ to 1.
  - If $\rho(i) = \ast$, leave the $i$th input of $C$ as a free variable.

The proof of Theorem 1 has, roughly, the following structure. First, we show that parity circuits of depth 2 must have exponential size and use gates of linear width (Lemma 4). Next, we suppose by way of contradiction that we have a circuit $C$ of some constant depth $d$ which computes parity. We give a way of squashing $C$ down to depth $d - 1$ while still computing parity on many variables. We can repeat the method $d - 2$ times to obtain a parity circuit of depth 2 and sublinear width, which contradicts Lemma 4. (The actual proof, indeed, will be a direct one, and we will not need to argue by contradiction.)

Restrictions enter in our method of circuit squashing. We show that, after a random restriction, the top two layers of the circuit can be replaced by an equivalent set of gates but with a switched order of AND and OR gates. That is, if, before the restriction the top
layer had AND gates and second layer had OR gates, then, after the restriction, the first
two layers can be equivalently realized with OR gates at the first layer and AND gates and
the second layer, with no significant size increase.\footnote{In the actual proof, it will be convenient to just keep track of the width of the top gates and of the size of the circuit not counting the first layer, instead of the total size. Through the construction, neither of these parameters will increase at all.} Using associativity, we can then collapse
the second and the third layer, obtaining a depth \((d - 1)\) circuit.

3 Proof of the Lower Bound

We begin with the simple case of depth 2.

**Lemma 4** If a DNF or a CNF computes parity of \(n\) variables, then:

1. Each term includes all \(n\) variables, and
2. There are at least \(2^{n-1}\) terms.

**Proof:** We will prove the lemma for CNFs, which have OR gates at their top level and a single AND of all the ORs at the second level. The proof for DNFs is quite similar.

For any CNF circuit \(C\):

1. *Each term includes all \(n\) variables:* Suppose by way of contradiction that \(C\) has some term \(t\) which does not depend on some variable \(x_i\). Then when all inputs to \(t\) are 0, \(t\) outputs 0 and the single AND gate on the next level outputs 0, which is the output of the whole circuit. Now flip the value of \(x_i\). The output of \(t\) is still 0, and thus the output of \(C\) has not changed. But since we've only changed one variable, the parity has flipped. Alas, we have a contradiction! So every term must depend on all variables.

2. *There are at least \(2^{n-1}\) terms:* To compute parity, \(C\) must output 0 on \(2^{n-1}\) different settings of the input variables. \(C\) outputs 0 only when one of the terms (OR gates) outputs 0. But each OR gate outputs 0 on exactly one setting of the input variables. Thus, \(C\) must have at least \(2^{n-1}\) terms.

\(\Box\)

The following result is the technical core of the lower bound. It has a difficult proof that we omit

**Lemma 5** *(Switching Lemma)* Suppose \(f\) is a \(k\)-CNF or \(k\)-DNF over the variables \(x_1, \ldots, x_n\). Pick at random a restriction that leaves a fraction \(p\) of the variables unfixed. For each of the \(n(1 - p)\) variables that are fixed, independently hardwire 0 or 1 as that variable’s value. Then for every \(t\),

\[
\Pr[\text{after the restriction } f \text{ can be expressed as a decision tree of depth } t ] > 1 - (7pk)^t.
\]
Notice that if a function can be specified by a depth-$t$ decision tree then, for a stronger reason, it can be specified by a $t$-CNF and also by a $t$-DNF. We proceed with the proof of Theorem 1.

**Proof:** Let $C$ be a depth $d$ circuit for parity of size $S$, satisfying the conventions specified earlier.

Consider the gates at the first level, and suppose they are OR gates (a symmetric argument applies if they are AND gates). We think of each such gate as a 1-DNF formula. We apply the Switching Lemma with $t = \log S$ and $p = 1/14$, and we deduce that there relative to a random restriction each of the top-level gates becomes a $\log S$-CNF with probability bigger than $1 - 1/S$. In particular, there is a random restriction that makes all top-level gates expressible as a $\log S$-CNF formula. We apply such a restriction, we substitute each top gate by a $\log S$-CNF, and finally we use associativity to collapse the AND gate of each CNF into the AND gates of the second level of the original circuit.

Now we have a circuit of depth $d$ such that each top gate has fan-in at most $\log S$, there are at most $S$ gates from level 2 to level $d$, and the circuit computes parity of $n/14$ variables.

Now we apply the Switching Lemma with $k = \log S$, $p = 1/(14 \log S)$ and $t = \log S$. We get that, for each of the AND gates at level 2, after the restriction the gate can be replaced by a $(\log S)$-DNF with probability more than $1 - 1/S$. Then, there is a restriction for which this is true for all the at most $S$ gates at level 2, we apply this restriction, we replace each level-2 gate with a $(\log S)$-DNF, and we collapse level 2 with level 3.

Now we have a circuit of depth $d - 1$ that computes parity of $n/(14 \cdot (14 \log S))$ bits, and such that every top gate has fan-in at most $\log S$ and there are at most $S$ gates from level 2 to level $d - 1$.

If we repeat the same argument another $d - 3$ times, we end up with a circuit of depth 2 such that the fan-in of the top gates is at most $\log S$ and the circuit computes parity of $n/(14 \cdot (14 \log S)^{d-2})$ inputs. From Lemma 4 we have

$$\log S \geq \frac{1}{14 \cdot (14 \log S)^{d-2}}$$

which is equivalent to

$$S \geq 2^{\frac{n}{d-1}}.$$

□

4 References

The idea of using random restrictions, as well as the first proof that Parity is in $\text{AC}^0$, is due to Furst, Saxe and Sipser [FSS84]. The lower bound was improved to exponential by Yao [Yao85], and the optimal lower bound is due to Håstad [Hås86].

References


the new decade.

## Classes and Lecture Notes

1. **Lecture 1** (8/30) Introduction [notes]
2. **Lecture 2** (9/1) P and NP, hierarchy theorems. (same notes as lecture 1)

   - 9/6 No class (Labor day)
3. **Lecture 3** (9/8) More on P, NP and hierarchy theorems (same notes as lecture 1)
4. **Lecture 4** (9/13) Space bounded computations [notes - revised 10/6]
5. **Lecture 5** (9/15) NL-completeness, Savitch's theorem, NL=coNL (same notes as last lecture)
6. **Lecture 6** (9/20) Non-uniform computations [notes]
7. **Lecture 7** (9/22) Polynomial hierarchy, Karp-Lipton theorem [notes]
8. **Lecture 8** (9/27) Probabilistic algorithms [notes]
9. **Lecture 9** (9/29) Valiant-Vazirani (same notes as last lecture)
10. **Lecture 10** (10/4) #P and approximate counting [notes]
11. **Lecture 11** (10/6) #P and approximate counting (same notes as last lecture)
   (see these notes on probability if you had difficulty following today's proofs)
12. **Lecture 12** (10/11) One-way permutations and pseudorandom generators [notes - revised 11/30]
13. **Lecture 13** (10/13) more on one-way permutations and pseudorandom generators (same notes as last lecture)

   - 10/18 and 10/20 No class (FOCS)
   - this is a good time to start thinking about the final project
15. **Lecture 15** (10/27) Learning decision trees (see this very well written paper) [notes]
   - midterm to be posted on 10/31, due 11/8
16. **Lecture 16** (11/1) More on learning decision trees (same notes as last lecture)
17. **Lecture 17** (11/3) Pseudorandom functions and Razborov Rudich (notes in preparation)
18. **Lecture 18** (11/8) More on pseudorandom functions and Razborov Rudich (notes in preparation)
19. **Lecture 19** (11/10) Levin's theory of average-case complexity [notes]
21. **Lecture 21** (11/17) Even more on Levin's theory of average-case complexity
22. **Lecture 22** (11/22) Reingold's theorem [notes]
23. **Lecture 23** (11/24) Reingold's theorem, continued [notes]
24. **Lecture 24** (11/29) Reingold's theorem, end
25. **Lecture 25** (12/1) Parity is not in AC0: proof with polynomials [notes]
26. **Lecture 26** (12/6) Parity is not in AC0: proof with the switching lemma [notes]
27. **Lecture 27** (12/8) Applications of parity lower bounds to learning [notes]

## Midterm and Project
Solutions to homeworks for the first 14 lectures

Midterm and Midterm Solutions

Information about the final project (accessible only from within the berkeley.edu domain)

3,453 since 8/30/04