**Notes on Complexity Theory** Last updated: August, 2011

# Lecture 1

*Jonathan Katz*

## 1 Turing Machines

I assume that most students have encountered Turing machines before. (Students who have not may want to look at Sipser's book [3].) A Turing machine is defined by an integer $k \geq 1$, a finite set of states $Q$, an alphabet $\Gamma$, and a transition function $\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{L, S, R\}^k$ where:

- $k$ is the number of (infinite, one-dimensional) tapes used by the machine. In the general case we have $k \geq 3$ and the first tape is a read-only *input tape*, the last is a write-once *output tape*, and the remaining $k-2$ tapes are *work tapes*. For Turing machines with boolean output (which is what we will mostly be concerned with in this course), an output tape is unnecessary since the output can be encoded into the final state of the Turing machine when it halts.
- $Q$ is assumed to contain a designated start state $q_s$ and a designated halt state $q_h$. (In the case where there is no output tape, there are two halting states $q_{h,0}$ and $q_{h,1}$.)
- We assume that $\Gamma$ contains $\{0, 1\}$, a "blank symbol", and a "start symbol".
- There are several possible conventions for what happens when a head on some tape tries to move left when it is already in the left-most position, and we are agnostic on this point. (Anyway, by our convention, below, that the left-most cell of each tape is "marked" there is really no reason for this to ever occur...).

The computation of a Turing machine $M$ on input $x \in \{0, 1\}^*$ proceeds as follows: All tapes of the Turing machine contain the start symbol followed by blank symbols, with the exception of the input tape which contains the start symbol followed by $x$ (and then the remainder of the input tape is filled with blank symbols). The machine starts in state $q = q_s$ with its $k$ heads at the left-most position of each tape. Then, until $q$ is a halt state, repeat the following:

1. Let the current contents of the cells being scanned by the $k$ heads be $\gamma_1, \ldots, \gamma_k \in \Gamma$.
2. Compute $\delta(q, \gamma_1, \ldots, \gamma_k) = (q', \gamma'_2, \ldots, \gamma'_k, D_1, \ldots, D_k)$ where $q' \in Q$ and $\gamma'_2, \ldots, \gamma'_k \in \Gamma$ and $D_i \in \{L, S, R\}$.
3. Overwrite the contents of the currently scanned cell on tape $i$ to $\gamma'_i$ for $2 \leq i \leq k$; move head $i$ to the left, to the same position, or to the right depending on whether $D_i = L, S$, or $R$, respectively; and then set the current state to $q = q'$.

The output of $M$ on input $x$, denoted $M(x)$, is the binary string contained on the output tape (between the initial start symbol and the trailing blank symbols) when the machine halts. (When there is no output tape, then the output is '1' if $M$ halts in state $q_{h,1}$ and the output is '0' is $M$ halts in state $q_{h,0}$.) It is also possible that $M$ never halts when run on some input $x$. We return to this point later.

The *running time* of a Turing machine $M$ on input $x$ is simply the number of "steps" $M$ takes before it halts; that is, the number of iterations (equivalently, the number of times $\delta$ is computed) in the above loop. Machine $M$ is said to run in time $T(\cdot)$ if for every input $x$ the running time of $M(x)$ is at most $T(|x|)$. The *space* used by $M$ on input $x$ is the number of cells written to by $M$ on all its *work* tapes[1] (a cell that is written to multiple times is only counted once); $M$ is said to use space $T(\cdot)$ if for every input $x$ the space used during the computation of $M(x)$ is at most $T(|x|)$. We remark that these time and space measures are *worst-case* notions; i.e., even if $M$ runs in time $T(n)$ for only a fraction of the inputs of length $n$ (and uses less time for all other inputs of length $n$), the running time of $M$ is still said to be $T$. (Average-case notions of complexity have also been considered, but are somewhat more difficult to reason about.)

A Turing machine $M$ *computes a function* $f : \{0,1\}^* \to \{0,1\}^*$ if $M(x) = f(x)$ for all $x$. Assuming $f$ is a total function, and so is defined on all inputs, this in particular means that $M$ halts on all inputs. We will focus most of our attention on boolean functions, a context in which it is more convenient to phrase computation in terms of *languages*. A language is simply a subset of $\{0,1\}^*$. There is a natural correspondence between languages and boolean functions: for any boolean function $f$ we may define the corresponding language $L$ as the set $L = \{x \mid f(x) = 1\}$. Conversely, for any language $L$ we can define the boolean function $f$ so that $f(x) = 1$ iff $x \in L$. A Turing machine $M$ *decides a language* $L$ if

$$x \in L \Leftrightarrow M(x) = 1$$

(we sometimes also say that $M$ *accepts* $L$, though we will try to be careful); this is the same as computing the boolean function $f$ that corresponds to $L$.

## 1.1 Comments on the Model

Turing machines are *not* meant as a model of modern computer systems. Rather, they were introduced (before computers were even built!) as a mathematical model of *what computation is*. Explicitly, the axiom is that "any function that can be computed in the physical world, can be computed by a Turing machine"; this is the so-called *Church-Turing thesis*. (The thesis cannot be proved unless one can formally define what it means to "compute a function in the physical world" without reference to Turing machines. In fact, several alternate notions of computation have been defined and shown to be equivalent to computation by a Turing machine; there are no serious candidates for alternate notions of computation that are *not* equivalent to computation by a Turing machine. See [1] for further discussion.) In fact, an even stronger axiom known as the *strong Church-Turing thesis* is sometimes assumed to hold: this says that "any function that can be computed in the physical world, can be computed with at most a polynomial reduction in efficiency by a Turing machine". This thesis is challenged by notions of *randomized* computation that we will discuss later. In the past 15 years or so, however, this axiom has been called into question by results on *quantum computing* that show polynomial-time algorithms in a quantum model of computation for problems not known to have polynomial-time algorithms in the classical setting.

There are several variant definitions of Turing machines that are often considered; none of these contradict the strong Church-Turing thesis. (That is, any function that can be computed on any of these variant Turing machines, including the variant defined earlier, can be computed on any other

[1] Note that we do not count the space used on the input or output tapes; this allows us to meaningfully speak of sub-linear space machines (with linear- or superlinear-length output).

variant with at most a polynomial increase in time/space.) Without being exhaustive, we list some examples (see [1, 2] for more):

- One may fix $\Gamma$ to *only* include $\{0, 1\}$ and a blank symbol.
- One may restrict the tape heads to only moving left or right, not staying in place.
- One may fix $k = 3$, so that there is only one work tape. In fact, one may even consider $k = 1$ so that there is only a single tape that serves as input tape, work tape, and output tape.
- One can allow the tapes to be infinite in both directions, or two-dimensional.
- One can allow random access to the work tapes (so that the contents of the $i$th cell of some tape can be read in one step). This gives a model of computation that fairly closely matches real-world computer systems, at least at an algorithmic level.

The upshot of all of this is that it does not matter much which model one uses, as long as one is ok with losing polynomial factors. On the other hand, if one is concerned about "low level" time/space complexities then it is important to fix the exact model of computation under discussion. For example, the problem of deciding whether an input string is a palindrome can be solved in time $O(n)$ on a two-tape Turing machine, but requires time $\Omega(n^2)$ on a one-tape Turing machine.

## 1.2 Universal Turing Machines and Uncomputable Functions

An important observation (one that is, perhaps, obvious nowadays but was revolutionary in its time) is that *Turing machines can be represented by binary strings.* In other words, we can view a "program" (i.e., a Turing machine) equally well as "data", and run one Turing machine on (a description of) another. As a powerful example, a *universal* Turing machine is one that can be used to simulate any other Turing machine. We define this next.

Fix some representation of Turing machines by binary strings, and assume for simplicity that every binary string represents some Turing machine (this is easy to achieve by mapping badly formed strings to some fixed Turing machine). Consider the function $f(M, x) = M(x)$. Is $f$ computable?

> **Note:** Here $f$ is a partial function, since in this context the given Turing machine $M$ may not halt on the given input $x$ and we leave $f$ undefined in that case. A partial function $f$ is computable if there is a Turing machine $U$ such that for all $x$ where $f$ is defined we have $U(x) = f(x)$. When $f(x)$ is undefined the behavior of $U$ may be arbitrary. An alternative is to consider the (total) function
>
> $$f'(M, x, 1^t) = \begin{cases} 1 & \text{if } M(x) \text{ halts within } t \text{ steps with output } 1 \\ 0 & \text{otherwise} \end{cases},$$
>
> whose computability is closely linked to that of $f$. Another natural possibility is to consider the (total) function
>
> $$f_{halt}(M, x) = \begin{cases} 1 & \text{if } M(x) \text{ halts with output } 1 \\ 0 & \text{otherwise} \end{cases};$$
>
> as we will see, however, $f_{halt}$ is *not* computable.

Perhaps surprisingly, $f$ is computable. We stress that here we require there to be a *fixed* Turing machine $U$, with a fixed number of tapes and a fixed alphabet (not to mention a fixed set of states) that can simulate the behavior of an *arbitrary* Turing machine $M$ that may use any number of tapes and any size alphabet. A Turing machine computing $f$ is called a *universal* Turing machine.

**Theorem 1** *There exists a Turing machine $U$ such that (1) $U(M, x) = M(x)$ for all $x$ for which $M(x)$ is defined; furthermore, (2) for every $M$ there exists a constant $c$ such that the following holds: for all $x$, if $M(x)$ halts within $T$ steps, then $U(M, x)$ halts within $c \cdot T \log T$ steps.*

**Proof** We only sketch the proof here. We consider the case where $M$ computes a boolean function, and so has no output tape; $U$ will not have an output tape either. $U$ will use 3 work tapes, and the alphabet $\Gamma$ that only includes $\{0, 1\}$, a blank symbol, and a start symbol. At a high level, $U$ proceeds as follows:

1. First, $U$ applies a transformation to $M$ that results in a description of an equivalent machine $M'$ that uses only a single work tape (in addition to its input tape). This is known to be possible, and moreover is possible in such a way that the following holds: if $M(x)$ halts within $T$ steps, then $M'(x)$ halts within $O(T \log T)$ steps (see [2, Chap. 12] or [1, Sect. 17]). The description of $M'$ is stored on the second work tape of $U$ (the remaining work tapes of $U$ are used to perform the transformation).

2. Next, $U$ applies a transformation to $M'$ that results in a description of an equivalent machine $M''$ that uses the binary alphabet (plus blank and start symbol). This is known to be possible with only a constant-factor loss of efficiency (see [1, 2]). Thus if $M(x)$ halts within $T$ steps, then $M''(x)$ halts within $O(T \log T)$ steps. The description of $M''$ is stored on the first work tape of $U$ (the 3rd work tape of $U$ can be used to perform the transformation).

3. Finally, $U$ simulates the execution of $M''$ on input $x$. It can do this by recording the current state of $M''$ on its second work tape (recall that the description of $M''$ itself is stored on the first work tape of $U$) and using its third work tape to store the contents of the work tape of $M''$. To simulate each step of $M''$, we have $U$ simply scan the entire description of $M''$ until it finds a transition rule matching the current state of $M''$, the current value being scanned in the input $x$, and the current value being scanned in the work tape of $M''$. This rule is then used by $U$ to update the recorded state of $M''$, to move its heads on its input tape and third work tape (which is storing the work tape of $M''$), and to rewrite the value being scanned on the work tape of $M''$. If $M''$ halts, then $U$ simply needs to check whether the final state of $M''$ was an accepting state or a rejecting state, and move into the appropriate halting state of its own.

It is not hard to see that $U(M, x) = M(x)$ for any $x$ for which $M(x)$ halts. As for the claim about the running time, we note the following: the first and second steps of $U$ take time that depends on $M$ but is *independent* of $x$. In the third step of $U$, each step of $M''$ is simulated using some number of steps that depends on $M''$ (and hence $M$) but is again independent of $x$. We have noted already that if $M(x)$ halts in $T$ steps then $M''(x)$ halts in $c'' \cdot T \log T$ steps for some constant $c''$ that depends on $M$ but not on $x$. Thus $U(M, x)$ halts in $c \cdot T \log T$ steps for some constant $c$ that depends on $M$ but not on $x$. ■

We have shown that (the partial function) $f$ is computable. What about (the function) $f_{halt}$? By again viewing Turing machines as data, we can show that this function is not computable.

**Theorem 2** *The function $f_{halt}$ is not computable.*

**Proof** Say there is some Turing machine $M_{halt}$ computing $f_{halt}$. Then we can define the following machine $M^*$:

> On input (a description of) a Turing machine $M$, output $M_{halt}(M, M)$. If the result is 1, output 0; otherwise output 1.

What happens when we run $M^*$ on *itself*? Consider the possibilities for the result $M^*(M^*)$:

- Say $M^*(M^*) = 1$. This implies that $M_{halt}(M^*, M^*) = 0$. But that means that $M^*(M^*)$ does not halt with output 1, a contradiction.
- Say $M^*(M^*) = 0$. This implies that $M_{halt}(M^*, M^*) = 1$. But that means that $M^*(M^*)$ halts with output 1, a contradiction.
- It is not possible for $M^*(M^*)$ to never halt, since $M_{halt}(M^*, M^*)$ is a total function (and so is supposed to halt on all inputs).

We have reached a contradiction in all cases, implying that $M_{halt}$ as described cannot exist. ■

**Remark:** The fact that $f_{halt}$ is not computable does *not* mean that the halting problem cannot be solved "in practice". In fact, checking termination of programs is done all the time in industry. Of course, they are not using algorithms that are solving the halting problem – this would be impossible! Rather, they use programs that may give *false negative*, i.e., that may claim that some other program does not halt when it actually does. The reason this tends to work in practice is that the programs that people want to reason about in practice tend to have a form that makes them amenable to analysis.

## References


[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[2] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.

[3] M. Sipser. *Introduction to the Theory of Computation* (2nd edition). Course Technology, 2005.

**Notes on Complexity Theory** Last updated: December, 2011

# Lecture 2

*Jonathan Katz*

## 1 Review

The *running time* of a Turing machine $M$ on input $x$ is the number of "steps" $M$ takes before it halts. Machine $M$ is said to run in time $T(\cdot)$ if for every input $x$ the running time of $M(x)$ is at most $T(|x|)$. (In particular, this means it halts on all inputs.) The *space* used by $M$ on input $x$ is the number of cells written to by $M$ on all its *work* tapes[1] (a cell that is written to multiple times is only counted once); $M$ is said to use space $T(\cdot)$ if for every input $x$ the space used during the computation of $M(x)$ is at most $T(|x|)$. We remark that these time and space measures are *worst-case* notions; i.e., even if $M$ runs in time $T(n)$ for only a fraction of the inputs of length $n$ (and uses less time for all other inputs of length $n$), the running time of $M$ is still $T$. (Average-case notions of complexity have also been considered, but are somewhat more difficult to reason about. We may cover this later in the semester; or see [1, Chap. 18].)

Recall that a Turing machine $M$ *computes* a function $f : \{0,1\}^* \to \{0,1\}^*$ if $M(x) = f(x)$ for all $x$. We will focus most of our attention on boolean functions, a context in which it is more convenient to phrase computation in terms of *languages*. A language is simply a subset of $\{0,1\}^*$. There is a natural correspondence between languages and boolean functions: for any boolean function $f$ we may define the corresponding language $L = \{x \mid f(x) = 1\}$. Conversely, for any language $L$ we can define the boolean function $f$ by $f(x) = 1$ iff $x \in L$. A Turing machine $M$ *decides a language* $L$ if

$$\begin{aligned} x \in L &\Rightarrow M(x) = 1 \\ x \notin L &\Rightarrow M(x) = 0 \end{aligned}$$

(we sometimes also say that $M$ *accepts* $L$, though we will try to be careful); this is the same as computing the boolean function $f$ that corresponds to $L$. Note in particular that we require $M$ to halt on all inputs.

**What is complexity theory about?** The fundamental question of complexity theory is to understand the inherent complexity of various languages/problems/functions; i.e., what is the most efficient algorithm (Turing machine) deciding some language? A convenient terminology for discussing this is given by introducing the notion of a *class*, which is simply a set of languages. Two basic classes are:

- $\text{TIME}(f(n))$ is the set of languages decidable in time $O(f(n))$. (Formally, $L \in \text{TIME}(f(n))$ if there is a Turing machine $M$ and a constant $c$ such that (1) $M$ decides $L$, and (2) $M$ runs in time $c \cdot f$; i.e., for all $x$ (of length at least 1) $M(x)$ halts in at most $c \cdot f(|x|)$ steps.)
- $\text{SPACE}(f(n))$ is the set of languages that can be decided using space $O(f(n))$.

[1] Note that we do not count the space used on the input or output tapes; this allows us to meaningfully speak of sub-linear space machines (with linear- or superlinear-length output).

Note that we ignore constant factors in the above definitions. This is convenient, and lets us ignore low-level details about the model of computation.[2]

Given some language $L$, then, we may be interested in determining the "smallest" $f$ for which $L \in \text{TIME}(f(n))$. Or, perhaps we want to show that $\text{SPACE}(f(n))$ is strictly larger than $\text{SPACE}(f'(n))$ for some functions $f, f'$; that is, that there is some language in the former that is not in the latter. Alternately, we may show that one class contains another. As an example, we start with the following easy result:

**Lemma 1** *For any $f(n)$ we have $\text{TIME}(f(n)) \subseteq \text{SPACE}(f(n))$.*

**Proof** This follows from the observation that a machine cannot write on more than a constant number of cells per move. ■

# 2 $\mathcal{P}$, $\mathcal{NP}$, and $\mathcal{NP}$-Completeness

## 2.1 The Class $\mathcal{P}$

We now introduce one of the most important classes, which we equate (roughly) with *problems that can be solved efficiently*. This is the class $\mathcal{P}$, which stands for *polynomial time*:

$$\mathcal{P} \stackrel{\text{def}}{=} \bigcup_{c \geq 1} \text{TIME}(n^c).$$

That is, a language $L$ is in $\mathcal{P}$ if there exists a Turing machine $M_L$ and a polynomial $p$ such that $M_L(x)$ runs in time $p(|x|)$, and $M_L$ decides $L$.

Does $\mathcal{P}$ really capture efficient computation? There are debates both ways:

- For many problems nowadays that operate on extremely large inputs (think of Google's search algorithms), only linear-time are really desirable. (In fact, one might even want *sub*linear-time algorithms, which are only possible by relaxing the notion of correctness.) This is related to the (less extreme) complaint that an $n^{100}$ algorithm is not really "efficient" in any sense.

  The usual response here is that $n^{100}$-time algorithms rarely occur. Moreover, when algorithms with high running times (e.g., $n^8$) *do* get designed, they tend to be quickly improved to be more efficient.

- From the other side, one might object that $\mathcal{P}$ does not capture all efficiently solvable problems. In particular, a *randomized* polynomial-time algorithm (that is correct with high probability) seems to also offer an efficient way of solving a problem. Most people today would agree with this objection, and would classify problems solvable by randomized polynomial-time algorithms as "efficiently solvable". Nevertheless, it may turn out that such problems all lie in $\mathcal{P}$ anyway; this is currently an unresolved conjecture. (We will discuss the power of randomization, and the possibility of derandomization, later in the semester.)

  As mentioned previously, *quantum* polynomial-time algorithms may also be considered "efficient". It is fair to say that until general-purpose quantum computers are implemented, this is still debatable.

[2] This decision is also motivated by "speedup theorems" which state that if a language can be decided in time (resp., space) $f(n)$ then it can be decided in time (resp., space) $f(n)/c$ for any constant $c$. (This assumes that $f(n)$ is a "reasonable" function, but the details need not concern us here.)

Another important feature of $\mathcal{P}$ is that it is closed under composition. That is, if an algorithm $A$ (that otherwise runs in polynomial time) makes polynomially many calls to an algorithm $B$, and if $B$ runs in polynomial time, then $A$ runs in polynomial time. See [1] for further discussion.

## 2.2 The Classes $\mathcal{NP}$ and co$\mathcal{NP}$

Another important class of problems are those whose solutions can be *verified* efficiently. This is the class $\mathcal{NP}$. (Note: $\mathcal{NP}$ does *not* stand for "non-polynomial time". Rather, it stands for "non-deterministic polynomial-time" for reasons that will become clear later.) Formally, $L \in \mathcal{NP}$ if there exists a Turing machine $M_L$ and a polynomial $p$ such that (1) $M_L(x, w)$ runs in time[3] $p(|x|)$, and (2) $x \in L$ iff there exists a $w$ such that $M_L(x, w) = 1$; such a $w$ is called a *witness* (or, sometimes, a *proof*) that $x \in L$. Compare this to the definition of $\mathcal{P}$: a language $L \in \mathcal{P}$ if there exists a Turing machine $M_L$ and a polynomial $p$ such that (1) $M_L(x)$ runs in time $p(|x|)$, and (2) $x \in L$ iff $M_L(x) = 1$.

Stated informally, a language $L$ is in $\mathcal{P}$ if membership in $L$ can be decided efficiently. A language $L$ is in $\mathcal{NP}$ if membership in $L$ can be efficiently verified (given a correct proof). A classic example is given by the following language:

$$\text{IndSet} = \left\{ (G, k) \ : \ \begin{array}{c} G \text{ is a graph that has} \\ \text{an } \textit{independent set} \text{ of size } k \end{array} \right\}.$$

We do not know an efficient algorithm for determining the size of the largest independent set in an arbitrary graph; hence we do not have any efficient algorithm deciding IndSet. However, if we know (e.g., through brute force, or because we constructed $G$ with this property) that an independent set of size $k$ exists in some graph $G$, it is easy to prove that $(G, k) \in$ IndSet by simply listing the nodes in the independent set: verification just involves checking that every pair of nodes in the given set is *not* connected by an edge in $G$, which is easy to do in polynomial time. Note further than if $G$ does *not* have an independent set of size $k$ then there is no proof that could convince us otherwise (assuming we are using the stated verification algorithm).

It is also useful to keep in mind an analogy with mathematical statements and proofs (though the correspondence is not rigorously accurate). In this view, $\mathcal{P}$ would correspond to the set of mathematical statements (e.g., "1+1=2") whose truth can be easily determined. $\mathcal{NP}$, on the other hand, would correspond to the set of (true) mathematical statements that have "short" proofs (whether or not such proofs are easy to find).

We have the following simple result, which is the best known as far as relating $\mathcal{NP}$ to the time complexity classes we have introduced thus far:

**Theorem 2** $\mathcal{P} \subseteq \mathcal{NP} \subseteq \bigcup_{c \geq 1} \text{TIME}(2^{n^c})$.

**Proof** The containment $\mathcal{P} \subseteq \mathcal{NP}$ is trivial. As for the second containment, say $L \in \mathcal{NP}$. Then there exists a Turing machine $M_L$ and a polynomial $p$ such that (1) $M_L(x, w)$ runs in time $p(|x|)$, and (2) $x \in L$ iff there exists a $w$ such that $M_L(x, w) = 1$. Since $M_L(x, w)$ runs in time $p(|x|)$, it can read at most the first $p(|x|)$ bits of $w$ and so we may assume that $w$ in condition (2) has length at most $p(|x|)$. The following is then a deterministic algorithm for deciding $L$:

[3] It is essential that the running time of $M_L$ be measured in terms of the length of $x$ alone. An alternate approach is to require the length of $w$ to be at most $p(|x|)$ in condition (2).

On input $x$, run $M_L(x, w)$ for all strings $w \in \{0,1\}^{\leq p(|x|)}$. If any of these results in $M_L(x, w) = 1$ then output 1; else output 0.

The algorithm clearly decides $L$. Its running time on input $x$ is $O\left(p(|x|) \cdot 2^{p(|x|)}\right)$, and therefore $L \in \text{TIME}\left(2^{n^c}\right)$ for some constant $c$. ■

The "classical" definition of $\mathcal{NP}$ is in terms of non-deterministic Turing machines. Briefly, the model here is the same as that of the Turing machines we defined earlier, except that now there are *two* transition functions $\delta_0, \delta_1$, and at each step we imagine that the machine makes an arbitrary ("non-deterministic") choice between using $\delta_0$ or $\delta_1$. (Thus, after $n$ steps the machine can be in up to $2^n$ possible configurations.) Machine $M$ is said to output 1 on input $x$ if there exists *at least one* sequence of choices that would lead to output 1 on that input. (We continue to write $M(x) = 1$ in this case, though we stress again that $M(x) = 1$ when $M$ is a non-deterministic machine just means that $M(x)$ outputs 1 for *some* set of non-deterministic choices.) $M$ decides $L$ if $x \in L \Leftrightarrow M(x) = 1$. A non-deterministic machine $M$ runs in time $T(n)$ if for every input $x$ and every sequence of choices it makes, it halts in time at most $T(|x|)$. The class $\text{NTIME}(f(n))$ is then defined in the natural way: $L \in \text{NTIME}(f(n))$ if there is a non-deterministic Turing machine $M_L$ such that $M_L(x)$ runs in time $O(f(|x|))$, and $M_L$ decides $L$. Non-deterministic space complexity is defined similarly: non-deterministic machine $M$ uses space $T(n)$ if for every input $x$ and every sequence of choices it makes, it halts after writing on at most $T(|x|)$ cells of its work tapes. The class $\text{NSPACE}(f(n))$ is then the set of languages $L$ for which there exists a non-deterministic Turing machine $M_L$ such that $M_L(x)$ uses space $O(f(|x|))$, and $M_L$ decides $L$.

The above leads to an equivalent definition of $\mathcal{NP}$ paralleling the definition of $\mathcal{P}$:

**Claim 3** $\mathcal{NP} = \bigcup_{c \geq 1} \text{NTIME}(n^c)$.

This is a good exercise; a proof can be found in [1].

*The* major open question of complexity theory is whether $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$; in fact, this is one of the outstanding questions in mathematics today. The general belief is that $\mathcal{P} \neq \mathcal{NP}$, since it seems quite "obvious" that non-determinism is stronger than determinism (i.e., verifying should be easier than solving, in general), and there would be many surprising consequences if $\mathcal{P}$ were equal to $\mathcal{NP}$. (See [1] for a discussion.) But we have had no real progress toward proving this belief.

**Conjecture 4** $\mathcal{P} \neq \mathcal{NP}$.

A (possibly feasible) open question is to prove that non-determinism is even *somewhat* stronger than determinism. It is known that $\text{NTIME}(n)$ is strictly stronger than $\text{TIME}(n)$ (see [2, 3, 4] and references therein), but we do not know, e.g., whether $\text{TIME}(n^3) \subseteq \text{NTIME}(n^2)$.

### 2.2.1 The Class co$\mathcal{NP}$

For any class $\mathcal{C}$, we define the class co$\mathcal{C}$ as $\text{co}\mathcal{C} \stackrel{\text{def}}{=} \left\{L \mid \bar{L} \in \mathcal{C}\right\}$, where $\bar{L} \stackrel{\text{def}}{=} \{0,1\}^* \setminus L$ is the complement of $L$. Applied to the class $\mathcal{NP}$, we get the class co$\mathcal{NP}$ of languages where *non*-membership can be efficiently verified. In other words, $L \in \text{co}\mathcal{NP}$ if there exists a Turing machine $M_L$ and a polynomial $p$ such that (1) $M_L(x, w)$ runs in time[4] $p(|x|)$, and (2) $x \in L$ iff for all $w$ we have $M_L(x, w) = 1$. Note why this (only) implies efficiently verifiable proofs of *non*-membership:

[4] See footnote 3.

a single $w$ where $M_L(x,w) = 0$ is enough to convince someone that $x \notin L$, but a single $w$ where $M_L(x,w) = 1$ means nothing.

A $\mathsf{co}\mathcal{NP}$ language is easily obtained by taking the complement of any language in $\mathcal{NP}$. So, for example, the complement of IndSet is the language

$$\text{NoIndSet} = \left\{ (G,k) \ : \ \begin{array}{c} G \text{ does } \textit{not} \text{ have} \\ \text{an independent set of size } k \end{array} \right\}.$$

Let us double-check that this is in $\mathsf{co}\mathcal{NP}$: we can prove that $(G,k) \notin$ NoIndSet by giving a set of $k$ vertices that *do* form an independent set in $G$ (this assumes the obvious verification algorithm); note that (assuming we use the obvious verification algorithm) we can never be "fooled" into believing that $(G,k)$ is not in NoIndSet when it actually is.

As another example of languages in $\mathcal{NP}$ and $\mathsf{co}\mathcal{NP}$, consider the *satisfiability problem* which asks whether a boolean formula in conjunctive normal form is satisfiable (see [1] for a formal definition if you have not encountered these terms before). That is,

$$\mathsf{SAT} = \{\phi \mid \phi \text{ has a satisfying assignment}\}.$$

Then $\overline{\mathsf{SAT}}$ consists of boolean formulae with *no* satisfying assignment. We have $\mathsf{SAT} \in \mathcal{NP}$ and $\overline{\mathsf{SAT}} \in \mathsf{co}\mathcal{NP}$. As another example, consider the language $\mathsf{TAUT}$ of tautologies:

$$\mathsf{TAUT} = \{\phi : \phi \text{ is satisfied by every assignment}\}.$$

$\mathsf{TAUT}$ is also in $\mathsf{co}\mathcal{NP}$.

The class $\mathsf{co}\mathcal{NP}$ can also be defined in terms of non-deterministic Turing machines. This is left as an exercise.

Note that $\mathcal{P} \subseteq \mathcal{NP} \cap \mathsf{co}\mathcal{NP}$. (Why?) Could it be that $\mathcal{NP} = \mathsf{co}\mathcal{NP}$? Once again, we don't know the answer but it would be surprising if this were the case. In particular, there does not seem to be any way to give an efficiently verifiable proof that, e.g., a boolean formula does *not* have any satisfying assignment (which is what would be implied by $\overline{\mathsf{SAT}} \in \mathcal{NP}$).

**Conjecture 5** *$\mathcal{NP} \neq \mathsf{co}\mathcal{NP}$.*

## 2.3 $\mathcal{NP}$-Completeness

### 2.3.1 Defining $\mathcal{NP}$-Completeness

What does it mean for one language $L'$ to be harder[5] to decide than another language $L$? There are many possible answers to this question, but one way to start is by capturing the intuition that if $L'$ is harder than $L$, then an algorithm for deciding $L'$ should be useful for deciding $L$. We can formalize this idea using the concept of a *reduction*. Various types of reductions can be defined; we start with one of the most central:

**Definition 1** *A language $L$ is* Karp reducible *(or* many-to-one reducible*) to a language $L'$ if there exists a polynomial-time computable function $f$ such that $x \in L$ iff $f(x) \in L'$. We express this by writing $L \leq_p L'$.*

[5]Technically speaking, I mean "at least as hard as".

The existence of a Karp reduction from $L$ to $L'$ gives us exactly what we were looking for. Say there is a polynomial-time Turing machine (i.e., algorithm) $M'$ deciding $L'$. Then we get a polynomial-time algorithm $M$ deciding $L$ by setting $M(x) \stackrel{\text{def}}{=} M'(f(x))$. (Verify that $M$ does, indeed, run in polynomial time.) This explains the choice of notation $L \leq_p L'$. We state some basic properties, all of which are straightforward to prove.

**Claim 6** *We have:*

1. *(Transitivity) If $A \leq_p B$ and $B \leq_p C$ then $A \leq_p C$.*
2. *If $A \leq_p B$ and $B \in \mathcal{NP}$ then $A \in \mathcal{NP}$.*
3. *If $A \leq_p B$ and $B \in \mathcal{NP}$ then $A \in \mathcal{NP}$.*

A problem is $\mathcal{NP}$-hard if it is "at least as hard to solve" as any problem in $\mathcal{NP}$. It is $\mathcal{NP}$-complete if it is $\mathcal{NP}$-hard and also in $\mathcal{NP}$. Formally:

**Definition 2** *Language $L'$ is $\mathcal{NP}$-*hard *if for every $L \in \mathcal{NP}$ it holds that $L \leq_p L'$. Language $L'$ is $\mathcal{NP}$-*complete *if $L' \in \mathcal{NP}$ and $L'$ is $\mathcal{NP}$-hard.*

Note that if $L$ is $\mathcal{NP}$-hard and $L \leq_p L'$, then $L'$ is $\mathcal{NP}$-hard as well.

co$\mathcal{NP}$-completeness is defined analogously: a language $L'$ is co$\mathcal{NP}$-hard if for every $L \in$ co$\mathcal{NP}$ it holds that $L \leq_p L'$; language $L'$ is co$\mathcal{NP}$-complete if $L'$ is co$\mathcal{NP}$-hard and $L' \in$ co$\mathcal{NP}$.

### 2.3.2 Existence of $\mathcal{NP}$-Complete Problems

A priori, it is not clear that there should be any $\mathcal{NP}$-complete problems. One of the surprising results from the early 1970s is that $\mathcal{NP}$-complete problems exist. Soon after, it was shown that many important problems are, in fact, $\mathcal{NP}$-complete. Somewhat amazingly, we now know thousands of $\mathcal{NP}$-complete problems arising from various disciplines.

Here is a trivial $\mathcal{NP}$-complete language:

$$L = \left\{ (M, x, 1^t) : \exists w \in \{0,1\}^t \text{ s.t. } M(x,w) \text{ halts within } t \text{ steps with output } 1. \right\}.$$

Next time we will show more natural $\mathcal{NP}$-complete languages

## References


[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[2] R. Kannan. Towards separating nondeterminisn from determinisn. *Math. Systems Theory* 17(1): 29–45, 1984.

[3] W. Paul, N. Pippenger, E. Szemeredi, and W. Trotter. On determinism versus non-determinisn and related problems. FOCS 1983.

[4] R. Santhanam. On separators, segregators, and time versus space. *IEEE Conf. Computational Complexity* 2001.

**Notes on Complexity Theory** Last updated: October, 2011

Lecture 3

*Jonathan Katz*

# 1 Natural $\mathcal{NP}$-Complete Problems

Last time we saw a "non-natural" $\mathcal{NP}$-complete language. Other important $\mathcal{NP}$-complete languages are SAT (satisfiable boolean formulae in conjunctive normal form) and 3-SAT (satisfiable boolean formulae in conjunctive normal form, where each clause contains at most 3 literals). Besides being more "natural" languages, they are useful for proving $\mathcal{NP}$-completeness of other languages.

**Theorem 1 (Cook-Levin Theorem)** SAT *is $\mathcal{NP}$-complete.*

**Proof** We give a detailed proof sketch. (Note that the proof we give here is different from the one in [1]; in particular, we do not rely on the existence of oblivious Turing machines.)

Let $L$ be a language in $\mathcal{NP}$. This means there is a Turing machine $M$ and a polynomial $p$ such that (1) $M(x, w)$ runs in time $p(|x|)$, and (2) $x \in L$ if and only if there exists a $w$ for which $M(x, w) = 1$. Note that we may assume that any such $w$, if it exists, has length exactly $p(|x|) - |x| - 1$. We also assume for simplicity (and without loss of generality) that $M$ has a single tape (that is used as both its input tape and work tape) and a binary alphabet.

A simple observation is that we can represent the computation of $M(x, w)$ (where $|x| = n$) by a *tableau* of $p(n) + 1$ rows, each $O(p(n))$ bits long. Each row corresponds to the entire configuration of $M$ at some step during its computation; there are $p(n) + 1$ rows since $M$ always halts after at most $p(n)$ steps. (If $M(x, w)$ halts before $p(n)$ steps, the last rows may be duplicates of each other. Or we may assume that $M(x, w)$ always runs for exactly $p(|x|)$ steps.) Each row can be represented using $O(p(n))$ bits since a configuration contains (1) the contents of $M$'s tape (which can be stored in $O(p(n))$ bits — recall that $\text{SPACE}(p(n)) \subseteq \text{TIME}(p(n))$); (2) the location of $M$'s head on its tape (which can be stored in $p(n)$ bits[1]); and (3) the value of $M$'s state (which requires $O(1)$ bits).

Moreover, given a tableau that is claimed to correspond to an accepting computation of $M(x, w)$, it is possible to *verify* this via a series of "local" checks. (This notion will become more clear below.) Specifically, letting $p = p(n)$ and assuming we are given some tableau, do:

1. Check that the first row is formed correctly. (The tape should contain $x$, followed by a space and then a sequence of bits representing $w$; $M$'s head should be in the left-most position; and $M$ should be in its start state.)

2. Number the rows from 0 to $T$, and recall that these correspond to time steps of $M$'s execution. Let $t_{i,j}$ denote the value written in cell $j$ at time $i$. Then for $i = 1, \ldots, T$ and $j = 1, \ldots, T$, check that $t_{i,j}$ has the correct value given $t_{i-1,j-1}, t_{i-1,j}$, and $t_{i-1,j+1}$ and the value of the state at time $i - 1$. We also need to check that the state at time $i$ takes the correct value; this is discussed in detail below.

3. Check that the state in the final row is the accepting state.

[1] In fact, $O(\log p(n))$ bits suffice, but for this proof it is somewhat simpler to use a more wasteful representation.

Each of these checks involves looking at only a small (in fact, constant) part of the tableau. This is important, as it implies that each check can be represented as a constant-size CNF formula. Then correctness of the entire tableau can be represented as a conjunction of a polynomial number of these formulae. We give further details below.

We begin with the following claim:

**Claim 2** *Any function $f : \{0,1\}^\ell \to \{0,1\}$ can be expressed as a CNF formula of size at most $\ell \cdot 2^\ell$.*

**Proof** Let $x = (x_1, \ldots, x_\ell)$ denote the input variables of $f$. For some fixed string $y \in \{0,1\}^\ell$, we can express the predicate "$\mathsf{neq}_y(x) \stackrel{\text{def}}{=} [x \neq y]$" as

$$(x_1 \neq y_1) \vee \cdots \vee (x_\ell \neq y_\ell);$$

remembering that $y$ is fixed, this is just a disjunctive clause in the $\ell$ variables $x_1, \ldots, x_\ell$. If we let $Y \subseteq \{0,1\}^\ell$ denote the set of inputs on which $f$ evaluates to 0, then we can express the predicate "$f(x) = 1$" by

$$\textstyle\bigwedge_{y \in Y} \mathsf{neq}_y(x_1, \ldots, x_\ell),$$

which is a CNF formula of size at most $\ell \cdot 2^\ell$. ■

To prove the theorem, we need to show a polynomial-time transformation $f$ that outputs CNF formula with the property that $x \in L$ iff $f(x) \in \mathsf{SAT}$. Our transformation $f$ will output a CNF formula corresponding to the verification of an accepting tableau of the computation of $M(x, w)$ for some $w$. For a given $x$ of length $n = |x|$, let $p = p(n)$; then $f(x)$ does as follows:

- Create variables $\{t_{i,j}\}$ for $i = 0$ to $p$ and $j = 1$ to $p$. Each $t_{i,j}$ represents the value written in cell $j$ at time $i$. (Each $t_{i,j}$ will actually be two bits, since we need two bits to represent the 0, 1, start symbol, and space character.)
- Create variables $u_{i,j}$ for $i = 0$ to $p$ and $j = 1$ to $p$. Each $u_{i,j}$ is a single bit indicating whether the head is in position $j$ at time $i$.
- Create variables $\vec{s}_i \stackrel{\text{def}}{=} (s_{i,1}, \ldots, s_{i,q})$ for $i = 1$ to $p$ and some constant $q$ that depends on the number of states that $M$ uses. (Namely, if the set of states is $Q$ then $q = \lceil \log |Q| \rceil$.)
- Create the following CNF formulae:
  - $\chi_0$ checks that row 0 is correct: namely, that $t_{0,1}, \ldots, t_{0,p}$ contains a start symbol, followed by $x_1, \ldots, x_\ell$, followed by a blank, and then $\{0,1\}$ in the remaining positions; furthermore, $u_{0,1} = 1$ and $u_{0,j} = 0$ for all $j > 1$, and $\vec{s}_0$ encodes the start state of $M$. Even though $\chi_0$ involves $O(p)$ variables, it is easy to see that it can be expressed as a CNF formula of size $O(p)$.
  - For $i, j = 1$ to $p$, let $\phi_{i,j}$ be a CNF formula that checks correctness of cell $j$ at time $i$. This is a formula in the variables $t_{i,j}$, $u_{i,j}$, the three[2] cells in the neighborhood of cell $j$ at the previous time period (namely, $N_{i-1,j} \stackrel{\text{def}}{=} \{t_{i-1,j-1}, u_{i-1,j-1}, t_{i-1,j}, u_{i-1,j}, t_{i-1,j+1}, u_{i-1,j+1}\}$), and the current and previous states $\vec{s}_i, \vec{s}_{i-1}$. This formula encodes the following predicate:

[2] Of course, if $j = 1$ or $j = p$ then the cell has only two neighbors.

$t_{i,j}, u_{i,j}$ contain the correct values given $N_{i-1,j}$ and $\vec{s}_{i-1}$.

and

if $u_{i,j} = 1$, then $\vec{s}_i$ contains the correct value given $N_{i-1,j}$ and $\vec{s}_{i-1}$.

The above can be a complicated predicate, but it involves only a *constant* (i.e., independent of $n$) number of variables, and hence (by Claim 2) can be encoded by a CNF formula of constant size.

- $\chi_p$ simply checks that $\vec{s}_p$ encodes the accepting state of $M$.
- The output of $f$ is $\Phi = \chi_0 \wedge \left(\bigwedge_{i,j} \phi_{i,j}\right) \wedge \chi_p$.

One can, somewhat tediously, convince oneself that $\Phi$ is satisfiable if and only if there is some $w$ for which $M(x, w) = 1$. ■

To show that 3-SAT is $\mathcal{NP}$-complete, we show a reduction from any CNF formula to a CNF formula with (at most) 3 literals per clause. We illustrate the idea by showing how to transform a clause involving 4 literals to two clauses involving 3 literals each: given clause $a \vee b \vee c \vee d$ we introduce the auxiliary variable $z$ and then output $(a \vee b \vee z) \wedge (\bar{z} \vee c \vee d)$; one can check that the latter is satisfiable iff the former is satisfiable.

### 1.1 Other $\mathcal{NP}$-Complete Problems

SAT and 3-SAT are useful since they can be used to prove many other problems $\mathcal{NP}$-complete. Recall that we can show that some language $L$ is $\mathcal{NP}$-complete by demonstrating a Karp reduction from 3-SAT to $L$. As an example, consider IndSet (see [1] for more details): Given a formula $\phi$ with $n$ variables and $m$ clauses, we define a graph $G$ with $7m$ vertices. There will be 7 vertices for each clause, corresponding to 7 possible satisfying assignments. $G$ contains edges between all vertices that are inconsistent (including those in the same cluster). One can check that there is an independent set of size $m$ iff $\phi$ has a satisfying assignment.

## 2 Self-Reducibility and Search vs. Decision

We have so far been talking mainly about decision problems, which can be viewed as asking whether a solution *exists*. But one often wants to solve the corresponding search problem, namely to *find* a solution (if one exists). For many problems, the two have equivalent complexity.

Let us define things more formally. Say $L \in \mathcal{NP}$. Then there is some polynomial-time Turing machine $M$ such that $x \in L$ iff $\exists w : M(x, w) = 1$. The *decision problem* for $L$ is: given $x$, determine if $x \in L$. The *search problem for* $L$ is: given $x \in L$, find $w$ such that $M(x, w) = 1$. (Note that we should technically speak of the *search problem for $L$ relative to $M$* since there can be multiple non-deterministic Turing machines deciding $L$, and each such machine will define its own set of "solutions". Nevertheless, we stick with the inaccurate terminology and hope things will be clear from the context.) The notion of reducibility we want in this setting is *Cook-Turing reducibility*. We define it for decision problems, but can apply it to search problems via the natural generalization.

**Definition 1** *Language $L$ is* Cook-Turing reducible to *$L'$ if there is a poly-time Turing machine $M$ such that for any oracle $\mathcal{O}'$ deciding $L'$, machine $M^{\mathcal{O}'(\cdot)}$ decides $L$. (I.e., $M^{\mathcal{O}'(\cdot)}(x) = 1$ iff $x \in L$.)*

Note that if $L$ is *Karp*-reducible to $L'$, then there is also a Cook-Turing reduction from $L$ to $L'$. In general, however, the converse is not belied to hold. Specifically, any language in $\mathsf{co}\mathcal{NP}$ is Cook-Turing reducible to any $\mathcal{NP}$-complete language, but there is no Karp-reduction from a $\mathsf{co}\mathcal{NP}$-complete language to a language in $\mathcal{NP}$ unless $\mathsf{co}\mathcal{NP} = \mathcal{NP}$.

Returning to the question of search vs. decision, we have:

**Definition 2** *A language $L \in \mathcal{NP}$ is* self-reducible *if there is a Cook-Turing reduction from the search problem for $L$ to the decision problem for $L$. Namely, there is polynomial-time Turing machine $M$ such that for any oracle $\mathcal{O}_L$ deciding $L$, and any $x \in L$ we have $(x, M^{\mathcal{O}_L(\cdot)}(x)) \in R_L$.*

(One could also ask about reducing the decision problem to the search problem. For languages in $\mathcal{NP}$, however, such a reduction always exists.)

**Theorem 3** $\mathsf{SAT}$ *is self-reducible.*

**Proof** Assume we have an oracle that tells us whether any CNF formula is satisfiable. We show how to use such an oracle to find a satisfying assignment for a given (satisfiable) CNF formula $\phi$. Say $\phi$ is a formula on $n$ variables $x_1, \ldots, x_n$. If $b_1, \ldots, b_\ell \in \{0,1\}$ (with $\ell \leq n$), then by $\phi|_{b_1,\ldots,b_\ell}$ we mean the CNF formula on the variables $x_{\ell+1}, \ldots, x_n$ obtained by setting $x_1 = b_1$, $\ldots$, $x_\ell = b_\ell$ in $\phi$. ($\phi|_{b_1,\ldots,b_\ell}$ is easily computed given $\phi$ and $b_1, \ldots, b_\ell$.) The algorithm proceeds as follows:

- For $i = 1$ to $n$ do:
  - Set $b_i = 0$.
  - If $\phi_{b_1,\ldots,b_i}$ is not satisfiable, set $b_i = 1$. (Note: we determine this using our oracle for $\mathsf{SAT}$.)
- Output $b_1, \ldots, b_n$.

We leave it to the reader to show that this always returns a satisfying assignment (assuming $\phi$ is satisfiable to begin with). ■

The above proof can be generalized to show that *every* $\mathcal{NP}$-complete language is self-reducible.

**Theorem 4** *Every $\mathcal{NP}$-complete language $L$ is self-reducible.*

**Proof** The idea is similar to above, with one new twist. Let $M$ be a polynomial-time non-deterministic Turing machine such that

$$L = \{x \mid \exists w : M(x,w) = 1\}.$$

We first define a new language $L'$:

$$L' = \{(x,b) \mid \exists w' : M(x, bw') = 1\}.$$

I.e., $(x,b) \in L'$ iff there exists a $w$ *with prefix* $b$ such that $M(x,w) = 1$. Note that $L' \in \mathcal{NP}$; thus, there is a Karp reduction $f$ such that $x \in L'$ iff $f(x) \in L$. (Here is where we use the fact that $L$ is $\mathcal{NP}$-complete.)

Assume we have an oracle deciding $L$; we design an algorithm that, given $x \in L$, finds $w$ with $M(x,w) = 1$. Say the length of $w$ (given $x$) is $n = \mathsf{poly}(|x|)$. The algorithm proceeds as follows:

- For $i = 1$ to $n$ do:

- Set $b_i = 0$.
- If $f((x, b_1, \ldots, b_i)) \notin L$, set $b_i = 1$. (We run this step using our oracle for $L$.)

- Output $b_1, \ldots, b_n$.

We leave it to the reader to show that this algorithm gives the desired result. ■

Other languages in $\mathcal{NP}$ (that are not $\mathcal{NP}$-complete) may be self-reducible as well. An example is given by graph isomorphism, a language that is not known (or believed) to be in $\mathcal{P}$ or $\mathcal{NP}$-complete. On the other hand, it is believed that not all languages in $\mathcal{NP}$ are self-reducible. One conjectured example is the natural relation derived from factoring: although compositeness can be decided in polynomial time, we do not believe that polynomial-time factoring algorithms exist.

## References

[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

**Notes on Complexity Theory** Last updated: September, 2011

# Lecture 4

*Jonathan Katz*

## 1 Diagonalization

In this lecture and the next one, we discuss two types of results that are related by the technique used in their proofs. Both kinds of results are also fundamental in their own right.

The common proof technique is called *diagonalization*. It is somewhat difficult to formally define the term, but roughly the idea is that we want to show the existence of some language $L$ (with certain properties) that cannot be decided by any Turing machine within some set $S = \{M_1, \ldots\}$.[1] We do so by starting with some $L_0$ (with the property we want) and then, for $i = 1, \ldots$ changing $L_{i-1}$ to $L_i$ such that none of $M_1, \ldots, M_i$ decide $L_i$. Of course part of the difficulty is to make sure that $L_i$ has the property we want also.

Actually, one can also prove the existence of an undecidable language using this technique (though not quite as explicitly as stated above). Consider an enumeration $x_1, \ldots$ of all binary strings, and an enumeration $M_1, \ldots$ of all Turing machines.[2] Define $L$ as follows: $x_i \notin L$ iff $M_i(x_i) = 1$. (A picture really helps here. For those who have seen it before, this is exactly analogous to the proof that there is no bijection from the integers to the reals. In fact, that gives a 1-line proof of the existence of undecidable languages: the set of languages is uncountable, while the set of Turing machines is countable.) Say some machine $M$ decides $L$, and let $i$ be such that $M = M_i$. But then consider $x_i$: if $M(x_i) = 1$ then $x_i \notin L$ and so $M$ is wrong; if $M(x_i)$ rejects or doesn't halt then $x_i \in L$ and $M$ is again wrong!

### 1.1 Hierarchy Theorems

It is natural to wonder whether additional resources actually give additional power. We show that this is the case (at least to a certain extent) for space and time. We first give a definitions of "well-behaved" functions.

**Definition 1** *A function $f : \mathbb{N} \to \mathbb{N}$ is* space constructible *if it is non-decreasing and there exists a Turing machine that on input $1^n$ outputs the binary representation of $f(n)$ using $O(f(n))$ space. Note that if $f$ is space constructible, then there exists a Turing machine that on input $1^n$ marks off exactly $f(n)$ cells on its work tapes (say, using a special symbol) without ever exceeding $O(f(n))$ space.*

For space bounds, it is often assumed that $f(n) \geq \log n$ as well. We will make this assumption throughout this class, unless explicitly stated otherwise. Note that non-trivial algorithms using sub-logarithmic space do exist; in particular, $\text{SPACE}(1)$ is a proper subset of $\text{SPACE}(\log \log n)$ (see [2, Lecture 4]). Nevertheless, sub-logarithmic space causes difficulties because there is not even enough space to store a counter indicating the position of the input-tape head.

[1] Note that the set of all Turing machines is countably infinite, and so $S$ is countable.

[2] An efficiently computable enumeration is obtained by letting $M_i$ denote the Turing machine represented by the binary representation of $i$.

**Definition 2** *A function $f : \mathbb{N} \to \mathbb{N}$ with $f(n) \geq n$ for all $n$ is* time constructible *if it is non-decreasing and there exists a Turing machine that on input $1^n$ outputs the binary representation of $f(n)$ in $O(f(n))$ steps. Note that if $f$ is time constructible, then there exists a Turing machine that on input $1^n$ runs for $O(f(n))$ steps and then halts.*

All functions you would "normally encounter" are space and time constructible; functions that aren't are specifically constructed counterexamples.

We first show that more space gives more power.

**Theorem 1 (Space hierarchy theorem)** *Let $G(n) \geq \log n$ be space constructible, and $g(n) = o(G(n))$. Then* $\text{SPACE}(g(n))$ *is a proper subset of* $\text{SPACE}(G(n))$.

**Proof** We show the existence of a language $L$ such that $L \in \text{SPACE}(G(n))$ but $L \notin \text{SPACE}(g(n))$. We define $L$ by describing a Turing machine $M_L$, using space $O(G(n))$, that decides it. $M_L$ does the following on input $w = (M, y)$ of length $|w| = n$:

1. Run $M(w)$ with at most $G(n)$ space and for at most $2^{2G(n)}$ steps (these bounds are imposed on $M$), using space at most $3 \cdot G(n)$.
2. If $M(w)$ accepts within the given time and space bounds, then reject. Otherwise, accept.

In step 1, we can use the fact that $G$ is space constructible to mark off exactly $G(n)$ tape cells for $M$ to use. We can similarly mark off an additional $2G(n)$ cells to use as a counter for checking the number of steps $M$ makes, and one last set of $G(n)$ cells to use for any remaining computation. By construction, $M_L$ uses space $\tilde{G}(n) = 4 \cdot G(n)$.

We need to show that no machine using space $O(g(n))$ can decide $L$. Assume the contrary. Then there exists a machine $M'_L$ deciding $L$ and using space $\tilde{g}(n) = O(g(n))$. Choose $k$ large enough so that $\tilde{g}(k) < G(k)$, so that[3] $M'_L$ makes fewer than $2^{G(k)}$ steps on inputs of length $k$, and so that[4] the simulation of $M'_L$ on inputs of length $k$ can be performed in $G(k)$ space. Consider the input $w = (M'_L, 1^k)$. If we run $M_L(w)$ then (1) $M_L$ has enough time and space to simulate the entire execution of $M'_L(w)$, and thus (2) $M_L(w)$ outputs the opposite of whatever $M'_L(w)$ outputs. We conclude that $M_L$ and $M'_L$ do not decide the same language. ■

We have a completely analogous time hierarchy theorem, though the result is quantitatively (slightly) weaker.

**Theorem 2 (Time hierarchy theorem)** *Let $G$ be time constructible. If $g(n) \log g(n) = o(G(n))$, then* $\text{TIME}(g(n))$ *is a proper subset of* $\text{TIME}(G(n))$.

**Proof** The high-level structure of the proof is the same as in the proof of the previous theorem. We define $L$ by giving a Turing machine $M_L$, using time $O(G(n))$, that decides it. $M_L$ does the following on input $w = (M, y)$ of length $|w| = n$:

1. Run $M(w)$ using at most $c \cdot G(n)$ steps for some fixed constant $c$ (see below).
2. If $M(w)$ accepts within the given time bound, then reject. Otherwise, accept.

[3] This condition is achievable because $M'_L$ runs in time at most $O(n2^{O(g(n))})$ (something we will show later in the course), which is asymptotically smaller than $2^{2G(n)}$.

[4] This condition is achievable because universal simulation with constant space overhead is possible.

We can implement step 1 using the fact that $G$ is time constructible: in alternating steps, simulate $M(w)$ and run a Turing machine that is guaranteed to stop within $O(G(n))$ steps; halt the entire computation once the latter machine halts. We thus have that $M_L$ runs in time $O(G(n))$.

We need to show that no machine using time $O(g(n))$ can decide $L$. Assume the contrary. Then there exists a machine $M'_L$ deciding $L$ in time $O(g(n))$. Consider an input of the form $w = (M'_L, 1^k)$. If we run $M_L(w)$ then, for $k$ large enough, $M_L$ has enough time to simulate the entire execution of $M'_L(w)$. (Here we use the fact that universal simulation is possible with logarithmic overhead.) But then, for $k$ large enough, $M_L(w)$ outputs the opposite of whatever $M'_L(w)$ outputs. We conclude that $M_L$ and $M'_L$ do not decide the same language. ■

The barrier to getting a tighter time hierarchy theorem is the logarithmic time overhead in universal simulation. If a better simulation were possible, we would obtain a tighter separation.

There is a non-deterministic time hierarchy as well; the details are more complicated because it is not possible to simply "flip" the output of a non-deterministic machine. (Do you see why?)

**Theorem 3 (Non-deterministic time hierarchy theorem)** *Let $g, G$ be time constructible. If $g(n+1) = o(G(n))$, then* $\text{NTIME}(g(n))$ *is a proper subset of* $\text{NTIME}(G(n))$.

**Proof** We sketch a proof different from the one in the book. We will also rely on the fact that non-deterministic universal simulation with only *constant* time overhead is possible.

Once again, we define a language $L$ by describing a machine that decides it. Consider the non-deterministic machine $M_L$ that on input $w = (M, 1^k, y)$ of length $|w| = n$, where $M$ is now interpreted as a non-deterministic Turing machine, does:

1. If $|y| < G(|M| + k)$ then run $M(M, 1^k, y0)$ and $M(M, 1^k, y1)$ for at most $G(n)$ steps (each), and accept iff they both accept.

2. If $|y| \geq G(|M|+k)$ then accept iff $M(M, 1^k, \varepsilon)$ rejects when using non-deterministic choices $y$. (Here $\varepsilon$ denotes the empty string.) Note that if $M$ does not halt on this computation path (e.g., because $y$ is not long enough), then $M_L$ rejects.

By what we have said before regarding universal simulation of non-deterministic Turing machines, and using the conditions of the theorem, $M_L$ runs in time $O(G(n))$.

Say there exists a non-deterministic machine $M'_L$ running in time $\tilde{g}(n) = O(g(n))$ and deciding $L$. Consider an input of the form $w = (M'_L, 1^k, \varepsilon)$ for $k$ sufficiently large. We have

$$\begin{array}{rcll} w \in L & \Leftrightarrow & M'_L(M'_L, 1^k, 0) = M'_L(M'_L, 1^k, 1) = 1 & \text{Definition of } M_L \\ & \Leftrightarrow & (M'_L, 1^k, 0), (M'_L, 1^k, 1) \in L & M'_L \text{ decides } L \\ & \Leftrightarrow & M'_L(M'_L, 1^k, 00) = M'_L(M'_L, 1^k, 01) = 1 & \\ & & M'_L(M'_L, 1^k, 10) = M'_L(M'_L, 1^k, 11) = 1 & \text{Definition of } M_L \end{array}$$

Let $t$ be the smallest integer with $t \geq G(|M'_L| + k)$. Continuing the above line of reasoning we get

$$\begin{array}{rcll} w \in L & \Leftrightarrow & \forall y \in \{0,1\}^t : M'_L(M'_L, 1^k, y) = 1 & \text{As above...} \\ & \Leftrightarrow & \forall y \in \{0,1\}^t : (M'_L, 1^k, y) \in L & M'_L \text{ decides } L \\ & \Leftrightarrow & \forall y \in \{0,1\}^t : M'_L(M'_L, 1^k, \varepsilon) \text{ rejects} & \\ & & \qquad \text{on computation path } y & \text{Definition of } M_L \\ & \Leftrightarrow & M'_L(M'_L, 1^k, \varepsilon) = 0 & \text{Definition of non-determinism} \\ & \Leftrightarrow & w \notin L & M'_L \text{ decides } L \end{array}$$

This is a contradiction, so we conclude that no such $M'_L$ can exist. ■

## Bibliographic Notes

The proof of the non-deterministic time hierarchy theorem given here is due to [1].

## References

[1] L. Fortnow and R. Santhanam. Robust Simulations and Significant Separations. *ICALP (1)*, 2011. Available at `http://arxiv.org/abs/1012.2034`.

[2] O. Goldreich. Introduction to Complexity Theory (July 31, 1999).

**Notes on Complexity Theory** Last updated: September 14, 2011

Lecture 5

*Jonathan Katz*

# 1 Diagonalization, Continued

## 1.1 Ladner's Theorem

We know that there exist $\mathcal{NP}$-complete languages. Assuming $\mathcal{P} \neq \mathcal{NP}$, any $\mathcal{NP}$-complete language lies in $\mathcal{NP} \setminus \mathcal{P}$. Are there languages that are neither in $\mathcal{P}$ nor $\mathcal{NP}$-complete? Ladner's theorem tells us that there are.

As some intuition for Ladner's theorem, take some language $L \in \mathcal{NP} \setminus \mathcal{P}$. Using padding, we will make $L$ "easy enough" so that it can't be $\mathcal{NP}$-complete, while keeping it "hard enough" so it is not in $\mathcal{P}$ either. Say the best algorithm for deciding $L$ runs in time $n^{\log n}$ for concreteness. (The same argument, though messier, works as long as the best algorithm deciding $L$ requires super-polynomial time.) Define

$$L' = \{(x, y) \mid x \in L \text{ and } |x| + |y| = |x|^{\log\log|x|}\}.$$

If $L' \in \mathcal{P}$, then $L$ would be decidable in time $n^{O(\log\log n)}$, a contradiction. On the other hand, $L'$ *is* decidable in time $N^{\log N}$ where $N$ is such that $N^{\log\log N} = n$ (the input length). We have $N = n^{o(1)}$, and so $L'$ is decidable in time $n^{o(\log\log(n^{o(1)}))}$. If $L$ were Karp-reducible to $L'$, then $L$ would be solvable in time $n^{o(\log n)}$, a contradiction. The main challenge in making the above formal is that it is hard to pin down the "best" algorithm for deciding some language $L$, or that algorithm's exact running time.

**Theorem 1** *Assuming $\mathcal{P} \neq \mathcal{NP}$, there exists a language $A \in \mathcal{NP} \setminus \mathcal{P}$ which is not $\mathcal{NP}$-complete.*

**Note:** We did not cover the proof of Ladner's theorem in class, but one is included here for completeness.

**Proof** The high-level intuition behind the proof is that we construct $A$ by taking an $\mathcal{NP}$-complete language and "blowing holes" in it in such a way that the language is no longer $\mathcal{NP}$-complete yet not in $\mathcal{P}$ either. The specific details are quite involved.

Let $M_1, \ldots$ denote an enumeration of all polynomial-time Turing machines with boolean output; formally, this can be achieved by considering an enumeration[1] of $\mathcal{M} \times \mathbb{Z}$ (where $\mathcal{M}$ is the set of Turing machines), and defining $M_i$ as follows: if the $i^{\text{th}}$ item in this enumeration is $(M, j)$, then $M_i(x)$ runs $M(x)$ for at most $|x|^j$ steps. We remark that $M_1, \ldots$ also gives an enumeration of languages in $\mathcal{P}$ (with languages appearing multiple times). In a similar way, let $F_1, \ldots$ denote an enumeration of polynomial-time Turing machines without the restriction of their output length. Note that this gives an enumeration of functions computable in polynomial time.

Define language $A$ as follows:

$$A = \{x \mid x \in \mathsf{SAT} \wedge f(|x|) \text{ is even}\},$$

[1] Since both $\mathcal{M}$ and $\mathbb{Z}$ are countable, it follows that $\mathcal{M} \times \mathbb{Z}$ is countable.

for some function $f$ that remains to be defined. Note that as long as we ensure that $f$ is computable in polynomial time, then $A \in \mathcal{NP}$. We define $f$ by a polynomial-time Turing machine $M_f$ that computes it. Let $M_{\text{SAT}}$ be a machine that decides SAT (not in polynomial time, of course...), and let $f(0) = f(1) = 2$. On input $1^n$ (with $n > 1$), $M_f$ proceeds in two stages, each lasting for exactly $n$ steps:

1. During the first stage, $M_f$ computes $f(0), f(1), \ldots$ until it runs out of time. Suppose the last value of $f$ it was able to compute was $f(x) = k$. The output of $M_f$ will be either $k$ or $k+1$, to be determined by the next stage.

2. Then:

   - If $k = 2i$ is even, then $M_f$ tries to find a $z \in \{0,1\}^*$ such that $M_i(z)$ outputs the "wrong" answer as to whether $z \in A$. (That is, $M_f$ tries to find $z$ such that either $z \in A$ but $M_i(z) = 0$, or the opposite.) This is done by computing $M_i(z), M_{\text{SAT}}(z)$, and $f(|z|)$ for all strings $z$ in lexicographic order. If such a string is found within the allotted time, the output of $M_f$ is $k+1$. Otherwise, the output of $M_f$ is $k$.
   - If $k = 2i-1$ is odd, then $M_f$ tries to find a string $z$ such that $F_i(z)$ is an incorrect Karp reduction from SAT to $A$. (That is, $M_f$ tries to find a $z$ such that either $z \in \text{SAT}$ but $F_i(z) \notin A$, or the opposite.) This is done by computing $F_i(z)$, $M_{\text{SAT}}(z)$, $M_{\text{SAT}}(F_i(z))$, and $f(|F_i(z)|)$. If such a string is found within the allotted time, then the output of $M_f$ is $k+1$; otherwise, the output is $k$.

By its definition, $M_f$ runs in polynomial time. Note also that $f(n+1) \geq f(n)$ for all $n$.

We claim that $A \notin \mathcal{P}$. Suppose the contrary. Then $A$ is decided by some $M_i$. In this case, however, the second stage of $M_f$ with $k = 2i$ will never find a $z$ satisfying the desired property, and so $f$ is eventually a constant function and in particular $f(n)$ is odd for only finitely-many $n$. But this implies that $A$ and SAT coincide except for finitely-many strings. But this implies that $\text{SAT} \in \mathcal{P}$, a contradiction to our assumption that $\mathcal{P} \neq \mathcal{NP}$.

Similarly, we claim that $A$ is not $\mathcal{NP}$-complete. Suppose the contrary. Then there is a polynomial-time function $F_i$ which gives a Karp reduction from SAT to $A$. Now $f(n)$ will be even for only finitely-many $n$, implying that $A$ is a finite language. But then $A \in \mathcal{P}$, a contradiction to our assumption that $\mathcal{P} \neq \mathcal{NP}$. ■

As an addendum to the theorem, we note that (assuming $\mathcal{P} \neq \mathcal{NP}$, of course) there are no "natural" languages provably in $\mathcal{NP} \setminus \mathcal{P}$ that are not $\mathcal{NP}$-complete. However, there are a number of languages conjectured to fall in this category, including graph isomorphism and essentially all languages derived from cryptographic assumptions (e.g., factoring).

## 1.2 Relativizing the $\mathcal{P}$ vs. $\mathcal{NP}$ Question

We conclude by showing some limitations of the diagonalization technique. (Interestingly, these limitations are proven by diagonalization!). Informally, diagonalization relies on the following properties of Turing machines:

1. The fact that Turing machines can be represented by finite strings.

2. The fact that one Turing machine can simulate another (without much overhead).

Any proof that relies only on these facts is essentially treating Turing machines as black boxes (namely, looking only at their input/output), without caring much about the details of how they work. In that case, the proof should apply just as well to *oracle* Turing machines.

An oracle is just a function $\mathcal{O} : \{0,1\}^* \to \{0,1\}$, and of course for any $\mathcal{O}$ we have a corresponding language $L$. Fixing $\mathcal{O}$, an oracle Turing machine $M^{\mathcal{O}}$ is given the ability to make "queries" to $\mathcal{O}$ and obtain the result in a single time step.[2] (We have already seen this notation when we talked about Cook-Turing reductions.) Fixing some $\mathcal{O}$, we say $L \in \mathcal{P}^{\mathcal{O}}$ if there exists a polynomial-time Turing machine $M$ such that $x \in L \Leftrightarrow M^{\mathcal{O}}(x) = 1$. Similarly, $L \in \mathcal{NP}^{\mathcal{O}}$ if there exists a polynomial-time Turing machine $M$ such that $x \in L \Leftrightarrow \exists w : M^{\mathcal{O}}(x, w) = 1$. More generally, for any class $\mathcal{C}$ defined in terms of Turing machines deciding languages in that class, we can define the class $\mathcal{C}^{\mathcal{O}}$ in the natural way.

Given a result about two complexity classes $\mathcal{C}_1, \mathcal{C}_2$, we can ask whether that same result holds about $\mathcal{C}_1^{\mathcal{O}}, \mathcal{C}_2^{\mathcal{O}}$ for *any* oracles $\mathcal{O}$. If so, then the result *relativizes*. Any result proved via diagonalization, as defined above, relativizes. As examples: the result about universal simulation relativizes, as does the time-hierarchy theorem.

We now show that the $\mathcal{P}$ vs. $\mathcal{NP}$ question does *not* relativize. We demonstrate this by showing that there exists oracles $A, B$ such that

$$\mathcal{P}^A = \mathcal{NP}^A \text{ but } \mathcal{P}^B \neq \mathcal{NP}^B.$$

When this result was first demonstrated [3], it was taken as an indication of the difficulty of resolving the $\mathcal{P}$ vs. $\mathcal{NP}$ question using "standard techniques". It is important to note, however, that various non-relativizing results are known. As one important example, the proof that $\mathsf{SAT}$ is $\mathcal{NP}$-complete does not relativize. (This is not the best example, since $\mathsf{SAT}$ is a problem and not a class.) See [5, Lect. 26] and [2, 4, 6] for further discussion.

**An oracle $A$ for which $\mathcal{P}^A = \mathcal{NP}^A$.** Recall that $\mathsf{EXP} = \cup_c \text{TIME}(2^{n^c})$. Let $A$ be an $\mathsf{EXP}$-complete language. It is obvious that $\mathcal{P}^A \subseteq \mathcal{NP}^A$ for any $A$, so it remains to show that $\mathcal{NP}^A \subseteq \mathcal{P}^A$. We do this by showing that

$$\mathcal{NP}^A \subseteq \mathsf{EXP} \subseteq \mathcal{P}^A.$$

The second inclusion is immediate (just use a Karp reduction from any language $L \in \mathsf{EXP}$ to the $\mathsf{EXP}$-complete problem $A$), and so we have only to prove the first inclusion. This, too, is easy: Let $L \in \mathcal{NP}^A$ and let $M$ be a polynomial-time non-deterministic machine such that $M^A$ decides $L$. Then using a deterministic exponential-time machine $M'$ we simply try all possible non-deterministic choices for $M$, and whenever $M$ makes a query to $A$ we have $M'$ answer the query by itself.

**An oracle $B$ for which $P^B \neq \mathcal{NP}^B$.** This is a bit more interesting. We want to find an oracle $B$ such that $\mathcal{NP}^B \setminus \mathcal{P}^B$ is not empty. For any oracle (i.e., language) $B$, define language $L_B$ as follows:

$$L_B \stackrel{\text{def}}{=} \{1^n \mid B \cap \{0,1\}^n \neq \emptyset\}.$$

It is immediate that $L_B \in \mathcal{NP}^B$ for any $B$. (On input $1^n$, guess $x \in \{0,1\}^n$ and submit it to the oracle; output 1 iff the oracle returns 1.) As a "warm-up" to the desired result, we show:

[2] There are subtleties in dealing with space-bounded oracle machines. We only discuss time-bounded oracle machines here.

**Claim 2** *For any deterministic, polynomial-time oracle machine $M$, there exists a language $B$ such that $M^B$ does not decide $L_B$.*

**Proof** Given $M$ with polynomial running time $p(\cdot)$, we construct $B$ as follows: let $n$ be the smallest integer such that $2^n > p(n)$. Note that on input $1^n$, machine $M$ cannot query its oracle on all strings of length $n$. We exploit this by defining $B$ in the following way:

> Run $M(1^n)$ and answer "0" to all queries of $M$. Let $b$ be the output of $M$, and let $Q = \{q_1, \ldots\}$ denote all the queries of length exactly $n$ that were made by $M$. Take arbitrary $x \in \{0,1\}^n \setminus Q$ (we know such an $x$ exists, as discussed above). If $b = 0$, then put $x$ in $B$; if $b = 1$, then take $B$ to just be the empty set.

Now $M^B(1^n) = b$ (since $B$ returns 0 for every query made by $M(1^n)$), but this answer is incorrect by construction of $B$. ■

This claim is not enough to prove the desired result, since we need to reverse the order of quantifiers and show that there exists a language $B$ such that for *all* deterministic, polynomial-time $M$ we have that $M^B$ does not decide $L_B$. We do this by extending the above argument. Consider an enumeration $M_1, \ldots$ of all deterministic, polynomial-time machines with running times $p_1, \ldots$. We will build $B$ inductively. Let $B_0 = \emptyset$ and $n_0 = 1$. Then in the $i^{\text{th}}$ iteration do the following:

- Let $n_i$ be the smallest integer such that $2^{n_i} > p_i(n_i)$ and also $n_i > p_j(n_j)$ for all $1 \leq j < i$.
- Run $M_i(1^{n_i})$ and respond to its queries according to $B_{i-1}$. Let $Q = \{q_1, \ldots\}$ be the queries of length exactly $n_i$ that were made by $M_i$, and let $x \in \{0,1\}^{n_i} \setminus Q$ (again, we know such an $x$ exists). If $b = 0$ then set $B_i = B_{i-1} \cup \{x\}$; if $b = 1$ then set $B_i = B_{i-1}$ (and so $B_i$ does not contain any strings of length $n_i$).

Let $B = \bigcup_i B_i$. We claim that $B$ has the desired properties. Indeed, when we run $M_i(1^{n_i})$ with oracle access to $B_i$, we can see (following the reasoning in the previous proof) that $M_i$ will output the wrong answer (and thus $M_i^{B_i}$ does not decide $L_{B_i}$). But the output of $M_i(1^{n_i})$ with oracle access to $B$ is the same as the output of $M_i(1^{n_i})$ with oracle access to $B_i$, since all strings in $B \setminus B_i$ have length greater than $p_i(n_i)$ and so none of $M_i$'s queries (on input $1^{n_i}$) will be affected by using $B$ instead of $B_i$. It follows that $M_i^B$ does not decide $L_B$.

## 2 Space Complexity

Recall that for space complexity (in both the deterministic and non-deterministic cases) we measure the number of cells used on the *work tape* only. This allows us to talk meaningfully of sublinear-space algorithms, and algorithms whose output may be longer than the space used.

Note also that in the context of space complexity we may assume without loss of generality that machines have only a single work tape. This is so because we can perform universal simulation of a $k$-tape Turing machine on a Turing machine with just a single work tape, with only a constant factor blowup in the space complexity.

When we talk about non-deterministic space complexity we refer to our original notion of non-deterministic Turing machines, where there are two transition functions and at every step the machine makes a non-deterministic choice as to which one to apply. It turns out that, just as we did in the case of $\mathcal{NP}$, we can give a "certificate-based" definition of non-deterministic space

classes as well, though we need to be a little careful since the length of the certificate may exceed the space-bound of the machine. In particular, we imagine a (deterministic) machine with an input tape and work tape as usual, and also a special *certificate tape*. When measuring the space used by this machine, we continue to look at the space on the work tape only. The certificate tape (like the input tape) is a *read-only* tape; moreover (and unlike the input tape), we restrict the Turing machine so that it may only move its certificate-tape head from left to right (or stay in place). This gives a definition equivalent to the definition in terms of non-deterministic Turing machines; in particular:

**Claim 3** *$L \in \text{NSPACE}(s(n))$ iff there exists a (deterministic) Turing machine with a special "read-once" certificate tape as described above that uses space $O(s(n))$, and such that $x \in L$ iff there exists a certificate $w$ such that $M(x, w) = 1$.*

If the certificate-tape head is allowed to move back-and-forth across its tape, this gives the machine significantly more power; in fact, if we consider log-space machines that move freely on their certificate tape we get the class $\mathcal{NP}$! See [5, Chap. 5] for further discussion regarding the above.

## Bibliographic Notes

The intuition before the proof of Ladner's theorem is due to Russell Impagliazzo (personal communication).

## References


[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[2] E. Allender. Oracles versus Proof Techniques that Do Not Relativize. *SIGAL Intl. Symposium on Algorithms*, pp. 39–52, 1990.

[3] T. Baker, J. Gill, and R. Solovay. Relativizations of the $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ Question. *SIAM J. Computing* 4(4): 431–442, 1975.

[4] L. Fortnow. The Role of Relativization in Complexity Theory. *Bulletin of the European Association for Theoretical Computer Science*, 52: 229–243, 1994.

[5] O. Goldreich. Introduction to Complexity Theory (July 31, 1999).

[6] J. Hartmanis, R. Chang, S. Chari, D. Ranjan, and P. Rohatgi. Relativization: A Revisionist Retrospective. *Current Trends in Theoretical Computer Science*, 1993. Available from `http://www.cs.umbc.edu/ chang/papers/revisionist`.

**Notes on Complexity Theory** Last updated: September, 2011

# Lecture 6

*Jonathan Katz*

## 1 Space Complexity

We define some of the important space-complexity classes we will study:

**Definition 1**

$$\begin{aligned} \mathsf{PSPACE} &\stackrel{\text{def}}{=} \bigcup_c \text{SPACE}(n^c) \\ \mathsf{NPSPACE} &\stackrel{\text{def}}{=} \bigcup_c \text{NSPACE}(n^c) \\ \mathsf{L} &\stackrel{\text{def}}{=} \text{SPACE}(\log n) \\ \mathsf{NL} &\stackrel{\text{def}}{=} \text{NSPACE}(\log n). \end{aligned}$$

We have seen that $\text{TIME}(t(n)) \subseteq \text{NTIME}(t(n)) \subseteq \text{SPACE}(t(n))$. What can we say in the other direction? To study this we look at *configurations* of a Turing machine, where a configuration consists of all the information necessary to describe the Turing machine at some instant in time. We have the following easy claim.

**Claim 1** *Let $M$ be a (deterministic or non-deterministic) machine using space $s(n)$. The number of configurations $\mathcal{C}_M(n)$ of $M$ on any fixed input of length $n$ is bounded by:*

$$\mathcal{C}_M(n) \leq |Q_M| \cdot n \cdot s(n) \cdot |\Sigma_M|^{s(n)}, \tag{1}$$

*where $Q_M$ are the states of $M$ and $\Sigma_M$ is the alphabet of $M$. In particular, when $s(n) \geq \log n$ we have $\mathcal{C}_M(n) = 2^{\Theta(s(n))}$.*

**Proof** The first term in Eq. (1) comes from the number of states, the second from the possible positions of the input head, the third from the possible positions of the work-tape head, and the last from the possible values stored on the work tape. (Note that since the input is fixed and the input tape is read-only, we do not need to consider all possible length-$n$ strings that can be written on the input tape.) ■

We can use this to obtain the following relationship between space and time:

**Theorem 2** *Let $s(n)$ be space constructible with $s(n) \geq \log n$. Then $\text{SPACE}(s(n)) \subseteq \text{TIME}(2^{O(s(n))})$ and $\text{NSPACE}(s(n)) \subseteq \text{NTIME}(2^{O(s(n))})$.*

**Proof** Let $L \in \text{SPACE}(s(n))$, and let $M$ be a machine using space $O(s(n))$ and deciding $L$. Consider the computation of $M(x)$ for some input $x$ of length $n$. There are at most $\mathcal{C}_M(n) = 2^{\Theta(s(n))}$ configurations of $M$ on $x$, but if $M(x)$ ever repeats a configuration then it would cycle and never halt. Thus, the computation of $M(x)$ must terminate in at most $\mathcal{C}_M(n) = 2^{\Theta(s(n))}$ steps.

Let $L \in \text{NSPACE}(s(n))$. Then there is a non-deterministic Turing machine $M$ deciding $L$ and using space $O(s(n))$ on every computation path (i.e., regardless of the non-deterministic choices it makes). Consider a machine $M'$ that runs $M$ but only for at most $2^{O(s(n))} \geq \mathcal{C}_M(n)$ steps (and rejects if $M$ has not halted by that point); this can be done using a counter of length $O(s(n))$ and so $M'$ still uses $O(s(n))$ space. We claim that $M'$ still decides $L$. Clearly if $M(x) = 0$ then $M'(x) = 0$. If $M(x) = 1$, consider the *shortest* computation path on which $M(x)$ accepts. If this computation path uses more than $\mathcal{C}_M(|x|)$ steps, then some configuration of $M$ must repeat. But then there would be another sequence of non-deterministic choices that would result in a shorter accepting computation path, a contradiction. We conclude that $M(x)$ has an accepting computation path of length at most $\mathcal{C}_M(|x|)$, and so if $M(x)$ accepts then so does $M'(x)$. ■

The theorem above may seem to give a rather coarse bound for $\text{SPACE}(s(n))$, but intuitively it does appear that space is more powerful than time since space can be *re-used* while time cannot. In fact, it is known that $\text{TIME}(s(n))$ is a strict subset of $\text{SPACE}(s(n))$ (for space constructible $s(n) \geq n$), but we do not know much more than that. We conjecture that space is much more powerful than time; in particular, we believe:

**Conjecture 3** $\mathcal{P} \neq \mathsf{PSPACE}$.

Note that $\mathcal{P} = \mathsf{PSPACE}$ would, in particular, imply $\mathcal{P} = \mathcal{NP}$.

## 1.1 PSPACE and PSPACE-Completeness

As in our previous study of $\mathcal{NP}$, it is useful to identify those problems that capture the essence of PSPACE in that they are the "hardest" problems in that class. We can define a notion of PSPACE-completeness in a manner exactly analogous to $\mathcal{NP}$-completeness:

**Definition 2** *Language $L'$ is* PSPACE-hard *if for every $L \in \mathsf{PSPACE}$ it holds that $L \leq_p L'$. Language $L'$ is* PSPACE-complete *if $L' \in \mathsf{PSPACE}$ and $L'$ is* PSPACE*-hard.*

Note that if $L$ is PSPACE-complete and $L \in \mathcal{P}$, then $\mathcal{P} = \mathsf{PSPACE}$.

As usual, there is a "standard" (but unnatural) complete problem; in this case, the following language is PSPACE-complete:

$$L \stackrel{\text{def}}{=} \{(M, x, 1^s) \mid M(x) \text{ accepts using space at most } s\}.$$

For a more natural PSPACE-complete problem we turn to a variant of SAT. Specifically, we consider the language of *totally quantified boolean formulae* (denoted TQBF) which consists of quantified formulae of the form:

$$\exists x_1 \forall x_2 \cdots Q_n x_n \quad \phi(x_1, \ldots, x_n),$$

where $\phi$ is a boolean formula, and $Q_i = \exists$ and $Q_{i+1} = \forall$ alternate (it does not matter which is first). An expression of the above form is in TQBF if it is true: that is, if it is the case that "for all $x_1 \in \{0,1\}$, there exists an $x_2 \in \{0,1\}$ ...such that $\phi(x_1, \ldots, x_n)$ evaluates to true". More generally, if $M$ is a polynomial-time Turing machine then any statement of the form

$$\exists x_1 \in \{0,1\}^{\mathsf{poly}(n)} \forall x_2 \in \{0,1\}^{\mathsf{poly}(n)} \cdots Q_n x_n \in \{0,1\}^{\mathsf{poly}(n)} \quad M(x_1, \ldots, x_n) = 1,$$

can be converted to a totally quantified boolean formula.

**Theorem 4** TQBF *is* PSPACE*-complete.*

**Proof** It is not too difficult to see that TQBF $\in$ PSPACE, since in polynomial space we can try all settings of all the variables and keep track of whether the quantified expression is true.

We next show that TQBF is PSPACE-complete. Given a PSPACE machine $M$ deciding some language $L$, we reduce the computation of $M(x)$ to a totally quantified boolean formula. Since $M$ uses space $n^k$ for some constant $k$, we may encode configurations of $M$ on some input $x$ of length $n$ using $O(n^k)$ bits. Given an input $x$, we construct (in polynomial time) a sequence of totally quantified boolean formulae $\psi_0(a,b),\ldots$, where $\psi_i(a,b)$ is true iff there is a path (i.e., sequence of steps of $M$) of length at most $2^i$ from configuration $a$ to configuration $b$. We then output $\psi_{n^k}(\mathsf{start},\mathsf{accept})$, where start denotes the initial configuration of $M(x)$, and accept is the (unique) accepting configuration of $M$. Note that $M(x)=1$ iff $\psi_{n^k}(\mathsf{start},\mathsf{accept})$ is true (using Theorem 2).

We need now to construct the $\{\psi_i\}$. Constructing $\psi_0$ is easy: to evaluate $\psi_0(a,b)$ we simply test whether $a=b$ or whether configuration $b$ follows from configuration $a$ in one step. (Recalling the proof that SAT is $\mathcal{NP}$-complete, it is clear that this can be expressed as a polynomial-size boolean formula.) Now, given $\psi_i$ we construct $\psi_{i+1}$. The "obvious" way of doing this would be to define $\psi_{i+1}(a,b)$ as:

$$\exists c: \quad \psi_i(a,c) \wedge \psi_i(c,b).$$

While this is correct, it would result in a formula $\psi_{n^k}$ of *exponential* size! (To see this, note that the size of $\psi_{i+1}$ is roughly double the size of $\psi_i$.) Also, we have not made use of any universal quantifiers. Instead, we proceed a bit more cleverly and "encode" $\psi_i(a,c) \wedge \psi_i(c,b)$ in a smaller expression. Define $\psi_{i+1}(a,b)$ as:

$$\exists c \forall x,y: \quad \Big( \big((x,y)=(a,c)\big) \vee \big((x,y)=(c,b)\big) \Big) \Rightarrow \psi_i(x,y);$$

it is easy to see that constructing $\psi_{i+1}$ from $\psi_i$ can be done efficiently. (A technical point: although the quantifiers of $\psi_i$ are "buried" inside the expression for $\psi_{i_1}$, it is easy to see that the quantifiers of $\psi_i$ can be migrated to the front without changing the truth of the overall expression.) The key point is that whereas previously the size of $\psi_{i+1}$ was double the size of $\psi_i$, here the size of $\psi_{i+1}$ is only an $O(n^k)$ *additive* factor larger than $\psi_i$ and so the size of $\psi_{n^k}$ will be polynomial. ■

A very observant reader may note that everything about the above proof applies to NPSPACE as well, and so the above implies that TQBF is NPSPACE-complete and PSPACE = NPSPACE! We will see another proof of this later.

**Playing games.** The class TQBF captures the existence of a winning strategy for a certain player in bounded-length perfect-information games (that can be played in polynomial time). Specifically, consider a two-player game where players alternate making moves for a total of $n$ turns. Given moves $p_1,\ldots,p_n$ by the players, let $M(p_1,\ldots,p_n)=1$ iff player 1 has won the game. (Note that $M$ can also encode a check that every player made a legal move; a player loses if it makes the first non-legal move.) Then player 1 has a winning strategy in the game iff there exists a move $p_1$ that player 1 can make such that for every possible response $p_2$ of player 2 there is a move $p_3$ for player 1, $\ldots$, such that $M(p_1,\ldots,p_n)=1$. Many popular games have been proven to be PSPACE-complete. (For this to be made formal, the game must be allowed to grow without bound.)

## 2 Configuration Graphs and the Reachability Method

In this section, we will see several applications of the so-called *reachability method*. The basic idea is that we can view the computation of a non-deterministic machine $M$ on input $x$ as a directed graph (the *configuration graph* of $M(x)$) with vertices corresponding to configurations of $M(x)$ and an edge from vertex $i$ to vertex $j$ if there is a one-step transition from configuration $i$ to configuration $j$. Each vertex in this graph has out-degree at most 2. (We can construct such a graph for deterministic machines as well. In that case the graph has out-degree 1 and is less interesting.) If $M$ uses space $s(n) \geq \log n$, then vertices in the configuration graph of $M(x)$ can be represented using $O(s(n))$ bits.[1] If we assume, without loss of generality, that $M$ has only a single accepting state, then the question of whether $M(x)$ accepts is equivalent to the question of whether there is a path from the initial configuration of $M(x)$ to the accepting configuration. We refer to this as the *reachability* problem in the graph of interest.

### 2.1 NL and NL-Completeness

We further explore the connection between graphs and non-deterministic computation by looking at the class NL. As usual, we can try to understand NL by looking at the "hardest" problems in that class. Here, however, we need to use a more refined notion of reducibility:

**Definition 3** *$L$ is* log-space reducible *to $L'$ if there is a function $f$ computable in space $O(\log n)$ such that $x \in L \Leftrightarrow f(x) \in L'$.*

Note that if $L$ is log-space reducible to $L'$ then $L$ is Karp-reducible to $L'$ (by Theorem 2); in general, however, we don't know whether the converse is true.

**Definition 4** *$L$ is NL-complete if (1) $L \in$ NL, and (2) for all $L' \in$ NL it holds that $L'$ is log-space reducible to $L$.*

Log-space reducibility is needed[2] for the following result:

**Lemma 5** *If $L$ is log-space reducible to $L'$ and $L' \in$ L (resp., $L' \in$ NL) then $L \in$ L (resp., $L \in$ NL).*

**Proof** Let $f$ be a function computable in log space such that $x \in L$ iff $f(x) \in L'$. The "trivial" way of trying to prove this lemma (namely, on input $x$ computing $f(x)$ and then determining whether $f(x) \in L'$) does *not* work: the problem is that $|f(x)|$ may potentially have size $\omega(\log |x|)$ in which case this trivial algorithm uses superlogarithmic space. Instead, we need to be a bit more clever. The basic idea is as follows: instead of computing $f(x)$, we simply compute the $i^{\text{th}}$ bit of $f(x)$ whenever we need it. In this way, although we are wasting time (in re-computing $f(x)$ multiple times), we never uses more than logarithmic space. ■

[1] Note that $x$ is fixed, so need not be stored as part of a configuration. Whenever we construct an algorithm $M'$ that operates on the configuration graph of $M(x)$, the input $x$ itself will be written on the input tape of $M'$ and so $M'$ will not be "charged" for storing $x$.

[2] In general, to study completeness in some class $\mathcal{C}$ we need to use a notion of reducibility computable within $\mathcal{C}$.

**Notes on Complexity Theory** Last updated: September, 2011

# Lecture 7

*Jonathan Katz*

## 1 Configuration Graphs and the Reachability Method

### 1.1 NL and NL-Completeness

Coming back to problems on graphs, consider the problem of *directed connectivity* (denoted CONN). Here we are given a directed graph on $n$-vertices (say, specified by an adjacency matrix) and two vertices $s$ and $t$, and want to determine whether there is a directed path from $s$ to $t$.

**Theorem 1** CONN *is* NL*-complete.*

**Proof** To see that it is in NL, we need to show a non-deterministic algorithm using log-space that never accepts if there is no path from $s$ to $t$, and that sometimes accepts if there is a path from $s$ to $t$. The following simple algorithm achieves this:

if $s = t$ accept
set $v_{\text{current}} := s$
for $i = 1$ to $n$:
    guess a vertex $v_{\text{next}}$
    if there is no edge from $v_{\text{current}}$ to $v_{\text{next}}$, reject
    if $v_{\text{next}} = t$, accept
    $v_{\text{current}} := v_{\text{next}}$
if $i = n$ and no decision has yet been made, reject

The above algorithm needs to store $i$ (using $\log n$ bits), and at most the labels of two vertices $v_{\text{current}}$ and $v_{\text{next}}$ (using $O(\log n)$ bits).

To see that CONN is NL-complete, assume $L \in \mathsf{NL}$ and let $M_L$ be a non-deterministic log-space machine deciding $L$. Our log-space reduction from $L$ to CONN takes input $x \in \{0,1\}^n$ and outputs a graph (represented as an adjacency matrix) in which the vertices represent configurations of $M_L(x)$ and edges represent allowed transitions. (It also outputs $s = \mathsf{start}$ and $t = \mathsf{accept}$, where these are the starting and accepting configurations of $M(x)$, respectively.) Each configuration can be represented using $O(\log n)$ bits, and the adjacency matrix (which has size $O(n^2)$) can be generated in log-space as follows:

For each configuration $i$:
    for each configuration $j$:
        Output 1 if there is a legal transition from $i$ to $j$, and 0 otherwise
        (if $i$ or $j$ is not a legal state, simply output 0)
Output start, accept

The algorithm requires $O(\log n)$ space for $i$ and $j$, and to check for a legal transition. ■

We can now easily prove the following:

**Theorem 2** *For $s(n) \geq \log n$ a space-constructible function,* $\text{NSPACE}(s(n)) \subseteq \text{TIME}(2^{O(s(n))})$.

**Proof** We can solve CONN in linear time (in the number of vertices) using breadth-first search, and so $\text{CONN} \in \mathcal{P}$. By the previous theorem, this means $\mathsf{NL} \subseteq \mathcal{P}$ (a special case of the theorem).

In the general case, let $L \in \text{NSPACE}(s(n))$ and let $M$ be a non-deterministic machine deciding $L$ using $O(s(n))$ space. We construct a deterministic machine running in time $2^{O(s(n))}$ that decides $L$ by solving the reachability problem on the configuration graph of $M(x)$, specifically, by determining whether the accepting state of $M(x)$ (which we may assume unique without loss of generality) is reachable from the start state of $M(x)$. This problem can be solved in time linear in the number of vertices in the configuration graph. ■

**Corollary 3** $\mathsf{NL} \subseteq \mathcal{P}$.

Summarizing what we know,

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathcal{P} \subseteq \mathcal{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP}.$$

By the hierarchy theorems (and Savitch's theorem, below) we know NL is a strict subset of PSPACE, and $\mathcal{P}$ is a strict subset of EXP. But we cannot prove that any of the inclusions above is strict.

## 1.2 Savitch's Theorem

In the case of time complexity, we believe that non-determinism provides a huge (exponential?) benefit. For space complexity, this is surprisingly not the case:

**Theorem 4 (Savitch's Theorem)** *Let $s(n) \geq \log n$ be a space-constructible function. Then* $\text{NSPACE}(s(n)) \subseteq \text{SPACE}(s(n)^2)$.

**Proof** This is another application of the reachability method. Let $L \in \text{NSPACE}(s(n))$. Then there is a non-deterministic machine $M$ deciding $L$ and using space $O(s(n))$. Consider the configuration graph $G_M$ of $M(x)$ for some input $x$ of length $n$, and recall that (1) vertices in $G_M$ can be represented using $O(s(n))$ bits, and (2) existence of an edge in $G_M$ from some vertex $i$ to another vertex $j$ can be determined using $O(s(n))$ space.

We may assume without loss of generality that $M$ has a single accepting configuration (e.g., $M$ erases its work tape and moves both heads to the left-most cell of their tapes before accepting). $M(x)$ accepts iff there is a directed path in $G_M$ from the starting configuration of $M(x)$ (called start) to the accepting configuration of $M(x)$ (called accept). There are $V = 2^{O(s(n))}$ vertices in $G_M$, and the crux of the proof comes down to showing that reachability on a general $V$-node graph can be decided in deterministic space $O(\log^2 V)$.

Turning to that general problem, we define a (deterministic) recursive algorithm Path with the property that $\mathsf{Path}(a, b, i)$ outputs 1 iff there is a path of length at most $2^i$ from $a$ to $b$ in a given graph $G$; the algorithm only needs the ability to enumerate the vertices of $G$ and to test for directed edges between any two vertices $i, j$ in this graph. The algorithm proceeds as follows:

$\underline{\mathsf{Path}(a,b,i)}$:

- If $i = 0$, output "yes" if $a = b$ or if there is an edge from $a$ to $b$. Otherwise, output "no".
- If $i > 0$ then for each vertex $v$:
  - If $\mathsf{Path}(a, v, i-1)$ and $\mathsf{Path}(v, b, i-1)$, return "yes" (and halt).
- Return "no".

Let $S(i)$ denote the space used by $\mathsf{Path}(a, b, i)$. We have $S(i) = O(\log V) + S(i-1)$ and $S(0) = O(\log V)$. This solves to $S(i) = O(i \cdot \log V)$.

We solve our original problem by calling $\mathsf{Path}(\mathsf{start}, \mathsf{accept}, \log V)$ using the graph $G_M$, where $G_M$ has $V = 2^{O(s(n))}$ vertices. This uses space $O(\log^2 V) = O(s(n)^2)$, as claimed. ■

We have seen the next result before, but it also follows as a corollary of the above:

**Corollary 5** $\mathsf{PSPACE} = \mathsf{NPSPACE}$.

Is there a better algorithm for directed connectivity than what Savitch's theorem implies? Note that the algorithm implies by Savitch's theorem uses polylogarithmic space but superpolynomial time (specifically, time $2^{O(\log^2 n)}$). On the other hand, we have *linear*-time algorithms for solving directed connectivity but these require linear space. The conjecture is that $\mathsf{L} \neq \mathsf{NL}$, in which case directed connectivity does not have a log-space algorithm, though perhaps it would not be earth-shattering if this conjecture were proven to be false. Even if $\mathsf{L} \neq \mathsf{NL}$, we could still hope for an algorithm solving directed connectivity in $O(\log^2 n)$ space and polynomial time.

### 1.3 The Immerman-Szelepcsényi Theorem

As yet another example of the reachability method, we will show the somewhat surprising result that non-deterministic space is closed under complementation.

**Theorem 6** $\overline{\text{CONN}} \in \mathsf{NL}$.

**Proof** Recall that

$$\overline{\text{CONN}} \stackrel{\text{def}}{=} \left\{ (G,s,t) : \begin{array}{c} G \text{ is a directed graph in which} \\ \text{there is } \textit{no} \text{ path from vertex } s \text{ to vertex } t \end{array} \right\}.$$

Let $V$ denote the number of vertices in the graph $G$ under consideration. We show that $\overline{\text{CONN}} \in \mathsf{NL}$ using the certificate-based definition of non-deterministic space complexity. Thus, we will show a (deterministic) machine $M$ using space $O(\log V)$ such that the following holds: if there is *no* directed path in $G$ from $s$ to $t$, then there exists a certificate that will make $M(G,s,t)$ accept. On the other hand if there *is* a directed path in $G$ from $s$ to $t$, then no certificate can make $M(G,s,t)$ accept. Note the difficulty here: it is easy to give a proof (verifiable in space $O(\log V)$) proving the *existence* of a path — the certificate is just the path itself. But how does one construct a proof (verifiable in space $O(\log V)$) proving *non-existence* of a path?

We build our certificate from a number of 'primitive' certificates. Fix $(G,s,t)$, let $C_i$ denote the set of vertices reachable from $s$ in at most $i$ steps, and let $c_i = |C_i|$. We want to prove that $t \notin C_V$. We already know that we can give a certificate $\mathsf{Path}_i(s,v)$ (verifiable in logarithmic space) proving that there is a path of length at most $i$ from $s$ to $v$. Now consider the following:

- Assuming $c_{i-1}$ is known, we can construct a certificate $\mathsf{noPath}_i(s,v)$ (verifiable in logarithmic space) proving that there is no path of length at most $i$ from $s$ to $v$. (I.e., $v \notin C_i$.) The certificate is

  $$v_1, \mathsf{Path}_{i-1}(s,v_1), \ldots, v_{c_{i-1}}, \mathsf{Path}_{i-1}(s,v_{c_{i-1}}),$$

  for $v_1, \ldots, v_{c_{i-1}} \in C_{i-1}$ in ascending order. This certificate is verified by checking that (1) the number of vertices listed is exactly $c_{i-1}$, (2) the vertices are listed in ascending order, (3) none of the listed vertices is equal to $v$ or is a neighbor of $v$, and (4) each certificate $\mathsf{Path}_{i-1}(s,v_j)$ is correct. This can all be done in $O(\log V)$ space with read-once access to the certificate.

- Assuming $c_{i-1}$ is known, we can construct a certificate $\mathsf{Size}_i(k)$ (verifiable in logarithmic space) proving that $c_i = k$. The certificate is simply the list of all the vertices $v_1, \ldots$ in $G$ (in ascending order), where each vertex is followed by either $\mathsf{Path}_i(s,v)$ or $\mathsf{noPath}_i(s,v)$, depending on whether $v \in C_i$ or not. This certificate can be verified by checking that (1) all vertices are in the list, in ascending order, (2) each certificate $\mathsf{Path}_i(s,v)$ or $\mathsf{noPath}_i(s,v)$ is correct, and (3) the number of vertices in $C_i$ is exactly $k$.

(Note that the verifier only needs the ability to detect edges between two given vertices of $G$.) Observing that the size of $C_0 = \{s\}$ is already known, the certificate that $(G,s,t) \in \overline{\text{CONN}}$ is just

$$\mathsf{Size}_1(c_1), \mathsf{Size}_2(c_2), \ldots, \mathsf{Size}_{V-1}(c_{V-1}), \mathsf{noPath}_V(s,t).$$

Each certificate $\mathsf{Size}_i(c_i)$ can be verified in logarithmic space, and after each such verification the verifier only needs to store $c_i$. Thus the entire certificate above is verifiable in logarithmic space. ■

**Corollary 7** *If $s(n) \geq \log n$ is space constructible, then* $\text{NSPACE}(s(n)) = \text{coNSPACE}(s(n))$.

**Proof** This is just an application of the reachability method. Let $L \in \text{coNSPACE}(s(n))$. Then there is a non-deterministic machine $M$ using space $s(n)$ and with the following property: if $x \in L$ then $M(x)$ accepts on *every* computation path, while if $x \notin L$ then there is some computation path on which $M(x)$ rejects. Considering the configuration graph $G_M$ of $M(x)$ for some input $x$ of length $n$, we see that $x \in L$ iff there is *no* directed path in $G_M$ from the starting configuration to the *rejecting* configuration. Since $G_M$ has $V = 2^{O(s(n))}$ vertices, and the existence of an edge between two vertices $i$ and $j$ can be determined in $O(s(n)) = O(\log V)$ space, we can apply the previous theorem to get a non-deterministic algorithm deciding $L$ in space $O(\log V) = O(s(n))$. ■

**Corollary 8** $\mathsf{NL} = \mathsf{coNL}$.

**Notes on Complexity Theory** Last updated: September, 2011

Lecture 8

*Jonathan Katz*

# 1 The Polynomial Hierarchy

We have seen the classes $\mathcal{NP}$ and $\mathsf{co}\mathcal{NP}$, which are defined as follows:

*$L \in \mathcal{NP}$ if there is a (deterministic) Turing machine $M$ running in time polynomial in its first input, such that*

$$x \in L \Leftrightarrow \exists w\, M(x, w) = 1.$$

*$L \in \mathsf{co}\mathcal{NP}$ if there is a (deterministic) Turing machine $M$ running in time polynomial in its first input, such that*

$$x \in L \Leftrightarrow \forall w\, M(x, w) = 1.$$

It is natural to generalize the above; doing so allows us to capture problems that are "more difficult" than $\mathcal{NP} \cup \mathsf{co}\mathcal{NP}$. As an example, consider the language IND-SET:

$$\text{IND-SET} \stackrel{\text{def}}{=} \{(G, k) : G \text{ has an independent set of size} \geq k\}.$$

We know that IND-SET $\in \mathcal{NP}$; a certificate is just an independent set of vertices of size at least $k$ (that the vertices are independent can easily be verified). How about the following language?

$$\text{MAX-IND-SET} \stackrel{\text{def}}{=} \{(G, k) : \text{the largest independent set in } G \text{ has size exactly } k\}.$$

This language does not appear to be in $\mathcal{NP}$: we can certify that some graph $G$ has an independent set of size $k$, but how do we certify that this is the *largest* independent set in $G$? The language does not appear to be in $\mathsf{co}\mathcal{NP}$, either: although we could prove that $(G, k) \notin$ MAX-IND-SET if $G$ happened to have an independent set of size *larger* than $k$, there is no easy way to prove that $(G, k) \notin$ MAX-IND-SET in case its largest independent set has size *smaller* than $k$.

As another example, consider the problem of CNF-formula minimization. A CNF formula $\phi$ on $n$ variables naturally defines a function $f_\phi : \{0,1\}^n \to \{0,1\}$, where $f_\phi(x) = 1$ iff the given assignment $x$ satisfies $\phi$. Can we tell when a given formula is minimal? Consider the language

$$\text{MIN-CNF} \stackrel{\text{def}}{=} \{\phi : \text{no formula with fewer clauses than } \phi \text{ computes } f_\phi\}.$$

This language does not appear to be in $\mathcal{NP}$ or $\mathsf{co}\mathcal{NP}$, either. (Even if I give you a smaller formula $\phi'$ that computes $f_\phi$, there is no obvious way for you to verify that fact efficiently.)

The above examples motivate the following definition:

**Definition 1** *Let $i$ be a positive integer. $L \in \Sigma_i$ if there is a (deterministic) Turing machine $M$ running in time polynomial in its first input, such that*

$$x \in L \Leftrightarrow \underbrace{\exists w_1 \forall w_2 \cdots Q_i w_i}_{i \text{ times}}\, M(x, w_1, \ldots, w_i) = 1.$$

*where $Q_i = \forall$ if $i$ is even, and $Q_i = \exists$ if $i$ is odd.*

*$L \in \Pi_i$ if there is a (deterministic) Turing machine $M$ running in time polynomial in its first input, such that*

$$x \in L \Leftrightarrow \underbrace{\forall w_1 \exists w_2 \cdots Q_i w_i}_{i \text{ times}} M(x, w_1, \ldots, w_i) = 1.$$

*where $Q_i = \forall$ if $i$ is odd, and $Q_i = \exists$ if $i$ is even.*

As in the case of $\mathcal{NP}$, we may assume without loss of generality that the $w_i$ each have length polynomial in $x$. Note also the similarity to TQBF. The crucial difference is that TQBF allows an *unbounded* number of alternating quantifiers, whereas $\Sigma_i, \Pi_i$ each allow (at most) $i$ quantifiers. Since TQBF is PSPACE-complete, this implies that $\Sigma_i, \Pi_i \in$ PSPACE for all $i$. (One could also prove this directly, via a PSPACE algorithm just like the one used to solve TQBF.)

Returning to our examples, note that MAX-IND-SET $\in \Sigma_2$ since $(G, k) \in$ MAX-IND-SET iff there exists a set of vertices $S$ such that for all sets of vertices $S'$ the following (efficiently verifiable) predicate is true:

> $|S| = k$ and $S$ is an independent set of $G$; moreover, either $|S'| \leq k$ or $S'$ is not an independent set of $G$.

(It is easier to express the above in English as: "there exists a set $S$ of $k$ vertices, such that for all sets $S'$ containing more than $k$ vertices, $S$ is an independent set and $S'$ is not an independent set". But one has to be careful to check that anytime the quantification is limited [e.g., by quantifying over all sets of size greater than $k$, rather than all sets], the limitation is efficient to verify.) We also have MAX-IND-SET $\in \Pi_2$ since we can swap the order of quantifiers in this case (since $S'$ does not depend on $S$, above). Turning to the second example, note MIN-CNF $\in \Pi_2$ since $\phi \in$ MIN-CNF iff for all formulas $\phi'$ that are smaller than $\phi$ there exists an input $x$ for which $\phi(x) \neq \phi(x')$. Here, however, we cannot just swap the order of quantifiers (do you see why?), and so we do not know, or believe, that MIN-CNF $\in \Sigma_2$.

We make some relatively straightforward observations, leaving their proofs as exercises.

- $\Sigma_1 = \mathcal{NP}$ and $\Pi_1 = \text{co}\mathcal{NP}$.
- For all $i$, we have $\text{co}\Sigma_i = \Pi_i$ (and $\text{co}\Pi_i = \Sigma_i$).
- For all $i$, we have $\Sigma_i, \Pi_i \subseteq \Pi_{i+1}$ and $\Sigma_i, \Pi_i \subseteq \Sigma_{i+1}$.

The *polynomial hierarchy* PH consists of all those languages of the form defined above; that is, $\text{PH} \stackrel{\text{def}}{=} \bigcup_i \Sigma_i = \bigcup_i \Pi_i$. Since $\Sigma_i \subseteq$ PSPACE for all $i$, we have PH $\subseteq$ PSPACE. Each $\Sigma_i$ (resp., $\Pi_i$) is called a *level* in the hierarchy.

As in the case of $\mathcal{NP}$ and $\text{co}\mathcal{NP}$, we believe

**Conjecture 1** *$\Sigma_i \neq \Pi_i$ for all $i$.*

If the above conjecture is not true, then the polynomial hierarchy *collapses* in the following sense:

**Theorem 2** *If $\Sigma_i = \Pi_i$ for some $i$, then* PH $= \Sigma_i$.

**Proof** We show that for all $j > i$, we have $\Sigma_j = \Sigma_{j-1}$. Let $L \in \Sigma_j$. So there is a polynomial-time Turing machine $M$ such that

$$x \in L \Leftrightarrow \exists w_j \forall w_{j-1} \cdots Q w_1 \; M(x, w_j, \ldots, w_1) = 1,$$

where we have numbered the variables in descending order for clarity in what follows. Assume $j - i$ is even. (A symmetric argument works if $j - i$ is odd.) Define language $L'$ as

$$(x, w_j, \ldots, w_{i+1}) \in L' \Leftrightarrow \exists w_i \forall w_{i-1} \cdots Q w_1 \; M(x, w_j, \ldots, w_1) = 1,$$

and note that $L' \in \Sigma_i$. By the assumption of the theorem, $L' \in \Pi_i$ and so there is some machine $M'$ running in time polynomial in $|x| + |w_j| + \cdots + |w_{i+1}|$ (and hence polynomial in $|x|$) such that

$$(x, w_j, \ldots, w_{i+1}) \in L' \Leftrightarrow \forall w'_i \exists w'_{i-1} \cdots Q' w'_1 \; M'(x, w_j, \ldots, w_{i+1}, w'_i, \ldots, w'_1) = 1.$$

(Note that the $\{w'_i, \ldots, w'_1\}$ need not have any relation with the $\{w_i, \ldots, w_1\}$.) But then

$$\begin{aligned} x \in L &\Leftrightarrow \exists w_j \forall w_{j-1} \cdots \forall w_{i+1} \;\; (x, w_j, \ldots, w_{i+1}) \in L' \\ &\Leftrightarrow \exists w_j \forall w_{j-1} \cdots \forall w_{i+1} \left[ \forall w'_i \exists w'_{i-1} \cdots Q' w'_1 \;\; M'(x, w_j, \ldots, w'_1) = 1 \right] \\ &\Leftrightarrow \exists w_j \forall w_{j-1} \cdots \forall w_{i+1} w'_i \exists w'_{i-1} \cdots Q' w'_1 \;\; M'(x, w_j, \ldots, w'_1) = 1, \end{aligned}$$

and there are only $j - 1$ quantifiers in the final expression. Thus, $L \in \Sigma_{j-1}$. ■

A similar argument gives:

**Theorem 3** *If $\mathcal{P} = \mathcal{NP}$ then $\mathsf{PH} = \mathcal{P}$.*

Each $\Sigma_i$ has the complete problem $\Sigma_i$-SAT, where this language contains all true expressions of the form $\exists w_1 \forall w_2 \cdots Q w_i \; \phi(w_1, \ldots, w_i) = 1$ for $\phi$ a boolean formula in CNF form. However, if PH has a complete problem, the polynomial hierarchy collapses: If $L$ were PH-complete then $L \in \Sigma_i$ for some $i$; but then every language $L' \in \Sigma_{i+1} \subseteq \mathsf{PH}$ would be reducible to $L$, implying $\Sigma_{i+1} = \Sigma_i$. Since PSPACE has complete languages, this indicates that PH is strictly contained in PSPACE.

## 1.1 Alternating Turing Machines

The polynomial hierarchy can also be defined using *alternating Turing machines*. An alternating Turing machine (ATM) is a non-deterministic Turing machine (so it has two transition functions, one of which is non-deterministically chosen in each step of its computation) in which every state is labeled with either $\exists$ or $\forall$. To determine whether a given ATM $M$ accepts an input $x$, we look at the configuration graph of $M(x)$ and recursively mark the configurations of $M(x)$ as follows:

- The accepting configuration is marked "accept".
- A configuration whose state is labeled $\exists$ that has an edge to a configuration marked "accept" is itself marked "accept".
- A configuration whose state is labeled $\forall$, both of whose edges are to configurations marked "accept," is itself marked "accept".

$M(x)$ accepts iff the initial configuration of $M(x)$ is marked "accept".

We can define classes $\text{ATIME}(t(n))$ and $\text{ASPACE}(s(n))$ in the natural way. We can also define $\Sigma_i\text{TIME}(t(n))$ (resp., $\Pi_i\text{TIME}(t(n))$) to be the class of languages accepted by an ATM running in time $O(t(n))$, whose initial state is labeled $\exists$ (resp., $\forall$), and that makes at most $i-1$ alternations between states labeled $\exists$ and states labeled $\forall$. Note that $L \in \Sigma_2\text{TIME}(t(n))$ iff

$$x \in L \Leftrightarrow \exists w_1 \forall w_2 \; M(x, w_1, w_2),$$

where $M$ is a deterministic Turing machine running in time $t(|x|)$. (The situation for space-bounded ATMs is tricker, since the length of the certificates themselves must be accounted for.) Given the definitions, it should not be surprising that $\Sigma_i = \bigcup_c \Sigma_i\text{TIME}(n^c)$ and $\Pi_i = \bigcup_c \Pi_i\text{TIME}(n^c)$. We also have $\mathsf{PSPACE} = \bigcup_c \text{ATIME}(n^c)$ (this follows readily from the fact that $\mathsf{TQBF}$ is $\mathsf{PSPACE}$-complete).

Defining the polynomial hierarchy in terms of ATMs may seem more cumbersome, but has some advantages (besides providing another perspective). Perhaps the main advantage is that it allows for studying more refined notions of complexity, e.g., $\Sigma_i\text{TIME}(n^2)$ vs. $\Sigma_i\text{TIME}(n)$.

It is known that

**Theorem 4**

- $\text{NSPACE}(s(n)) \subseteq \text{ATIME}(s(n)^2) \subseteq \text{SPACE}(s(n)^2)$ *for time/space-constructible* $s(n) \geq n$.
- $\text{ASPACE}(s(n)) = \text{TIME}(2^{O(s(n))})$ *for time/space-constructible* $s(n) \geq \log n$.

**Proof** See [2]. ■

We now show an application of ATMs to proving a result about standard non-deterministic machines. Let $\text{TIMESPC}(t(n), s(n))$ be the class of languages that can be decided by a (deterministic) Turing machine $M$ that runs in time $O(t(n))$ and uses space $O(s(n))$. (Note that $\text{TIMESPC}(t(n), s(n))$ is not the same as $\text{TIME}(t(n)) \cap \text{SPACE}(s(n))$.)

**Theorem 5** $\mathsf{SAT} \notin \text{TIMESPC}(n^{1.1}, n^{0.1})$.

**Proof** We show that $\text{NTIME}(n) \not\subseteq \text{TIMESPC}(n^{1.2}, n^{0.2})$, which implies the theorem because (by the strong version of the Cook-Levin theorem proved in [1]) deciding membership of $x \in \{0,1\}^n$ in some language $L \in \text{NTIME}(n)$ can be reduced to solving $\mathsf{SAT}$ on an instance of length $O(n \log n)$.

We begin with the following claim (which is of independent interest).

**Claim 6** $\text{TIMESPC}(n^{12}, n^2) \subseteq \Sigma_2\text{TIME}(n^8)$.

**Proof** Let $L \in \text{TIMESPC}(n^{12}, n^2)$, and let $M$ be a machine deciding $L$ in time $O(n^{12})$ and space $O(n^2)$. Considering the configuration graph of $M(x)$, we see that $M(x)$ accepts iff there exist a sequence of configurations $c_0, \ldots, c_{n^6}$ such that (1) $c_0$ is the initial configuration and $c_{n^6}$ is the (unique) accepting configuration, and (2) for all $i$, machine $M(x)$ goes from configuration $c_i$ to configuration $c_{i+1}$ in at most $O(n^6)$ steps. Each configuration can be specified using $O(n^2)$ bits, and so the sequence $c_1, \ldots, c_{n^6}$ can be written down in $O(n^8)$ time. Moreover, existence of a path of length $O(n^6)$ from $c_i$ to $c_{i+1}$ can be verified in $O(n^8)$ time. ■

The above claim was unconditional. The next claim uses the assumption that $\text{NTIME}(n) \subseteq \text{TIMESPC}(n^{1.2}, n^{0.2})$.

**Claim 7** *If* $\text{NTIME}(n) \subseteq \text{TIMESPC}(n^{1.2}, n^{0.2})$*, then* $\Sigma_2\text{TIME}(n^8) \subseteq \text{NTIME}(n^{9.6})$.

**Proof** Since $\text{TIMESPC}(n^{1.2}, n^{0.2}) \subseteq \text{TIME}(n^{1.2})$ and $\text{TIME}(s(n))$ is closed under complementation, the condition of the claim implies $\text{coNTIME}(n) \subseteq \text{TIME}(n^{1.2})$. If $L \in \Sigma_2\text{TIME}(n^8)$ then

$$x \in L \Leftrightarrow \exists w_1 \forall w_2 \; M(x, w_1, w_2) = 1,$$

where $M$ is a deterministic machine running in time $O(|x|^8)$. But then

$$L' \stackrel{\text{def}}{=} \{(x, w_1) : \forall w_2 \; M(x, w_1, w_2)\} \subseteq \text{coNTIME}(n^8) \subseteq \text{TIME}\left(\left(n^8\right)^{1.2}\right) = \text{TIME}(n^{9.6})$$

where $n$ denotes the length of $x$. If we let $M'$ be a deterministic machine deciding $L'$, where $M(x, w_1)$ runs in time $O(|x|^{9.6})$, then

$$x \in L \Leftrightarrow \exists w_1 \; M'(x, w_1) = 1$$

and hence $L \in \text{NTIME}(n^{9.6})$. ■

We now put everything together to prove the theorem. Assume toward a contradiction that $\text{NTIME}(n) \subseteq \text{TIMESPC}(n^{1.2}, n^{0.2})$. Then

$$\text{NTIME}(n^{10}) \subseteq \text{TIMESPC}(n^{12}, n^2) \subseteq \Sigma_2\text{TIME}(n^8) \subseteq \text{NTIME}(n^{9.6}),$$

using Claim 6 for the second inclusion, and Claim 7 for the last inclusion. But this contradicts the non-deterministic time-hierarchy theorem. ■

## References


[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[2] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *J. ACM* 28(1): 114–133, 1981.

**Notes on Complexity Theory** Last updated: October, 2011

Lecture 9

*Jonathan Katz*

# 1 The Polynomial Hierarchy

## 1.1 Defining the Polynomial Hierarchy via Oracle Machines

Here we show a third definition of the levels of the polynomial hierarchy in terms of oracle machines.

**Definition 1** *Define $\Sigma_i, \Pi_i$ inductively as follows:*

- $\Sigma_0 \stackrel{\text{def}}{=} \mathcal{P}$.
- $\Sigma_{i+1} \stackrel{\text{def}}{=} \mathcal{NP}^{\Sigma_i}$ *and* $\Pi_{i+1} = \mathsf{co}\mathcal{NP}^{\Sigma_i}$.

(Note that even though we believe $\Sigma_i \neq \Pi_i$, oracle access to $\Sigma_i$ gives the same power as oracle access to $\Pi_i$. Do you see why?)

We show that this leads to an equivalent definition. *For this section only*, let $\Sigma_i^O$ refer to the definition in terms of oracles. We prove by induction that $\Sigma_i = \Sigma_i^O$. (Since $\Pi_i^O = \mathsf{co}\Sigma_i^O$, this proves it for $\Pi_i, \Pi_i^O$ as well.) For $i = 1$ this is immediate, as $\Sigma_1 = \mathcal{NP} = \mathcal{NP}^{\mathcal{P}} = \Sigma_1^O$.

Assuming $\Sigma_i = \Sigma_i^O$, we prove that $\Sigma_{i+1} = \Sigma_{i+1}^O$. Let us first show that $\Sigma_{i+1} \subseteq \Sigma_{i+1}^O$. Let $L \in \Sigma_{i+1}$. Then there exists a polynomial-time Turing machine $M$ such that

$$x \in L \Leftrightarrow \exists w_1 \forall w_2 \cdots Q_{i+1} w_{i+1}\ M(x, w_1, \ldots, w_{i+1}) = 1.$$

In other words, there exists a language $L' \in \Pi_i$ such that

$$x \in L \Leftrightarrow \exists w_1\ (x, w_1) \in L'.$$

By our inductive assumption, $\Pi_i = \Pi_i^O$; thus, $L \in \mathcal{NP}^{\Pi_i^O} = \mathcal{NP}^{\Sigma_i^O} = \Sigma_{i+1}^O$ and so $\Sigma_{i+1} \subseteq \Sigma_{i+1}^O$.

It remains to show that $\Sigma_{i+1}^O \subseteq \Sigma_{i+1}$ (assuming $\Sigma_i^O = \Sigma_i$). Let $L \in \Sigma_{i+1}^O$. This means there exists a non-deterministic polynomial-time machine $M$ and a language $L' \in \Sigma_i^O$ such that $M$, given oracle access to $L_i$, decides $L$. In other words, $x \in L$ iff $\exists y, q_1, a_1, \ldots, q_n, a_n$ (here, $y$ represents the non-deterministic choices of $M$, while $q_j, a_j$ represent the queries/answers of $M$ to/from its oracle) such that:

1. $M$, on input $x$, non-deterministic choices $y$, and oracle answers $a_1, \ldots, a_n$, makes queries $q_1, \ldots, q_n$ and accepts.
2. For all $j$, we have $a_j = 1$ iff $q_j \in L'$.

Since $L' \in \Sigma_i^O = \Sigma_i$ (by our inductive assumption) we can express the second condition, above, as:

- $a_j = 1 \Leftrightarrow \exists y_1^j \forall y_2^j \cdots Q_i y_i^j\ M'(q_j, y_1^j, \ldots, y_i^j) = 1$
- $a_j = 0 \Leftrightarrow \forall y_1^j \exists y_2^j \cdots Q_i' y_i^j\ M'(q_j, y_1^j, \ldots, y_i^j) = 0$

for some (deterministic) polynomial-time machine $M'$. The above leads to the following specification of $L$ as a $\Sigma_{i+1}$ language:

$$x \in L \text{ iff } \exists \left( y, q_1, a_1, \ldots, q_n, a_n, \{y_1^j\}_{j=1}^n \right) \forall \left( \{y_2^j\}_{j=1}^n \right) \cdots Q_{i+1} \left( \{y_{i+1}^j\}_{j=1}^n \right):$$

- $M$, on input $x$, non-deterministic choices $y$, and oracle answers $a_1, \ldots, a_n$, makes queries $q_1, \ldots, q_n$ and accepts, and
- Let $Y$ be the set of $j$'s such that $a_j = 1$, and let $N$ be the set of $j$'s such that $a_j = 0$.
  - For all $j \in Y$, we have $M'(q_j, y_1^j, \ldots, y_i^j) = 1$
  - For all $j \in N$, we have $M'(q_j, y_2^j, \ldots, y_{i+1}^j) = 0$.

## 2 Non-Uniform Complexity

*Boolean circuits* offer an alternate model of computation: a *non-uniform* one as opposed to the *uniform* model of Turing machines. (The term "uniform" is explained below.) In contrast to Turing machines, circuits are not meant to model "realistic" computations for arbitrary-length inputs. Circuits are worth studying for at least two reasons, however. First, when one is interested in inputs of some *fixed size* (or range of sizes), circuits make sense as a computational model. (In the real world, efficient circuit design has been a major focus of industry.) Second, from a purely theoretical point of view, the hope has been that circuits would somehow be "easier to study" than Turing machines (even though circuits are more powerful!) and hence that it might be easier to prove lower bounds for the former than for the latter. The situation here is somewhat mixed: while some circuit lower bounds *have* been proved, those results have not really led to any significant separation of uniform complexity classes.

Circuits are directed, acyclic graphs where nodes are called *gates* and edges are called *wires*. *Input gates* are gates with in-degree zero, and we will take the *output gate* of a circuit to be the (unique) gate with out-degree zero. (For circuits having multiple outputs there may be multiple output gates.) In a boolean circuit, each input gate is identified with some bit of the input; each non-input gate is labeled with a value from a given *basis* of boolean functions. The standard basis is $\mathcal{B}_0 = \{\neg, \vee, \wedge\}$, where each gate has *bounded* fan-in. Another basis is $\mathcal{B}_1 = \{\neg, (\vee_i)_{i \in \mathbb{N}}, (\wedge_i)_{i \in \mathbb{N}}\}$, where $\vee_i, \wedge_i$ have in-degree $i$ and we say that this basis has *unbounded* fan-in. In any basis, gates may have unbounded fan-*out*.

A circuit $C$ with $n$ input gates defines a function $C : \{0,1\}^n \to \{0,1\}$ in the natural way: a given input $x = x_1 \cdots x_n$ immediately defines the values of the input gates; the values at any internal gate are determined inductively; $C(x)$ is then the value of the output gate. If $f : \{0,1\}^* \to \{0,1\}$ is a function, then a circuit family $\mathcal{C} = \{C_i\}_{i \in \mathbb{N}}$ computes $f$ if $f(x) = C_{|x|}(x)$ for all $x$. In other words, for all $n$ the circuit $C_n$ agrees with $f$ restricted to inputs of length $n$. (A circuit family decides a language if it computes the characteristic function for that language.) This is the sense in which circuits are *non*-uniform: rather than having a fixed algorithm computing $f$ on all input lengths (as is required, e.g., in the case of Turing machines), in the non-uniform model there may be a completely different "algorithm" (i.e., circuit) for each input length.

Two important complexity measures for circuits are their *size* and their *depth*.[1] The size of a

[1] When discussing circuit size and depth, it is important to be clear what *basis* for the circuit is assumed. By default, we assume basis $\mathcal{B}_0$ unless stated otherwise.

circuit is the number of gates it has. The depth of a circuit is the length of the longest path from an input gate to an output gate. A circuit family $\mathcal{C} = \{C_n\}_{n\in\mathbb{N}}$ has size $T(\cdot)$ if, for all sufficiently large $n$, circuit $C_n$ has size at most $T(n)$. It has depth $D(\cdot)$ if, for all sufficiently large $n$, circuit $C_n$ has depth at most $D(n)$. The usual convention is not to count "not" gates in either of the above: one can show that all the "not" gates of a circuit can be pushed to immediately follow the input gates; thus, ignoring "not" gates affects the size by at most $n$ and the depth by at most 1.

**Definition 2** *$L \in \text{SIZE}(T(n))$ if there is a circuit family $\mathcal{C} = \{C_n\}$ of size $T(\cdot)$ that decides $L$.*

We stress that the above is defined over $\mathcal{B}_0$. Note also that we do not use big-$O$ notation, since there is no "speedup theorem" in this context.

One could similarly define complexity classes in terms of circuit depth (i.e., $L \in \text{DEPTH}(D(n))$ if there is a circuit family $\mathcal{C} = \{C_n\}$ of depth $D(\cdot)$ that decides $L$); circuit depth turns out to be somewhat less interesting unless there is simultaneously a bound on the circuit size.

## 2.1 The Power of Circuits

We have seen this before (in another context) but it is worth stating again: *every* function — even an undecidable one! — is computable by a circuit over the basis $\mathcal{B}_0$. Let us first show how to express any $f$ as a circuit over $\mathcal{B}_1$. Fix some input length $n$. Define $F_0 \stackrel{\text{def}}{=} \{x \in \{0,1\}^n \mid f(x) = 0\}$ and define $F_1$ analogously. We can express $f$ (restricted to inputs of length $n$) as:

$$f(x) = \bigvee_{x' \in F_1} [x = x'],$$

where $[x = x']$ denotes a boolean expression which is true iff $x = x'$. (Here, $x$ represents the variables, and $x'$ is a fixed string.) Letting $x_i$ denote the $i$th bit of $x$, note that $[x = x'] \Leftrightarrow (\bigwedge_{i:x'=1} x_i) \wedge (\bigwedge_{i:x'=0} \bar{x}_i)$. Putting everything together, we have:

$$f(x) = \bigvee_{x' \in F_1} \left( (\textstyle\bigwedge_{i:x'=1} x_i) \wedge (\bigwedge_{i:x'=0} \bar{x}_i) \right). \tag{1}$$

But the above is just a circuit of depth[2] 2 over $\mathcal{B}_1$. (The *size* of the circuit is at most $\Theta(2^n)$.) The above representation is called the *disjunctive normal form* (DNF) for $f$. Another way to express $f$ is as:

$$f(x) = \bigwedge_{x' \in F_0} [x \neq x'],$$

where $[x \neq x']$ has the obvious meaning. Note, $[x \neq x'] \Leftrightarrow \left(\bigvee_{i:x'_i=1} \bar{x}_i\right) \vee \left(\bigvee_{i:x'_i=0} x_i\right)$; putting everything together gives:

$$f(x) = \bigwedge_{x' \in F_0} \left( \left(\textstyle\bigvee_{i:x'_i=1} \bar{x}_i\right) \vee \left(\bigvee_{i:x'_i=0} x_i\right) \right), \tag{2}$$

the *conjunctive normal form* (CNF) for $f$. This gives another circuit of depth 2 over $\mathcal{B}_1$.

The above show how to obtain a circuit for $f$ over the basis $\mathcal{B}_1$. But one can transform any circuit over $\mathcal{B}_1$ to one over $\mathcal{B}_0$. The idea is simple: each $\vee$-gate of in-degree $k$ is replaced by a "tree"

[2]Recall that "not" gates are not counted.

of degree-2 $\vee$-gates, and each $\wedge$-gate of in-degree $k$ is replaced by a "tree" of degree-2 $\wedge$-gates. In each case we transform a single gate having fan-in $k$ to a sub-circuit with $k-1$ gates having depth $\lceil \log k \rceil$. Applying this transformation to Eqs. (1) and (2), we obtain a circuit for any function $f$ over the basis $\mathcal{B}_0$ with at most $n \cdot 2^n$ gates and depth at most $n + \lceil \log n \rceil$. We thus have:

**Theorem 1** *Every function is in* $\text{SIZE}(n \cdot 2^n)$.

This can be improved to show that for every $\varepsilon > 0$ every function is in $\text{SIZE}\left((1+\varepsilon) \cdot \frac{2^n}{n}\right)$. This is tight up to low-order terms, as we show next time.

## Bibliographic Notes

For more on circuit complexity see the classic text by Wegener [6] and the excellent book by Vollmer [5]. (The forthcoming book by Jukna [2] also promises to be very good.) The claim that all functions can be computed by circuits of size $(1+\varepsilon) \cdot 2^n/n$ was proven by Lupanov. A proof of a weaker claim (showing this bound over the basis $\{\vee, \wedge, \neg, \oplus\}$) can be found in my notes from 2005 [3, Lecture 5]. A proof over the basis $\mathcal{B}_0$ can be found in [4, Section 2.13]. Frandsen and Miltersen [1] give another exposition, and discuss what is known about the low-order terms in both the upper and lower bounds.

## References


[1] G.S. Frandsen and P.B. Miltersen. Reviewing Bounds on the Circuit Size of the Hardest Functions. *Information Processing Letters* 95(2): 354–357, 2005. Available on-line at `http://www.daimi.au.dk/~bromille/Papers/shannon.pdf`

[2] S. Jukna. *Boolean Function Complexity: Advances and Frontiers*, Springer, 2012.

[3] J. Katz. Lecture notes for *CMSC 652 — Complexity Theory*. Fall 2005.

[4] J.E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, 1998.

[5] H. Vollmer. *Introduction to Circuit Complexity*. Springer, 1999.

[6] I. Wegener. *The Complexity of Boolean Functions*. John Wiley & Sons, 1987.

Lecture 10

*Jonathan Katz*

# 1 Non-Uniform Complexity

## 1.1 The Power of Circuits

Last time we saw that every function $f : \{0,1\}^n \to \{0,1\}$ could be computed by a circuit of size $n \cdot 2^n$, and noted that this bound could be improved to $(1+\varepsilon) \cdot 2^n/n$ for every $\varepsilon > 0$. We now show that this is essentially tight.

**Theorem 1** *Let $\varepsilon > 0$ and $q(n) = (1-\varepsilon)\frac{2^n}{n}$. Then for $n$ large enough there exists a function $f : \{0,1\}^n \to \{0,1\}$ that cannot be computed by a circuit of size at most $q(n)$.*

**Proof** In fact, the fraction of functions $f : \{0,1\}^n \to \{0,1\}$ that can be computed by circuits of size at most $q(n)$ approaches 0 as $n$ approaches infinity; this easily follows from the proof below.

Let $q = q(n)$. The proof is by a counting argument. We count the number of circuits of size $q$ (note that if a function can be computed by a circuit of size at most $q$, then by adding useless gates it can be computed by a circuit of size exactly $q$) and show that this is less than the number of $n$-ary functions. A circuit having $q$ internal gates is defined by (1) specifying, for each internal gate, its type and its two predecessor gates, and (2) specifying the output gate. We may assume without loss of generality that each gate of the circuit computes a different function — otherwise, we can simply remove all but one copy of the gate (and rewire the circuit appropriately). Under this assumption, permuting the numbering of the internal gates does not affect the function computed by the circuit. Thus, the number of circuits with $q$ internal gates is at most:

$$\frac{\left(3 \cdot (q+n)^2\right)^q \cdot q}{q!} \leq \frac{\left(12 \cdot (q)^2\right)^q \cdot q}{q!}.$$

In fact, we are over-counting since some of these are not valid circuits (e.g., they are cyclic). We have:

$$\begin{aligned} \frac{q \cdot \left(12 \cdot (q+n)^2\right)^q}{q!} &\leq q \cdot (36)^q \cdot \frac{q^{2q}}{q^q} \\ &= q \cdot (36 \cdot q)^q \\ &\leq (36 \cdot q)^{q+1} \\ &\leq (2^n)^{(1-\varepsilon)2^n/n\,+\,1} \;=\; 2^{(1-\varepsilon)2^n+n}, \end{aligned}$$

for $n$ sufficiently large, using Stirling's bound $q! \geq q^q/e^q \geq q^q/3^q$ for the first inequality. But this is less than $2^{2^n}$ (the number of $n$-ary boolean functions) for $n$ large enough. ■

We saw that any function can be computed by a circuit family of depth $n + \lceil \log n \rceil \leq (1+\varepsilon) \cdot n$ (for $n$ large enough) for any $\varepsilon > 0$. This, too, is essentially tight (see [1, Sect. 2.12]):

**Theorem 2** *Let $\varepsilon > 0$ and $d(n) = (1-\varepsilon) \cdot n$. Then for $n$ large enough there exists a function $f : \{0,1\}^n \to \{0,1\}$ that cannot be computed by a circuit of depth at most $d(n)$.*

## 1.2 Polynomial-Size Circuits

As with our interest in polynomial-time algorithms, we are also interested in polynomial-size circuits. Define $\mathcal{P}_{/\mathsf{poly}} \stackrel{\text{def}}{=} \bigcup_c \text{SIZE}(n^c)$. A second characterization of $\mathcal{P}_{/\mathsf{poly}}$ is:

**Definition 1** *$L \in \mathcal{P}_{/\mathsf{poly}}$ iff there exists a Turing machine $M$ running in time polynomial in its first input, and a sequence of "advice strings" $\{z_n\}_{n\in\mathbb{N}}$ such that $x \in L$ iff $M(x, z_n) = 1$.*

A proof that this is an equivalent definition uses the proof that follows, and is left as an exercise.

Perhaps not surprisingly, we have

**Theorem 3** *$\mathcal{P} \subseteq \mathcal{P}_{/\mathsf{poly}}$.*

**Proof** We prove a stronger result: for any $t(n) \geq n$ we have $\text{TIME}(t(n)) \subseteq \text{SIZE}(t(n) \cdot \log t(n))$. Given a language $L \in \text{TIME}(t(n))$, we first construct an *oblivious* Turing machine $M$ deciding $L$ in time $t'(n) = O(t(n) \cdot \log t(n))$. For simplicity, assume $M$ has a single work tape (in addition to its input tape); the proof is similar if $M$ has $k$ work tapes. Considering the tableau of the computation of $M(x)$ with $x \in \{0,1\}^n$, we see that $M(x)$ passes through (at most) $t'(n)+1$ configurations and each configuration can be represented using $O(t'(n))$ bits. We will imagine associating a set of wires with each configuration of $M(x)$. Note, however, that in passing from one configuration to the next most of the wire values will remain unchanged and, because of the obliviousness of $M$, we know in advance which wires can possibly change. Wires that do not change can simply be "extended" from one configuration to the next, without using any gates.

In a bit more detail, we show how to compute one configuration from the next using only a constant number of gates. To go from configuration $i$ to configuration $i+1$, we construct a boolean sub-circuit that takes as input (1) the state of $M$ at time $i$, (2) the wires representing the cell scanned by $M$ at time $i$ (here we use the assumption that $M$ is oblivious, so we know in advance where the work-tape head of $M$ will be at every time step); and (3) the position of the input scanned by $M$ at time $i$ (once again using obliviousness). The output of this sub-circuit is (1) the updated state of $M$ and (2) an updated value for what is written on the cell scanned by $M$ at the previous time step. Note that the number of inputs to this sub-circuit is constant, so by what we have shown previously we can compute the transition function using a constant number of gates. Using similar reasoning, we can also add a final sub-circuit that tests whether the final state of $M$ is accepting or not. We thus have $O(t'(n))$ constant-size sub-circuits, for a circuit of total size $O(t'(n))$. ■

The above proof can also be used to show that the following problem is $\mathcal{NP}$-complete:

$$\mathsf{CKT\text{-}SAT} \stackrel{\text{def}}{=} \{C \mid C \text{ is a circuit, and } \exists x \text{ s.t. } C(x) = 1\}.$$

Could it be that $\mathcal{P} = \mathcal{P}_{/\mathsf{poly}}$? Actually, here we know that the inclusion is strict, since $\mathcal{P}_{/\mathsf{poly}}$ is still powerful enough to contain undecidable languages. To see this, let $L \subseteq 1^*$ be an undecidable unary language, and note that any unary language is trivially in $\mathcal{P}_{/\mathsf{poly}}$.

Could it be that $\mathcal{NP} \subseteq \mathcal{P}_{/\mathsf{poly}}$? This would be less surprising than $\mathcal{P} = \mathcal{NP}$, and would not necessarily have any practical significance (frustratingly, $\mathcal{NP} \subseteq \mathcal{P}_{/\mathsf{poly}}$ but $\mathcal{P} \neq \mathcal{NP}$ would mean that efficient algorithms for $\mathcal{NP}$ *exist*, but can't be found efficiently). Nevertheless, the following result suggests that $\mathcal{NP} \not\subseteq \mathcal{P}_{/\mathsf{poly}}$:

**Theorem 4 (Karp-Lipton)** *If $\mathcal{NP} \subseteq \mathcal{P}_{/\mathsf{poly}}$ then $\Sigma_2 = \Pi_2$ (and hence $\mathsf{PH} = \Sigma_2$).*

**Proof** We begin with a claim that can be proved easily given our earlier work on self-reducibility of $\mathsf{SAT}$: if $\mathsf{SAT} \in \mathcal{P}_{/\mathsf{poly}}$ then there exists a polynomial-size circuit family $\{C_n\}$ such that $C_{|\phi|}(\phi)$ outputs a satisfying assignment for $\phi$ if $\phi$ is satisfiable. That is, if $\mathsf{SAT}$ can be *decided* by polynomial-size circuits, then $\mathsf{SAT}$ can be *solved* by polynomial-size circuits.

We use this claim to prove that $\Pi_2 \subseteq \Sigma_2$ (from which the theorem follows). Let $L \in \Pi_2$. This means there is a Turing machine $M$ running in time polynomial in its first input such that[1]

$$x \in L \Leftrightarrow \forall y \exists z : M(x, y, z) = 1.$$

Define $L' = \{(x, y) \mid \exists z : M(x, y, z) = 1\}$. Note that $L' \in \mathcal{NP}$, and so there is a Karp reduction $f$ from $L'$ to $\mathsf{SAT}$. (The function $f$ can be computed in time polynomial in $|(x, y)|$, but since $|y| = \mathsf{poly}(|x|)$ this means it can be computed in time polynomial in $|x|$.) We may thus express membership in $L$ as follows:

$$x \in L \Leftrightarrow \forall y : f(x, y) \in \mathsf{SAT}. \tag{1}$$

But we then have

$$x \in L \Leftrightarrow \exists C \forall y : C(f(x, y)) \text{ is a satisfying assignment of } f(x, y),$$

where $C$ is interpreted as a circuit, and is chosen from strings of (large enough) polynomial length. Thus, $L \in \Sigma_2$. ■

### 1.3 Small Depth Circuits and Parallel Computation

Circuit depth corresponds to the time required for the circuit to be evaluated; this is also evidenced by the proof of Theorem 3. Moreover, a circuit of size $s$ and depth $d$ for some problem can readily be turned into a parallel algorithm for the problem using $s$ processors and running in "wall clock" time $d$. Thus, it is interesting to understand when low-depth circuits for problems exist. From a different point of view, we might expect that *lower bounds* would be easier to prove for low-depth circuits. These considerations motivate the following definitions.

**Definition 2** *Let $i \geq 0$. Then*

- $L \in \mathsf{NC}^i$ *if $L$ is decided by a circuit family $\{C_n\}$ of polynomial size and $O(\log^i n)$ depth over the basis $\mathcal{B}_0$.*
- $L \in \mathsf{AC}^i$ *if $L$ is decided by a circuit family $\{C_n\}$ of polynomial size and $O(\log^i n)$ depth over the basis $\mathcal{B}_1$.*

$\mathsf{NC} = \bigcup_i \mathsf{NC}^i$ *and* $\mathsf{AC} = \bigcup_i \mathsf{AC}^i$.

Note $\mathsf{NC}^i \subseteq \mathsf{AC}^i \subseteq \mathsf{NC}^{i+1}$. Also, $\mathsf{NC}^0$ is not a very interesting class since the function computed by a constant-depth circuit over $\mathcal{B}_0$ can only depend on a constant number of bits of the input.

## Bibliographic Notes

Theorem 1 is due to Shannon; the proof here is adapted from Vollmer [2].

[1] By convention, quantification is done over strings of length some (appropriate) fixed polynomial in $|x|$.

## References

[1] J.E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, 1998.

[2] H. Vollmer. *Introduction to Circuit Complexity*. Springer, 1999.

**Notes on Complexity Theory** Last updated: October, 2011

Lecture 11

*Jonathan Katz*

# 1 Non-Uniform Complexity

## 1.1 Circuit Lower Bounds for a Language in $\Sigma_2 \cap \Pi_2$

We have seen that there *exist* "very hard" languages (i.e., languages that require circuits of size $(1-\varepsilon)2^n/n$). If we can show that there exists a language in $\mathcal{NP}$ that is even "moderately hard" (i.e., requires circuits of super-polynomial size) then we will have proved $\mathcal{P} \neq \mathcal{NP}$. (In some sense, it would be even nicer to show some *concrete* language in $\mathcal{NP}$ that requires circuits of super-polynomial size. But mere existence of such a language is enough.)

Here we show that for every $c$ there is a language in $\Sigma_2 \cap \Pi_2$ that is not in $\text{SIZE}(n^c)$. Note that this does not prove $\Sigma_2 \cap \Pi_2 \not\subseteq \mathcal{P}_{/\mathsf{poly}}$ since, for every $c$, the language we obtain is different. (Indeed, using the time hierarchy theorem, we have that for every $c$ there is a language in $\mathcal{P}$ that is not in $\text{TIME}(n^c)$.) What is particularly interesting here is that (1) we prove a non-uniform lower bound and (2) the proof is, in some sense, rather simple.

**Theorem 1** *For every $c$, there is a language in $\Sigma_4 \cap \Pi_4$ that is not in $\text{SIZE}(n^c)$.*

**Proof** Fix some $c$. For each $n$, let $C_n$ be the lexicographically first circuit on $n$ inputs such that (the function computed by) $C_n$ cannot be computed by any circuit of size at most $n^c$. By the non-uniform hierarchy theorem (see [1]), there exists such a $C_n$ of size at most $n^{c+1}$ (for $n$ large enough). Let $L$ be the language decided by $\{C_n\}$, and note that we trivially have $L \notin \text{SIZE}(n^c)$.

We claim that $L \in \Sigma_4 \cap \Pi_4$. Indeed, $x \in L$ iff (let $|x| = n$):

1. There exists a circuit $C$ of size at most $n^{c+1}$ such that
2. For all circuits $C'$ (on $n$ inputs) of size at most $n^c$,
   and for all circuits $B$ (on $n$ inputs) lexicographically preceding $C$,
3. There exists an input $x' \in \{0,1\}^n$ such that $C'(x) \neq C(x)$,
   and there exists a circuit $B'$ of size at most $n^c$ such that
4. For all $w \in \{0,1\}^n$ it holds that $B(w) = B'(w)$ and
5. $C(x) = 1$.

Note that that above guesses $C$ and then verifies that $C = C_n$, and finally computes $C(x)$. This shows that $L \in \Sigma_4$, and by flipping the final condition we have that $\bar{L} \in \Sigma_4$. ■

We now "collapse" the above to get the claimed result — non-constructively:

**Corollary 2** *For every $c$, there is a language in $\Sigma_2 \cap \Pi_2$ that is not in $\text{SIZE}(n^c)$.*

**Proof** Say $\mathcal{NP} \not\subseteq \mathcal{P}_{/\mathsf{poly}}$. Then $\mathsf{SAT} \in \mathcal{NP} \subseteq \Sigma_2 \cap \Pi_2$ but $\mathsf{SAT} \notin \text{SIZE}(n^c)$ and we are done. On the other hand, if $\mathcal{NP} \subseteq \mathcal{P}_{/\mathsf{poly}}$ then by the Karp-Lipton theorem $\mathsf{PH} = \Sigma_2 = \Pi_2$ and we may take the language given by the previous theorem. ■

## 1.2 Small Depth Circuits and Parallel Computation

Circuit depth corresponds to the time required for the circuit to be evaluated; this is also evidenced by the proof that $\mathcal{P} \subseteq \mathcal{P}_{/\mathsf{poly}}$. Moreover, a circuit of size $s$ and depth $d$ for some problem can readily be turned into a parallel algorithm for the problem using $s$ processors and running in "wall clock" time $d$. Thus, it is interesting to understand when low-depth circuits for problems exist. From a different point of view, we might expect that *lower bounds* would be easier to prove for low-depth circuits. These considerations motivate the following definitions.

**Definition 1** *Let $i \geq 0$. Then*

- *$L \in \mathsf{NC}^i$ if $L$ is decided by a circuit family $\{C_n\}$ of polynomial size and $O(\log^i n)$ depth over the basis $\mathcal{B}_0$.*
- *$L \in \mathsf{AC}^i$ if $L$ is decided by a circuit family $\{C_n\}$ of polynomial size and $O(\log^i n)$ depth over the basis $\mathcal{B}_1$.*

*$\mathsf{NC} = \bigcup_i \mathsf{NC}^i$ and $\mathsf{AC} = \bigcup_i \mathsf{AC}^i$.*

Note $\mathsf{NC}^i \subseteq \mathsf{AC}^i \subseteq \mathsf{NC}^{i+1}$. Also, $\mathsf{NC}^0$ is not a very interesting class since the function computed by a constant-depth circuit over $\mathcal{B}_0$ can only depend on a constant number of bits of the input.

If we want $\mathsf{NC}$ and $\mathsf{AC}$ to represent feasible algorithms then we need to make sure that the circuit family is *uniform*, i.e., can be computed efficiently. In the case of $\mathsf{NC}$ and $\mathsf{AC}$, the right notion to use is *logspace uniformity*:

**Definition 2** *Circuit family $\{C_n\}$ is logspace-uniform if the function mapping $1^n$ to $C_n$ can be computed using $O(\log n)$ space. Equivalently, each of the following functions can be computed in $O(\log n)$ space:*

- *$\mathsf{size}(1^n)$ returns the number of gates in $C_n$ (expressed in binary). By convention the first $n$ gates are the input gates and the final gate is the output gate.*
- *$\mathsf{type}(1^n, i)$ returns the label (i.e., type of gate) of gate $i$ in $C_n$.*
- *$\mathsf{edge}(1^n, i, j)$ returns 1 iff there is a (directed) edge from gate $i$ to gate $j$ in $C_n$.*

This gives rise to logspace-uniform $\mathsf{NC}^i$, etc., which we sometimes denote by prefixing $u$ (e.g., $u$-$\mathsf{NC}$).

Designing low-depth circuits for problems can be quite challenging. Consider as an example the case of binary addition. The "grade-school" algorithm for addition is inherently *sequential*, and expressing it as a circuit would yield a circuit of linear depth. (In particular, the high-order bit of the output depends on the high-order carry bit, which in the grade-school algorithm is only computed after the second-to-last bit of the output is computed.) Can we do better?

**Lemma 3** *Addition can be computed in logspace-uniform $\mathsf{AC}^0$.*

**Proof** Let $a = a_n \cdots a_1$ and $b = b_n \cdots b_1$ denote the inputs, written so that $a_n, b_n$ are the high-order bits. Let $c_i$ denote the "carry bit" for position $i$, and let $d_i$ denote the $i$th bit of the output. In the "grade-school" algorithm, we set $c_1 = 0$ and then iteratively compute $c_{i+1}$ and $d_i$ from $a_i, b_i$, and $c_i$. However, we can note that $c_{i+1}$ is 1 iff $a_i = b_i = 1$, or $a_{i-1} = b_{i-1} = 1$ (so $c_i = 1$) and at

least one of $a_i$ or $b_i$ is 1, or ..., or $a_1 = b_1 = 1$ and for $j = 2, \ldots, i$ at least one of $a_j$ or $b_j$ is 1. That is,

$$c_{i+1} = \bigvee_{k=1}^{i} (a_k \wedge b_k) \wedge (a_{k+1} \vee b_{k+1}) \cdots \wedge (a_i \vee b_i).$$

So the $\{c_i\}$ can be computed by a constant-depth circuit over $\mathcal{B}_1$. Finally, each bit $d_i$ of the output can be easily computed from $a_i, b_i$, and $c_i$. ■

(Variants of) the circuit given by the previous lemma are used for addition in modern hardware.

There is a close relationship between logarithmic-depth circuits and logarithmic-space algorithms:

**Theorem 4** $u\text{-}\mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq u\text{-}\mathsf{AC}^1$.

**Proof** (Sketch) A logarithmic-space algorithm for any language in logspace-uniform $\mathsf{NC}^1$ follows by recursively computing the values on the wires of a gate's parents, re-using space.

For the second inclusion, we show the more general result that $\text{NSPACE}(s(n))$ can be computed by a circuit family of depth $O(s(n))$ over the unbounded fan-in basis $\mathcal{B}_1$. The idea, once again, is to use reachability. Let $M$ be a non-deterministic machine deciding $L$ in space $t$. Let $N(n) = 2^{O(s(n))}$ denote the number of configurations of $M$ on any fixed input $x$ of length $n$. Fix $n$, let $N = N(n)$, and we will construct $C_n$. On input $x \in \{0,1\}^n$, our circuit does the following:

1. Construct the $N \times N$ adjacency matrix $A_x$ in which entry $(i, j)$ is 1 iff $M$ can make a transition (in one step) from configuration $i$ to configuration $j$ on input $x$.

2. Compute the transitive closure of $A_x$. In particular, this allows us to check whether there is a path from the initial configuration of $M$ (on input $x$) to the accepting configuration of $M$.

We show that these computations can be done in the required depth. The matrix $A_x$ can be computed in *constant* depth, since each entry $(i, j)$ is either always 0, always 1, or else depends on only 1 bit of the input (this is because the input head position is part of a configuration). To compute the transitive closure of $A_x$, we need to compute $(A_x \vee I)^N$. (*Note*: multiplication and addition here correspond to $\wedge$ and $\vee$, respectively.) Using associativity of matrix multiplication, this can be done in a tree-wise fashion using a tree of depth $\log N = O(s(n))$ where each node performs a single matrix multiplication. Matrix multiplication can be performed in constant depth over $\mathcal{B}_1$: to see this, note that the $(i, j)^{\text{th}}$ entry of matrix $AB$ (where $A, B$ are two $N \times N$ matrices given as input) is given by

$$(AB)_{i,j} = \bigvee_{1 \le k \le N} (A_{i,k} \wedge B_{k,j}) .$$

The theorem follows. ■

Can all of $\mathcal{P}$ be parallelized? Equivalently, is $\mathcal{P} = u\text{-}\mathsf{NC}$? To study this question we can, as usual, focus on the "hardest" problems in $\mathcal{P}$:

**Definition 3** *$L$ is $\mathcal{P}$-complete if $\mathsf{L} \in \mathcal{P}$ and every $L' \in \mathcal{P}$ is logspace-reducible to $L$.*

Using Theorem 4 we have

**Claim 5** *If $L$ is $\mathcal{P}$-complete, then $L \in \mathsf{NC}$ iff $\mathcal{P} \subset \mathsf{NC}$.*

An immediate $\mathcal{P}$-complete language is given by

$$\mathsf{CKT\text{-}EVAL} \stackrel{\text{def}}{=} \{(C, x) \mid C(x) = 1\},$$

where a logarithmic-space reduction from any language in $\mathcal{P}$ to CKT-EVAL can be derived from a more careful version of the proof that $\mathcal{P} \subseteq \mathcal{P}_{/\mathsf{poly}}$.

## Bibliographic Notes

The result of Section 1.1 is by Kannan [3]; the presentation here is adapted from [2].

## References


[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[2] S. Jukna. *Boolean Function Complexity: Advances and Frontiers*, Springer, 2012.

[3] R. Kannan. Cicruit-size lower bounds and non-reducibility to sparse sets. *Information and Control* 55(1–3): 40–56, 1982.

**Notes on Complexity Theory** Last updated: October, 2011

# Lecture 12

*Jonathan Katz*

## 1 Randomized Time Complexity

Is deterministic polynomial-time computation the only way to define "feasible" computation? Allowing *probabilistic* algorithms, that may fail with tiny probability, seems reasonable. (In particular, consider an algorithm whose error probability is lower than the probability that there will be a hardware error during the computation, or the probability that the computer will be hit by a meteor during the computation.) This motivates our exploration of probabilistic complexity classes.

There are two different ways to define a randomized model of computation. The first is via Turing machines with a *probabilistic transition function*: as in the case of non-deterministic machines, we have a Turing machine with two transitions functions, and a *random* one is applied at each step. The second way to model randomized computation is by augmenting Turing machines with an additional (read-only) *random tape*. For the latter approach, one can consider either one-way or two-way random tapes; the difference between these models is unimportant for randomized time complexity classes, but (as we will see) becomes important for randomized space classes. Whichever approach we take, we denote by $M(x)$ a random computation of $M$ on input $x$, and by $M(x;r)$ the (deterministic) computation of $M$ on input $x$ using random choices $r$ (where, in the first case, the $i$th bit of $r$ determines which transition function is used at the $i$th step, and in the second case $r$ is the value written on $M$'s random tape).

We now define some randomized time-complexity classes; in the following, PPT stands for "probabilistic, polynomial time" (where this is measured as *worst-case* time complexity over all inputs, and as always the running time is measured as a function of the length of the input $x$).

**Definition 1** *$L \in \mathcal{RP}$ if there exists a* PPT *machine $M$ such that:*

$$\begin{aligned} x \in L &\Rightarrow \Pr[M(x) = 1] \geq 1/2 \\ x \notin L &\Rightarrow \Pr[M(x) = 0] = 1. \end{aligned}$$

Note that if $M(x)$ outputs "1" we are sure that $x \in L$; if $M(x)$ outputs "0" we cannot make any definitive claim.

Viewing $M$ as a non-deterministic machine for $L$, the above means that when $x \in L$ at least half of the computation paths of $M(x)$ accept, and when $x \notin L$ then none of the computation paths of $M(x)$ accept. Put differently, a random tape $r$ for which $M(x;r) = 1$ serves as a witness that $x \in L$. We thus have $\mathcal{RP} \subseteq \mathcal{NP}$.

Symmetrically:

**Definition 2** *$L \in \mathsf{co}\mathcal{RP}$ if there exists a* PPT *machine $M$ such that:*

$$\begin{aligned} x \in L &\Rightarrow \Pr[M(x) = 1] = 1 \\ x \notin L &\Rightarrow \Pr[M(x) = 0] \geq 1/2. \end{aligned}$$

Here, if $M(x)$ outputs "0' we are sure that $x \notin L$, but if $M(x)$ outputs "1" we cannot make any definitive claim.

The above classes allow *one-sided* error. A more general notion of randomized computation allows for *two-sided* error. For a language $L$, let $\chi_L(x) = 1$ iff $x \in L$.

**Definition 3** *$L \in \mathcal{BPP}$ if there exists a* PPT *machine $M$ such that:*

$$\Pr[M(x) = \chi_L(x)] \geq 2/3.$$

*In other words,*

$$\begin{aligned} x \in L &\Rightarrow \Pr[M(x) = 1] \geq 2/3 \\ x \notin L &\Rightarrow \Pr[M(x) = 1] \leq 1/3. \end{aligned}$$

Finally, we may also consider randomized algorithms that make no errors (but may not give a result at all):

**Definition 4** *$L \in \mathcal{ZPP}$ if there exists a* PPT *machine $M$ such that:*

$$\Pr[M(x) = \chi_L(x)] \geq 1/2$$

$$\Pr\left[M(x) \in \{\chi_L(x), \perp\}\right] = 1.$$

We now explore these definitions further. A first observation is that, for all the above definitions, the constants are essentially arbitrary. We focus on the case of $\mathcal{BPP}$ and leave consideration of the rest as an exercise.

**Theorem 1** *The following are both equivalent definitions of $\mathcal{BPP}$:*

1. *$L \in \mathcal{BPP}$ if there exists a* PPT *machine $M$ and a polynomial $p$ such that:*

$$\Pr[M(x) = \chi_L(x)] \geq \frac{1}{2} + \frac{1}{p(|x|)}.$$

2. *$L \in \mathcal{BPP}$ if there exists a* PPT *machine $M$ and a polynomial $q$ such that:*

$$\Pr[M(x) = \chi_L(x)] \geq 1 - 2^{-q(|x|)}.$$

**Proof** We show how to transform an algorithm $M$ satisfying the first definition into an algorithm $M'$ satisfying the second definition. $M'(x)$ is defined as follows: run $M(x)$ a total of $t(|x|)$ times (for some polynomial $t$ to be fixed later) using independent random coins in each execution. Then $M'$ outputs the bit that was output by a majority of these executions.

To analyze the behavior of $M'$, we rely on the *Chernoff bound* [**?**, Chap. 4]:

**Claim 2** *Let $p \leq \frac{1}{2}$ and let $X_1, \ldots, X_n$ be independent, identically-distributed 0-1 random variables with $\Pr[X_i = 1] = p$ for each $i$. Then for all $\varepsilon$ with $0 < \varepsilon \leq p(1-p)$ we have:*

$$\Pr\left[\left|\frac{\sum_{i=1}^n X_i}{n} - p\right| > \varepsilon\right] < 2 \cdot e^{-\frac{\varepsilon^2 n}{2p(1-p)}} \leq 2 \cdot e^{-2n\varepsilon^2}.$$

Let $X_i$ denote the output of the $i^{\text{th}}$ execution of $M(x)$. When $x \notin L$

$$\Pr[X_i = 1] < \frac{1}{2} - \frac{1}{p(|x|)} \stackrel{\text{def}}{=} \rho.$$

Furthermore, by definition of $M'$ (letting $t \stackrel{\text{def}}{=} t(|x|)$):

$$\begin{aligned} \Pr[M'(x) = 1] &= \Pr\left[\frac{\sum_{i=1}^{t} X_i}{t} > \frac{1}{2}\right] \\ &\leq \Pr\left[\left|\frac{\sum_{i=1}^{t} X_i}{t} - \rho\right| > \frac{1}{p(|x|)}\right] \\ &< 2 \cdot e^{-\frac{2t}{p(|x|)^2}}. \end{aligned}$$

Setting $t = O\left(q(|x|) \cdot p(|x|)^2\right)$ gives the desired result. (An exactly analogous argument works for the case $x \in L$.) ■

How do the above classes relate to each other? It is immediate that

$$\mathcal{RP} \cup \mathsf{co}\mathcal{RP} \subseteq \mathcal{BPP},$$

and so $\mathcal{BPP}$ is a (potentially) more powerful class. Indeed, $\mathcal{BPP}$ appears to capture feasible probabilistic computation. We also claim that

$$\mathcal{ZPP} = \mathcal{RP} \cap \mathsf{co}\mathcal{RP};$$

this is left as an exercise. A third characterization of $\mathcal{ZPP}$ is in terms of expected polynomial-time algorithms that always output the correct answer. Let $M$ be a probabilistic Turing machine. We say that $M$ runs in expected time $t(n)$ if, for every $x \in \{0,1\}^n$, the expected running time of $M(x)$ is at most $t(n)$. Then:

**Claim 3** *$L \in \mathcal{ZPP}$ iff there exists an expected polynomial-time Turing machine $M$ such that*

$$\Pr[M(x) = \chi_L(x)] = 1.$$

How about a "minimal" notion of correctness for probabilistic algorithms, where we only require correctness with probability arbitrarily better than guessing? This gives rise to a class called $\mathcal{PP}$:

**Definition 5** *$L \in \mathcal{PP}$ if there exists a* PPT *machine $M$ such that:*

$$\Pr[M(x) = \chi_L(x)] > 1/2.$$

*In other words,*

$$\begin{aligned} x \in L &\Rightarrow \Pr[M(x) = 1] > 1/2 \\ x \notin L &\Rightarrow \Pr[M(x) = 1] < 1/2. \end{aligned}$$

A little thought shows that this is *not* a reasonable notion of probabilistic computation. The problem is that the gap between outputting the correct answer and the wrong answer might be exponentially small (in contrast to $\mathcal{BPP}$, where the gap must be some inverse polynomial); in particular, amplification does not work here. As some further evidence against the reasonableness of $\mathcal{PP}$, we have $\mathcal{NP} \subseteq \mathcal{PP}$ (this, too, is left as an exercise); thus, this notion of probabilistic computation can solve all of $\mathcal{NP}$!

## 1.1 Examples of Randomized Algorithms

There are several examples of where randomized algorithms are more efficient, or simpler, than known deterministic algorithms. However, there are not as many examples of problems that are known to be solvable by polynomial-time randomized algorithms, but not known to be solved by polynomial-time deterministic algorithms. One famous former example was testing primality: this problem was known to be in $\mathsf{co}\mathcal{RP}$ since the late 1970s, but was only shown to be in $\mathcal{P}$ in 2005. (Nevertheless, in practice the randomized algorithms are still used since they are faster.)

A search problem for which probabilistic polynomial-time algorithms are known, but deterministic polynomial-time algorithms are not, is computing square roots modulo a prime.

**Polynomial identity testing.** Another interesting example is given by the problem of testing equality of *arithmetic* circuits. Here we work with circuits that take integers (rather than boolean values) as input, and where gates compute $+$, $-$, and $\times$ (over the integers); the output is an integer. Say we want to test whether two circuits $C_1, C_2$ compute the same function. Note that this easily reduces to deciding the following language:

$$\mathsf{ZEROP} \stackrel{\text{def}}{=} \{C \mid C \text{ outputs 0 on all inputs}\}.$$

Any arithmetic circuit is equivalent to a multivariate polynomial over the integers; in principle, then, we can decide membership in $\mathsf{ZEROP}$ by expanding and writing out the polynomial to see whether it is identically 0. (This explains the name $\mathsf{ZEROP}$ for the language above: we are testing whether an implicitly defined polynomial is the 0 polynomial.) In an arithmetic circuit with $m$ gates, however, the (total) degree[1] of the equivalent polynomial can be as high as $2^m$, and so even just writing out all the terms of the polynomial may require exponential time! This is therefore not a viable approach for an efficient algorithm.

In fact, there is no known (efficient) deterministic algorithm for this problem. Instead, we make use of the Schwartz-Zippel lemma (which is useful in many contexts). We state it here for polynomials over the integers, but it also holds over any field.

**Lemma 4** *Let $p(X_1, \ldots, X_n)$ be a non-zero polynomial of total degree at most $d$, and let $S$ be any finite set of integers. Then*

$$\Pr_{x_1,\ldots,x_n \leftarrow S}[p(x_1, \ldots, x_n) = 0] \leq d/|S|.$$

**Proof** The proof is by induction on $n$. When $n = 1$, a non-zero polynomial $p(X_1)$ of degree at most $d$ has at most $d$ roots and the lemma follows. Now assume the lemma is true for $n-1$ and prove that it holds for $n$. Given a polynomial $p(X_1, \ldots, X_n)$ of total degree $d$, we may write

$$p(X_1, \ldots, X_n) = \sum_{i=0}^{d'} p_i(X_1, \ldots, X_{n-1}) \cdot X_n^i, \tag{1}$$

for some $d' \leq d$ and $p_{d'}(X_1, \ldots, X_{n-1})$ a non-zero polynomial of total degree at most $d - d'$. When $x_1, \ldots, x_{n-1}$ are chosen at random from $S$, the inductive assumption tells us that $p_{d'}(x_1, \ldots, x_{n-1}) = 0$ with probability at most $(d - d')/|S|$. When $p_{d'}(x_1, \ldots, x_{n-1}) \neq 0$, then (1) is a polynomial of

[1] The total degree of a monomial is the sum of the degrees in each variable; the total degree of a multivariate polynomial largest total degree of any monomial.

degree $d'$ in the single variable $X_n$, and so the probability (over random choice of $x_n \in S$) that $p(x_1, \ldots, x_{n-1}, x_n) = 0$ is at most $d'/|S|$. Putting everything together, we have

$$\begin{aligned} \Pr_{x_1,\ldots,x_n \leftarrow S}[p(x_1, \ldots, x_n) = 0] &\leq \frac{d-d'}{|S|} + \left(1 - \frac{d-d'}{|S|}\right) \cdot \frac{d'}{|S|} \\ &\leq \frac{d}{|S|}. \end{aligned}$$

This completes the proof. ■

The above suggests a simple randomized algorithm: given an arithmetic circuit $C$ with $m$ gates, and $n$ inputs $X_1, \ldots, X_n$, choose $x_1, \ldots, x_n \leftarrow \{1, \ldots, 2^{m+1}\}$ and evaluate $C(x_1, \ldots, x_n)$. If the output is 0, accept; if the output is non-zero, reject. If $C \in \mathsf{ZEROP}$, then this algorithm always accepts, while if $C \notin \mathsf{ZEROP}$ then the algorithm rejects with probability at least $1/2$. (This is thus a $\mathsf{co}\mathcal{RP}$ algorithm for $\mathsf{ZEROP}$.)

There is, however, a problem with the algorithm: as written, it is not efficient. The difficulty is that the value of $C(2^{m+1}, \ldots, 2^{m+1})$ may be as high as $(2^{m+1})^{2^m}$, which would require exponentially many bits to write down. We can solve this by "fingerprinting"; see [1] for details.

**Perfect matching in bipartite graphs.** We can use similar ideas to give an efficient randomized algorithm for detecting the existence of a perfect matching in a bipartite graph. (Although this problem is in $\mathcal{P}$, the randomized algorithm we show can be implemented in randomized-$\mathsf{NC}$.) Let $G$ be an $n \times n$ matrix representing a bipartite graph on $2n$ vertices, where $G_{i,j} = X_{i,j}$ if there is an edge from $i$ to $j$, and $G_{i,j} = 0$ otherwise. The determinant of $G$ is

$$\det(G) = \sum_{\sigma} (-1)^{\mathrm{sign}(\sigma)} \prod_{i=1}^{n} G_{i,\sigma(i)},$$

and we see that $\det(G)$ is a non-zero polynomial (in the variables $X_{1,1}, \ldots, X_{n,n}$) iff the underlying graph has a perfect matching. Calculating the polynomial $\det(G)$ cannot be done efficiently, since it may have exponentially many terms; however, we can evaluate $\det(G)$ for any given values o using standard algorithms for computing the determinant. We can thus use the Shwartz-Zippel lemma and the ideas seen previously to construct a randomized algorithm for this problem.

## References


[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

**Notes on Complexity Theory** Last updated: October, 2011

Lecture 13

*Jonathan Katz*

# 1 Randomized Time Complexity

## 1.1 How Large is $\mathcal{BPP}$?

We know that

$$\mathcal{P} \subseteq \mathcal{ZPP} = \mathcal{RP} \cap \mathsf{co}\mathcal{RP} \subseteq \mathcal{RP} \cup \mathsf{co}\mathcal{RP} \subseteq \mathcal{BPP} \subseteq \mathsf{PSPACE}.$$

We currently do not have a very good unconditional bound on the power of $\mathcal{BPP}$ — in particular, it could be that $\mathcal{BPP} = \mathsf{NEXP}$. Perhaps surprisingly, especially in light of the many randomized algorithms known, the current conjecture is that $\mathcal{BPP}$ is *not* more powerful than $\mathcal{P}$. We will return to this point later in the semester when we talk about derandomization.

What (unconditional) upper bounds *can* we place on $\mathcal{BPP}$? Interestingly, we know that it is not more powerful than polynomial-size circuits; actually, the following theorem is also a good illustration of the power of non-uniformity.

**Theorem 1** $\mathcal{BPP} \subset \mathcal{P}/\mathsf{poly}$.

**Proof** Let $L \in \mathcal{BPP}$. Using amplification, we know that there exists a polynomial-time Turing machine $M$ such that $\Pr[M(x) \neq \chi_L(x)] < 2^{-|x|^2}$. Say $M$ uses (at most) $p(|x|)$ random coins for some polynomial $p$. (Note that $p$ is upper-bounded by the running time of $M$.) An equivalent way of stating this is that for each $n$, and each $x \in \{0,1\}^n$, the set of "bad" coins for $x$ (i.e., coins for which $M(x)$ outputs the wrong answer) has size at most $2^{p(n)} \cdot 2^{-n^2}$. Taking the union of these "bad" sets over all $x \in \{0,1\}^n$, we find that the total number of random coins which are "bad" for *some* $x$ is at most $2^{p(n)} \cdot 2^{-n} < 2^{p(n)}$. In particular, there exists at least one set of random coins $r_n^* \in \{0,1\}^{p(n)}$ that is "good" for *every* $x \in \{0,1\}^n$ (in fact, there are many such random coins). If we let the sequence of "advice strings" be exactly $\{r_n^*\}$ (using the alternate definition of $\mathcal{P}_{/\mathsf{poly}}$), we obtain the result of the theorem. ■

We can also place $\mathcal{BPP}$ in the polynomial hierarchy:

**Theorem 2** $\mathcal{BPP} \subseteq \Sigma_2 \cap \Pi_2$.

**Proof** We show that $\mathcal{BPP} \subseteq \Sigma_2$; since $\mathcal{BPP}$ is closed under complement, this proves the theorem.

We begin by proving some probabilistic lemmas. Say $S \subseteq \{0,1\}^m$ is *large* if $|S| \geq (1 - \frac{1}{m})2^m$, and is *small* if $|S| < \frac{2^m}{m}$. For a string $z \in \{0,1\}^m$ define $S \oplus z \stackrel{\text{def}}{=} \{s \oplus z \mid s \in S\}$.

**Claim 3** *If $S$ is small, then for all $z_1, \ldots, z_m \in \{0,1\}^m$ we have $\bigcup_i (S \oplus z_i) \neq \{0,1\}^m$.*

This follows easily since

$$\left|\bigcup_i (S \oplus z_i)\right| \leq \sum_i |S \oplus z_i| = m \cdot |S| < 2^m.$$

**Claim 4** *If $S$ is large, then there exist $z_1, \ldots, z_m \in \{0,1\}^m$ such that $\bigcup_i (S \oplus z_i) = \{0,1\}^m$.*

In fact, we show that choosing at random works with high probability; i.e.,

$$\Pr_{z_1,\ldots,z_m \in \{0,1\}^m} \left[ \bigcup_i (S \oplus z_i) = \{0,1\}^m \right] \geq 1 - \left(\frac{2}{m}\right)^m.$$

To see this, consider the probability that some fixed $y$ is not in $\bigcup_i (S \oplus z_i)$. This is given by:

$$\begin{aligned} \Pr_{z_1,\ldots,z_m \in \{0,1\}^m} [y \notin \textstyle\bigcup_i (S \oplus z_i)] &= \textstyle\prod_i \Pr_{z \in \{0,1\}^m} [y \notin (S \oplus z)] \\ &\leq \left(\frac{1}{m}\right)^m. \end{aligned}$$

Applying a union bound by summing over all $y \in \{0,1\}^m$, we see that the probability that there *exists* a $y \in \{0,1\}^m$ which is not in $\bigcup_i (S \oplus z_i)$ is at most $\frac{2^m}{m^m}$.

We now prove the theorem. Given $L \in \mathcal{BPP}$, there exist a polynomial $m$ and an algorithm $M$ such that $M$ uses $m(|x|)$ random coins and errs with probability less than $1/m$. For any input $x$, let $S_x \subseteq \{0,1\}^{m(|x|)}$ denote the set of random coins for which $M(x;r)$ outputs 1. Thus, if $x \in L$ (letting $m = m(|x|)$) we have $|S_x| > (1 - \frac{1}{m}) \cdot 2^m$ while if $x \notin L$ then $|S_x| < \frac{2^m}{m}$. This leads to the following $\Sigma_2$ characterization of $L$:

$$x \in L \Leftrightarrow \exists z_1, \ldots, z_m \in \{0,1\}^m \; \forall y \in \{0,1\}^m : y \in \textstyle\bigcup_i (S_x \oplus z_i).$$

(Note the desired condition can be efficiently verified by checking if $M(x; y \oplus z_i) \stackrel{?}{=} 1$ for some $i$.) ■

## 1.2 Complete Problems for $\mathcal{BPP}$?

As usual, we might like to study a class by focusing on the "hardest" problems in that class. With this in mind, we can ask whether $\mathcal{BPP}$ has any complete problems. The obvious thought is to consider the following language:

$$\left\{ (M, x, 1^p) \mid \begin{array}{c} M \text{ is a probabilistic machine that accepts } x \\ \text{with probability at least } 2/3 \text{ within } p \text{ steps} \end{array} \right\}.$$

While this language is $\mathcal{BPP}$-hard, it is not known to be in $\mathcal{BPP}$! (Consider the case when $\Pr[M(x) = 1] = 2/3 - 2^{-|x|}$.)

We can address this issue using the notion of *promise problems*, which gives an alternative to languages as a way to define complexity classes. A promise problem consists of two disjoint sets of strings $\Pi_Y, \Pi_N \subseteq \{0,1\}^*$ with $\Pi_Y \cap \Pi_N = \emptyset$. The "promise" is that all inputs will be from $\Pi_Y \cup \Pi_N$, and we only need to "solve" the problem on such inputs; in particular, we do not care what happens if we get an input that is not in $\Pi_Y \cup \Pi_N$. Thus, using this formulation, promise-$\mathcal{P}$ would be defined as the class of promise problems $(\Pi_Y, \Pi_N)$ for which there exists a polynomial-time machine $M$ such that

$$\begin{aligned} x \in \Pi_Y &\Rightarrow M(x) = 1] \\ x \in \Pi_N &\Rightarrow M(x) = 0. \end{aligned}$$

Promise problems generalize languages, since we may view a language $L$ equivalently as the promise problem $(L,\ \{0,1\}^* \setminus L)$.

We can define the class promise-$\mathcal{BPP}$ as the class of promise problems $(\Pi_Y, \Pi_N)$ for which there exists a probabilistic polynomial-time machine $M$ such that

$$\begin{aligned} x \in \Pi_Y &\Rightarrow \Pr[M(x) = 1] \geq 2/3 \\ x \in \Pi_N &\Rightarrow \Pr[M(x) = 1] \leq 1/3. \end{aligned}$$

We don't care about the behavior of $M$ on inputs not in $(\Pi_Y, \Pi_N)$ — it might always accept, or accept some inputs but not others, or accept with arbitrary probability.

Arguably, promise problems are more natural than languages. (For example, we might speak of the input as representing an undirected graph and then need to ensure that every string encodes *some* undirected graph; it would be more natural to simply restrict our attention to strings that canonically represent undirected graphs.) And, indeed, promise-$\mathcal{BPP}$ does have a complete language:

$$\begin{aligned} \Pi_Y &= \left\{ (M, x, 1^p) \mid \begin{array}{c} M \text{ is a probabilistic machine that accepts } x \\ \text{with probability at least } 2/3 \text{ within } p \text{ steps} \end{array} \right\} \\ \Pi_N &= \left\{ (M, x, 1^p) \mid \begin{array}{c} M \text{ is a probabilistic machine that rejects } x \\ \text{with probability at least } 2/3 \text{ within } p \text{ steps} \end{array} \right\}. \end{aligned}$$

## 2 Randomized Space Complexity

When defining randomized space-complexity classes there are two subtleties to be aware of. The first subtlety arises if we model probabilistic computation by a Turing machine having a read-only random tape. (The issue does not come up if we instead view a probabilistic machine as a non-deterministic machine where the transition function is chosen at random.) Here, as in the case of the certificate-based definition of non-deterministic space complexity, we need to restrict the machine to having "read-once" access to its tape.

A second issue (regardless of which formulation of probabilistic computation we use) is that we must also impose a time bound on the machine.[1] E.g., for the case of one-sided error:

**Definition 1** *A language $L$ is in* $\text{RSPACE}(s(n))$ *if there exists a randomized Turing machine $M$ using $s(n)$ space* **and** $2^{O(s(n))}$ **time** *such that*

$$x \in L \Rightarrow \Pr[M(x) = 1] \geq 1/2 \quad \textit{and} \quad x \notin L \Rightarrow \Pr[M(x) = 0] = 1.$$

If we do not impose this restriction, then probabilistic space classes become too powerful:

**Proposition 5** *Define* $\text{RSPACE}'$ *as above, but without the time restriction. Then for any space-constructible $s(n) \geq \log n$ we have* $\text{RSPACE}'(s(n)) = \text{NSPACE}(s(n))$.

[1] An equivalent condition is to require that the machine halts for every possible set of random choices. This is different from requiring the machine to halt with probability 1; see the proof of Proposition 5 for an illustration of this phenomenon.

**Proof** (Sketch) Showing that $\text{RSPACE}'(s(n)) \subseteq \text{NSPACE}(s(n))$ is easy. We turn to the other direction. The basic idea is that, given a language $L \in \text{NSPACE}(s(n))$, we construct a machine which on input $x$ guesses valid witnesses for $x$ (where a witness here is an accepting computation of the non-deterministic machine on input $x$). Since there may only be a single witness, we guess a doubly-exponential number of times. This is where the absence of a time bound makes a difference.

In more detail, given $L$ as above we know that any $x \in L \cap \{0,1\}^n$ has a witness (i.e., an accepting computation) of length at most $\ell(n) = 2^{O(s(n))}$. If we happen to have such a witness written on the random tape, we can verify its correctness using space $O(s(n))$. So what we do is the following: alternately (1) read the next $\ell(n)$ bits of the random tape and check whether it encodes a witness, (2) read the next $\ell(n)$ bits of the random tape and halt if they are all 0. If $x \notin L$ this machine never accepts; if $x \in L$ then it accepts with probability $1/2$. Note that $M$ may run for an unbounded amount of time, however it halts on all inputs with probability 1. ■

The most interesting probabilistic space classes are those where logarithmic space is used:

**Definition 2** *$L \in \mathcal{RL}$ if there is a machine $M$ using logarithmic space and running in polynomial time such that:*

$$x \in L \Rightarrow \Pr[M(x) = 1] \geq 1/2 \quad \textit{and} \quad x \notin L \Rightarrow \Pr[M(x) = 0] = 1.$$

*$L \in \mathcal{BPL}$ if there is a machine $M$ as above such that:*

$$x \in L \Rightarrow \Pr[M(x) = 1] \geq 2/3 \quad \textit{and} \quad x \notin L \Rightarrow \Pr[M(x) = 1] \leq 1/3.$$

Here, too, the exact constants are arbitrary as error reduction still works. (We need only to maintain a counter of the fraction of accepting executions.) It is immediate that $\mathcal{RL} \subseteq \mathsf{NL}$. One can also show that $\mathcal{BPL} \subseteq \text{SPACE}(\log^2(n))$ and $\mathcal{BPL} \subseteq \mathcal{P}$.

## Bibliographic Notes

For an in-depth discussion of promise problems, and arguments in favor of taking that approach, see the survey by Goldreich [1].

## References

[1] O. Goldreich. On promise problems. Manuscript available on-line at `http://www.wisdom.weizmann.ac.il/~oded/prpr.html`

**Notes on Complexity Theory** Last updated: October, 2011

Lecture 14

*Jonathan Katz*

# 1 Randomized Space Complexity

## 1.1 Undirected Connectivity and Random Walks

A classic problem in $\mathcal{RL}$ is *undirected connectivity* (UConn). Here, we are given an *undirected* graph and two vertices $s, t$ and are asked to determine whether there is a path from $s$ to $t$. An $\mathcal{RL}$ algorithm for this problem is simply to take a "random walk" (of sufficient length) in the graph, starting from $s$. If vertex $t$ is ever reached, then output 1; otherwise, output 0. (We remark that this approach does *not* work for *directed* graphs.) We analyze this algorithm (and, specifically, the length of the random walk needed) in two ways; each illustrates a method that is independently useful in other contexts. The first method looks at random walks on *regular* graphs, and proves a stronger result showing that after sufficiently many steps of a random walk the location is close to uniform over the vertices of the graph. The second method is more general, in that it applies to any (non-bipartite) graph; it also gives a tighter bound.

### 1.1.1 Random Walks on Regular Graphs

Fix an undirected graph $G$ on $n$ vertices where we allow self-loops and parallel edges (i.e., integer weights on the edges). We will assume the graph is $d$-regular and has at least one self-loop at every vertex; any graph can be changed to satisfy these conditions (without changing its connectivity) by adding sufficiently many self-loops. Let $G$ also denote the (scaled) adjacency matrix corresponding to this graph: the $(i, j)$th entry is $k/d$ if there are $k$ edges between vertices $i$ and $j$. Note that $G$ is *symmetric* ($G_{i,j} = G_{j,i}$ for all $i, j$) and *doubly stochastic* (all entries are non-negative, and all rows and columns sum to 1). A *probability vector* $\mathbf{p} = (p_1, \ldots, p_n) \in \mathbb{R}^n$ is a vector each of whose entries is non-negative and such that $\sum_i p_i = 1$. If we begin by choosing a vertex $v$ of $G$ with probability determined by $\mathbf{p}$, and then take a "random step" by choosing (uniformly) an edge of $v$ and moving to the vertex $v'$ adjacent to that edge, the resulting distribution on $v'$ is given by $\mathbf{p}' = G \cdot \mathbf{p}$. Inductively, the distribution after $t$ steps is given by $G^t \cdot \mathbf{p}$. Note that if we set $\mathbf{p} = \mathbf{e}_i$ (i.e., the vector with a 1 in the $i$th position and 0s everywhere else), then $G^t \cdot \mathbf{p}$ gives the distribution on the location of a $t$-step random walk starting at vertex $i$.

An *eigenvector* of a matrix $G$ is a vector $\mathbf{v}$ such that $G \cdot \mathbf{v} = \lambda \mathbf{v}$ for some $\lambda \in \mathbb{R}$; in this case we call $\lambda$ the associated *eigenvalue*. Since $G$ is a symmetric matrix, standard results from linear algebra show that there is an orthonormal basis of eigenvectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$ with (real) eigenvalues $\lambda_1, \ldots, \lambda_n$, sorted so that $|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n|$. If we let $\mathbf{1}$ denote the vector with $1/n$ in each entry — i.e., it represents the uniform distribution over the vertices of $G$ — then $G \cdot \mathbf{1} = \mathbf{1}$ and so $G$ has eigenvalue 1. Moreover, since $G$ is a (doubly) stochastic matrix, it has no eigenvalues of absolute value greater than 1. Indeed, let $\mathbf{v} = (v_1, \ldots, v_n)$ be an eigenvector of $G$ with eigenvalue $\lambda$, and let $j$ be such that $|v_j|$ is maximized. Then $\lambda \mathbf{v} = G \cdot \mathbf{v}$ and so

$$|\lambda v_j| \;=\; \left|\textstyle\sum_{i=1}^{n} G_{j,i} \cdot v_i\right|$$

$$\leq \quad |v_j| \cdot \textstyle\sum_{i=1}^{n} |G_{j,i}| \;=\; |v_j|\,;$$

we conclude that $|\lambda| \leq 1$. If $G$ is connected, then it has no other eigenvector with eigenvalue 1. Since $G$ is non-bipartite (because of the self-loops), $-1$ is not an eigenvalue either.

To summarize, if $G$ is connected and not bipartite then it has (real) eigenvectors $\lambda_1, \ldots, \lambda_n$ with $1 = \lambda_1 > |\lambda_2| \geq \cdots \geq |\lambda_n|$. The (absolute value of the) second eigenvalue $\lambda_2$ determines how long a random walk in $G$ we need so that the distribution of the final location is close to uniform:

**Theorem 1** *Let $G$ be a $d$-regular, undirected graph on $n$ vertices with second eigenvalue $\lambda_2$, and let $\mathbf{p}$ correspond to an arbitrary probability distribution over the vertices of $G$. Then for any $t > 0$*

$$\left\| G^t \cdot \mathbf{p} - \mathbf{1} \right\|_2 \leq |\lambda_2|^t.$$

**Proof** Write $\mathbf{p} = \sum_{i=1}^{n} \alpha_i \mathbf{v}_i$, where the $\{\mathbf{v}_i\}$ are the eigenvectors of $G$ (sorted according to decreasing absolute value of their eigenvalues); recall $\mathbf{v}_1 = \mathbf{1}$. We have $\alpha_1 = 1$; this follows since $\alpha_1 = \langle \mathbf{p}, \mathbf{1} \rangle \,/\, \|\mathbf{1}\|_2^2 = (1/n)/(1/n) = 1$. We thus have

$$G^t \cdot \mathbf{p} = G^t \cdot \mathbf{1} + \sum_{i=2}^{n} \alpha_i G^t \cdot \mathbf{v}_i = \mathbf{1} + \sum_{i=2}^{n} \alpha_i \left(\lambda_i\right)^t \mathbf{v}_i$$

and so, using the fact that the $\{\mathbf{v}_i\}$ are orthogonal,

$$\begin{aligned} \left\| G^t \cdot \mathbf{p} - \mathbf{1} \right\|_2^2 &= \sum_{i=2}^{n} \alpha_i^2 \left(\lambda_i\right)^{2t} \cdot \|\mathbf{v}_i\|_2^2 \\ &\leq \lambda_2^{2t} \cdot \sum_{i=2}^{n} \alpha_i^2 \cdot \|\mathbf{v}_i\|_2^2 \\ &\leq \lambda_2^{2t} \cdot \|\mathbf{p}\|_2^2 \;\leq\; \lambda_2^{2t} \cdot \|\mathbf{p}\|_1^2 = \lambda_2^{2t}. \end{aligned}$$

The theorem follows. ■

It remains to show a bound on $|\lambda_2|$.

**Theorem 2** *Let $G$ be a $d$-regular, connected, undirected graph on $n$ vertices with at least one self-loop at each vertex and $d \leq n$. Then $|\lambda_2| \leq 1 - \frac{1}{\mathsf{poly}(n)}$.*

**Proof** Let $\mathbf{u} = (u_1, \ldots, u_n)$ be a unit eigenvector corresponding to $\lambda_2$, and recall that $\mathbf{u}$ is orthogonal to $\mathbf{1} = (1/n, \ldots, 1/n)$. Let $\mathbf{v} = G\mathbf{u} = \lambda_2 \mathbf{u}$. We have

$$\begin{aligned} 1 - \lambda_2^2 = \|\mathbf{u}\|_2^2 \cdot (1 - \lambda_2^2) &= \|\mathbf{u}\|_2^2 - \|\mathbf{v}\|_2^2 \\ &= \|\mathbf{u}\|_2^2 - 2\,\|\mathbf{v}\|_2^2 + \|\mathbf{v}\|_2^2 \\ &= \|\mathbf{u}\|_2^2 - 2\,\langle G\mathbf{u}, \mathbf{v} \rangle + \|\mathbf{v}\|_2^2 \\ &= \sum_i u_i^2 - 2 \sum_{i,j} G_{i,j}\, u_j v_i + \sum_j v_j^2 \\ &= \sum_{i,j} G_{i,j}\, u_i^2 - 2 \sum_{i,j} G_{i,j}\, u_j v_i + \sum_{i,j} G_{i,j}\, v_j^2 \\ &= \sum_{i,j} G_{i,j} \left(u_i - v_j\right)^2, \end{aligned}$$

using the fact that $G$ is a symmetric, doubly stochastic matrix for the second-to-last equality. Since $\mathbf{u}$ is a unit vector orthogonal to $\mathbf{1}$, there exist $i, j$ with $u_i > 0 > u_j$ and such that at least one of $u_i$ or $u_j$ has absolute value at least $1/\sqrt{n}$, meaning that $u_i - u_j \geq 1/\sqrt{n}$. Since $G$ is connected, there is a path of length $D$, say, between vertices $i$ and $j$. Renumbering as necessary, let $i = 1$, $j = D + 1$, and let the vertices on the path be $2, \ldots, D$. Then

$$\begin{aligned} \frac{1}{\sqrt{n}} \leq u_1 - u_{D+1} &= (u_1 - v_1) + (v_1 - u_2) + (u_2 - v_2) + (v_2 - u_3) + \cdots + (v_D - u_{D+1}) \\ &\leq |u_1 - v_1| + \cdots + |v_D - u_{D+1}| \\ &\leq \sqrt{(u_1 - v_1)^2 + \cdots + (v_D - u_{D+1})^2} \cdot \sqrt{2D} \end{aligned}$$

(using Cauchy-Schwarz for the last inequality). But then

$$\sum_{i,j} G_{i,j}(u_i - v_j)^2 \geq \frac{1}{d} \cdot \left((u_1 - v_1)^2 + \cdots + (v_D - u_{D+1})^2\right) \geq \frac{1}{2dnD},$$

using $G_{i,i} \geq 1/d$ (since every vertex has a self-loop) and $G_{i,i+1} \geq 1/d$ (since there is an edge from vertex $i$ to vertex $i + 1$). Since $D \leq n - 1$, we get $1 - \lambda_2^2 \geq 1/4dn^2$ or $|\lambda_2| \leq 1 - 1/8dn^2$, and the theorem follows. ■

We can now analyze the algorithm for undirected connectivity. Let us first specify the algorithm more precisely. Given an undirected graph $G$ and vertices $s, t$, we want to determine if there is a path from $s$ to $t$. We restrict our attention to the connected component of $G$ containing $s$, add at least one self-loop to each vertex in $G$, and add sufficiently many additional self-loops to each vertex in order to ensure regularity. Then we take a random walk of length $\ell = 16dn^2 \log n \geq 2 \cdot (1 - |\lambda_2|)^{-1} \log n$ starting at vertex $s$, and output 1 if we are at vertex $t$ at the end of the walk. (Of course, we do better if we output 1 if the walk ever passes through vertex $t$; our analysis does not take this into account.) By Theorem 1,

$$\left\| G^\ell \cdot \mathbf{e}_s - \mathbf{1} \right\|_2 \leq |\lambda_2|^\ell \leq 1/n^2.$$

If $t$ is in the connected component of $s$, the probability that we are at vertex $t$ at the end of the walk is at least $\frac{1}{n} - \frac{1}{n^2} \geq 1/2n$. We can, of course, amplify this by repeating the random walk sufficiently many times.

**Notes on Complexity Theory** Last updated: October, 2011

Lecture 15

*Jonathan Katz*

# 1 Randomized Space Complexity

## 1.1 Undirected Connectivity and Random Walks

### 1.1.1 Markov Chains

We now develop some machinery that gives a different, and somewhat more general, perspective on random walks. In addition, we get better bounds for the probability that we hit $t$. (Note that the previous analysis calculated the probability that we end at vertex $t$. But it would be sufficient to pass through vertex $t$ at any point along the walk.) The drawback is that here we rely on some fundamental results concerning Markov chains that are presented without proof.

We begin with a brief introduction to (finite, time-homogeneous) Markov chains. A sequence of random variables $X_0, \ldots$ over a space $\Omega$ of size $n$ is a *Markov chain* if there exist $\{p_{i,j}\}$ such that, for all $t > 0$ and $x_0, \ldots, x_{t-2}, x_i, x_j \in \Omega$ we have:

$$\Pr[X_t = x_j \mid X_0 = x_0, \ldots, X_{t-2} = x_{t-2}, X_{t-1} = x_i] \;=\; \Pr[X_t = x_j \mid X_{t-1} = x_i] \;=\; p_{j,i}.$$

In other words, $X_t$ depends only on $X_{t-1}$ (that is, the transition is *memoryless*) and is furthermore independent of $t$. We view $X_t$ as the "state" of a system at time $t$. If we have a probability distribution over the states of the system at time $t$, represented by a probability vector $\mathbf{p}_t$, then the distribution at time $t+1$ is given by $P \cdot \mathbf{p}_t$ (similar to what we have seen in the previous section). Similarly, the probability distribution at time $t + \ell$ is given by $P^\ell \cdot \mathbf{p}_t$.

A finite Markov chain corresponds in the natural way to a random walk on a (possibly directed and/or weighted) graph. Focusing on undirected graphs (which is all we will ultimately be interested in), a random walk on such a graph proceeds as follows: if we are at a vertex $v$ at time $t$, we move to a random neighbor of $v$ at time $t+1$. If the graph has $n$ vertices, such a random walk defines the Markov chain given by:

$$p_{j,i} = \begin{cases} k/\deg(i) & \text{there are } k \text{ edges between } j \text{ and } i \\ 0 & \text{otherwise} \end{cases}.$$

We continue to allow (multiple) self-loops; each self-loop contributes 1 to the degree of a vertex.

Let $\mathbf{p}$ be a probability distribution over the states of the system. We say $\mathbf{p}$ is *stationary* if $P \cdot \mathbf{p} = \mathbf{p}$. We have the following fundamental theorem of random walks on undirected graphs (which is a corollary of a more general result for Markov chains):

**Theorem 1** *Let $G$ be an undirected, connected, non-bipartite graph on $n$ vertices, and consider the transition matrix corresponding to a random walk on $G$. Then:*

1. *There is a unique stationary distribution $\mathbf{p} = (p_1, \ldots, p_n)$.*

2. *Let $h_{i,i}$ denote the expected number of steps for a random walk beginning at vertex $i$ to return to $i$. Then $h_{i,i} = 1/p_i$.*

In particular, the graph need not be regular.

We do not prove Theorem 1 here. (A proof of the first claim, and intuition for the second claim can be found in [1, Lecture 8] or dedicated texts on Markov chains, e.g., [2].) Note that for any undirected graph $G$, the conditions of the theorem can always be met by (1) restricting attention to a connected component of $G$, and (2) adding a self-loop to any vertex in the connected component.

What is the stationary distribution for a given graph? Say we have an undirected, connected, non-bipartite graph $G$ with $m$ edges and $\ell$ self-loops. It can be verified by a simple calculation that setting $p_i = \frac{\deg(i)}{2m+\ell}$ for each vertex $i$ gives a stationary distribution. (For each non-self-loop incident on vertex $i$, the probability mass exiting $i$ via that edge is $\frac{1}{2m+\ell}$, which is equal to the probability mass entering $i$ via that edge.) It follows that, for any vertex $i$, we have $h_{i,i} = \frac{2m+\ell}{\deg(i)}$.

There is another way to view the random walk on $G$: by looking at the graph $G'$ on $2m + \ell$ vertices where each vertex in $G'$ corresponds to an edge plus direction (for non-self-loops) of $G$, and there is an edge in $G'$ between vertices $(i, j)$ and $(j', k')$ iff $j = j'$. The graph $G'$ is now a directed graph, but Theorem 1 can be shown to apply here as well.[1] Note also that a random walk in $G$ corresponds exactly to a random walk in $G'$. In $G'$, however, the stationary distribution is the uniform distribution. (This can be verified by calculation, or derived from the stationary distribution on $G$.) Thus, for any edge $(i, j)$ in $G$ (which is just a vertex in $G'$), the expected number of steps to return to that edge (with direction) after crossing that edge is $1/(2m + \ell)$.

Let $h_{i,j}$ denote the expected number of steps to go from vertex $i$ to vertex $j$. With the above in hand we can prove the following:

**Theorem 2** *Consider a random walk on an undirected, connected, non-bipartite graph $G$ with $\ell$ self-loops and $m$ (other) edges. If there is an edge in $G$ from vertex $i$ to vertex $j$ then $h_{i,j} + h_{j,i} \leq 2m + \ell$ and, in particular, $h_{i,j} < 2m + \ell$.*

**Proof** We prove the theorem in two ways. Looking at the random walk in $G$, we have seen already that $h_{i,i} = \frac{2m+\ell}{\deg(i)}$. If $i = j$ in the theorem then there is a self-loop from $i$ to itself; because $G$ is connected we must have $\deg(i) \geq 2$ and so the theorem holds. For $i \neq j$, we have:

$$\frac{2m + \ell}{\deg(j)} = h_{j,j} = \frac{1}{\deg(j)} \cdot \sum_{k \in N(j)} (1 + h_{k,j}),$$

where $N(j)$ are the neighbors of $j$ (the above assumes $j$ has no self-loops or multiple edges, but the analysis extends to those cases as well). Thus if there is an edge connecting (distinct) vertices $i, j$ (so $i \in N(j)$), then $h_{i,j} < 2m + \ell$. (That $h_{i,j} + h_{j,i} \leq 2m + \ell$ is left as an exercise, but see next.)

Alternately, we may consider the random walk on the graph $G'$ defined earlier. When we take a step from vertex $i$ to vertex $j$ in our random walk on $G$, we view this as being at vertex $(i, j)$ in the graph $G'$. We have seen that the stationary distribution in $G'$ is uniform over the $2m + \ell$ edges (with direction), which means that the expected time to re-visit the edge $(i, j)$ is $2m + \ell$. But re-visiting edge $(i, j)$ corresponds to a one-step transition from $i$ to $j$, re-visiting $i$, and then following edge $(i, j)$ again. In other words, beginning at $j$, the expected number of steps to visit $i$ and then follow edge $(i, j)$ is $2m + \ell$. This gives the desired upper bound on $h_{j,i} + h_{i,j}$. ■

[1] Advanced note: $G'$ is connected since $G$ is, and is *ergodic* since $G$ is. Ergodicity is all that is needed for Theorem 1.

With can now analyze the random-walk algorithm for UConn. Given undirected graph $G$ with $n$ vertices and $|E|$ edges, and vertices $s, t$ in $G$, consider the connected component of $G$ containing $s$. (Technically, we can imagine adding a self-loop at $t$ to ensure that $G$ is non-bipartite. However, it is clear that this has no effect on the algorithm.) If $t$ is in the same connected component as $s$ then there is a path $(s = v_0, v_1, \ldots, v_\ell = t)$ with $\ell < n$; the expected number of steps to go from $v_i$ to $v_{i+1}$ is less than $2|E| + 1$. Thus the expected number of steps to go from $s = v_0$ to $t = v_\ell$ is $O(n|E|)$. Taking a random walk for twice as many steps, we will hit $t$ at some point with probability at least $1/2$.

### 1.1.2 A Randomized Algorithm for 2SAT

Another easy application of random walks is the following $\mathcal{RP}$ algorithm for 2SAT: Begin by choosing a random assignment for the $n$ variables. Then, while there exists an unsatisfied clause $C$, choose one of the variables in $C$ at random and flip its value. Repeat for at most $\Theta(n^2)$ steps, and output 1 if a satisfying assignment is ever found.

Let us show that this algorithm finds a satisfying assignment with high probability when one exists. Fix some satisfying assignment $\vec{x}$, and let the state of the algorithm be the number of positions in which the current assignment matches $\vec{x}$. (So the state $i$ ranges from 0 to $n$.) When the algorithm chooses an unsatisfied clause, the value of at least one of the variables in that clause must differ from the corresponding value of that variable in $\vec{x}$; thus, the state increases with probability at least $1/2$. The worst case is when the state increases with probability exactly $1/2$ (except when $i = 0$, of course). (We can mentally add a self-loop to state $n$ so the graph is non-bipartite.) We thus have a random walk on a line, in the worst case starting at $i = 0$. The expected number of steps to move from state 0 to state $n$ is $h_{0,1} + \cdots + h_{n-1,n} \leq n \cdot (2n + 1) = O(n^2)$.

## References


[1] J. Katz. Lecture notes for *CMSC 652 — Complexity Theory*. Fall 2005.

[2] M. Mitzenmacher and E. Upfal. *Probablity and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

**Notes on Complexity Theory** Last updated: October, 2011

# Lecture 16

*Jonathan Katz*

## 1 Interactive Proofs

Let us begin by re-examining our intuitive notion of what it means to "prove" a statement. Traditional mathematical proofs are *static* and are verified *deterministically*: the verifier checks the claimed proof of a given statement and is either convinced that the statement is true (if the proof is correct) or remains unconvinced (if the proof is flawed — note that the statement may possibly still be true in this case, it just means there was something wrong with the proof). A statement is true (in this traditional setting) iff there *exists* a valid proof that convinces a legitimate verifier.

Abstracting this process a bit, we may imagine a prover $\mathbf{P}$ and a verifier $\mathbf{V}$ such that the prover is trying to convince the verifier of the truth of some particular statement $x$; more concretely, let us say that $\mathbf{P}$ is trying to convince $\mathbf{V}$ that $x \in L$ for some fixed language $L$. We will require the verifier to run in polynomial time (in $|x|$), since we would like whatever proofs we come up with to be efficiently verifiable. A traditional mathematical proof can be cast in this framework by simply having $\mathbf{P}$ send a proof $\pi$ to $\mathbf{V}$, who then deterministically checks whether $\pi$ is a valid proof of $x$ and outputs $\mathbf{V}(x, \pi)$ (with 1 denoting acceptance and 0 rejection). (Note that since $\mathbf{V}$ runs in polynomial time, we may assume that the length of the proof $\pi$ is also polynomial.) The traditional mathematical notion of a proof is captured by requiring:

- If $x \in L$, then there *exists* a proof $\pi$ such that $\mathbf{V}(x, \pi) = 1$.
- If $x \notin L$, then *no matter what proof* $\pi$ the prover sends we have $\mathbf{V}(x, \pi) = 0$.

We refer to the above as a type of proof system, a term we will define more formally later. It should be obvious that $L$ has a proof system of the above sort iff $L \in \mathcal{NP}$.

There are two ways the above can be generalized. First, we can allow the verifier to be *probabilistic*. Assume for a moment that we restrict the prover to sending an empty proof. If the verifier is deterministic, then a language $L$ has a proof system of this sort only if $L \in \mathcal{P}$ (as the prover is no help here). But if the verifier is probabilistic then we can handle any $L \in \mathcal{BPP}$ (if we allow two-sided error). If we go back to allowing non-empty proofs, then we already gain something: we can eliminate the error when $x \in L$. To see this, recall the proof that $\mathcal{BPP} \in \Sigma_2$. The basic idea was that if a set $S \subset \{0,1\}^\ell$ is "small" then for any strings $z_1, \ldots, z_\ell \in \{0,1\}^\ell$, the set $\bigcup_i (S \oplus z_i)$ is still "small." To make this concrete, say $|S| \leq 2^\ell/4\ell$. Then for any $z_1, \ldots, z_\ell$ we have:

$$\left| \bigcup_{i=1}^{\ell} (S \oplus z_i) \right| \leq \ell \cdot |S| \leq 2^\ell/4. \tag{1}$$

On the other hand, if $S$ is "large" (specifically, if $|S| \geq (1 - \frac{1}{4\ell}) \cdot 2^\ell$) then there exist $z_1, \ldots, z_m$ such that $\bigcup_i (S \oplus z_i) = \{0,1\}^\ell$.

The above leads to the following proof system for any $L \in \mathcal{BPP}$: Let $M$ be a $\mathcal{BPP}$ algorithm deciding $L$, using a random tape of length $\ell$, and having error at most $1/4\ell$ (for some polynomial $\ell$).

The prover sends a proof $\pi = (z_1, \ldots, z_\ell)$ to the verifier (where each $z_i \in \{0,1\}^\ell$); $\mathbf{V}$ then chooses a random $r \in \{0,1\}^\ell$ and accepts iff

$$\bigvee_{i=1}^{\ell} M(x; r \oplus z_i) = 1.$$

For common input $x$, let $S_x$ be the set of random coins for which $M(x) = 1$. If $x \in L$, then $|S_x| \geq (1 - \frac{1}{4\ell}) \cdot 2^\ell$ and so there does indeed exist $\pi = (z_1, \ldots, z_\ell)$ such that $r \in \bigcup_i (S_x \oplus z_i)$ for *every* $r \in \{0,1\}^\ell$. Fixing such a $\pi$, this means that for every $r$ there exists an index $i$ for which $r \in S_x \oplus z_i$, and so $r \oplus z_i \in S_x$. Thus, if the prover sends this $\pi$ the verifier will always accept. On the other hand, if $x \notin L$ then $|S_x| \leq 2^\ell/4\ell$ and so, using Eq. (1), we have

$$\Pr_{r \in \{0,1\}^\ell} \left[ r \in \bigcup_{i=1}^{\ell} (S \oplus z_i) \right] \leq 1/4.$$

So $\mathbf{V}$ accepts in this case with probability at most 1/4.

To summarize, we have shown a proof system for any $L \in \mathcal{BPP}$ such that:

- If $x \in L$, then there *exists* a proof $\pi$ such that $\Pr[\mathbf{V}(x, \pi) = 1] = 1$.
- If $x \notin L$, then *no matter what proof* $\pi$ the prover sends we have $\Pr[\mathbf{V}(x, \pi) = 1] \leq 1/4$.

Thus, assuming $\mathcal{P} \neq \mathcal{BPP}$, we see that *randomization* helps. And assuming $\mathsf{co}\mathcal{RP} \neq \mathcal{BPP}$, allowing *communication from the prover to the verifier* helps.

We can further generalize proof systems by allowing *interaction* between the prover and verifier. (One can think of this as allowing the verifier to ask questions. In this sense, the notion of a proof becomes more like a lecture than a static proof written in a book.) Note that unless we also allow randomness, allowing interaction will not buy us anything: if the verifier is deterministic then the prover can predict all the verifier's questions in advance, and simply include all the corresponding answers as part of the (static) proof.

Before we explore the additional power of interaction, we introduce some formal definitions. For interactive algorithms $\mathbf{P}, \mathbf{V}$, let $\langle \mathbf{P}, \mathbf{V} \rangle (x)$ denote the output of $\mathbf{V}$ following an interaction of $\mathbf{P}$ with $\mathbf{V}$ on common input $x$.

**Definition 1** *$L \in \mathcal{IP}$ if there exist a pair of interactive algorithms $(\mathbf{P}, \mathbf{V})$, with $\mathbf{V}$ running in probabilistic polynomial time (in the length of the common input $x$), such that*

1. *If $x \in L$, then $\Pr[\langle \mathbf{P}, \mathbf{V} \rangle (x) = 1] = 1$.*
2. *If $x \notin L$, then for* any *(even cheating) $\mathbf{P}^*$ we have $\Pr[\langle \mathbf{P}^*, \mathbf{V} \rangle (x) = 1] \leq 1/2$.*

*(We stress that $\mathbf{P}$ and $\mathbf{P}^*$ are allowed to be computationally unbounded.) $(\mathbf{P}, \mathbf{V})$ satisfying the above are called a* proof system *for $L$. We say $L \in \mathcal{IP}[\ell]$ if it has a proof system as above using $\ell = \ell(|x|)$ rounds of interaction (where each message sent by either party counts as a round).*

Using this notation, we have seen already that $\mathcal{NP} \cup \mathcal{BPP} \subseteq \mathcal{IP}[1]$.

Some comments on the definition are in order:

- One could relax the definition to allow for *two-sided* error, i.e., error even when $x \in L$. It is known, however, that this results in an equivalent definition [1] (although the round complexity increases by a constant). On the other hand, if the definition is "flipped" so that we allow error only when $x \in L$ (and require no error when $x \notin L$) we get a definition that is equivalent to $\mathcal{NP}$.

- As usual, the error probability of 1/2 is arbitrary, and can be made exponentially small by repeating the proof system suitably many times. (It is easy to see that sequential repetition works, and a more detailed proof shows that parallel repetition works also [2, Appendix C].)

- Although the honest prover is allowed to be computationally unbounded, it suffices for it to be a PSPACE machine. In certain cases it may be possible to have **P** run in polynomial time (for example, if $L \in \mathcal{NP}$ and **P** is given a proof $\pi$ as auxiliary information). In general, it remains an open question as to how powerful **P** needs to be in order to give a proof for some particular class of languages.[1]

### 1.1 Graph Non-Isomorphism is in $\mathcal{IP}$

It is possible to show that $\mathcal{IP} \subseteq$ PSPACE (since, fixing some **V** and some $x$, we can compute the optimal prover strategy in polynomial space). But does interaction buy us anything? Does $\mathcal{IP}$ contain anything more than $\mathcal{NP}$ and $\mathcal{BPP}$? We begin by showing the rather surprising result that graph *non*-isomorphism is in $\mathcal{IP}$.

If $G$ is an $n$-vertex graph and $\pi$ is a permutation on $n$ elements, we let $\pi(G)$ be the $n$-vertex graph in which

$$(i,j) \text{ is an edge in } G \Leftrightarrow (\pi(i),\pi(j)) \text{ is an edge in } \pi(G).$$

Note that $G_0$ is isomorphic to $G_1$ (written $G_0 \cong G_1$) iff $G_0 = \pi(G_1)$ for some $\pi$. (We identify a graph with its adjacency matrix. So, there is a difference between two graphs being *equal* [i.e., having the *same* adjacency matrix] and being *isomorphic*.)

Let $G_0, G_1$ be two graphs. The proof system for graph non-isomorphism works as follows:

1. The verifier chooses a random bit $b$ and a random permutation $\pi$, and sends $G' = \pi(G_b)$.

2. If $G' \cong G_0$, the prover replies with 0; if $G' \cong G_1$, it replies with 1.

3. The verifier accepts iff the prover replies with **V**'s original bit $b$.

Note that if $G_0 \ncong G_1$, then it cannot be the case that both of $G' \cong G_0$ and $G' \cong G_1$ hold; so, the prover always answers correctly. On the other hand, if $G_0 \cong G_1$ (so that $(G_0, G_1)$ is *not* in the language) then the verifier's bit $b$ is completely hidden to the prover (even though the prover is all-powerful!); this is because a random permuted copy of $G_0$ is in this case distributed identically to a random permuted copy of $G_1$. So when $G_0, G_1$ *are* isomorphic, even a cheating prover can only make the verifier accept with probability 1/2.

[1] For example, we will soon see that co$\mathcal{NP} \subseteq \mathcal{IP}$. By what we have just said, we know that if $L \in$ co$\mathcal{NP}$ then there exists a proof system for $L$ with a prover running in PSPACE. But we do not know whether there exists a proof system for $L$ with a prover running in, say, $\mathcal{P}^{\text{co}\mathcal{NP}} = \mathcal{P}^{\mathcal{NP}}$.

## 2 Public-Coin Proof Systems

Crucial to the above protocol for graph non-isomorphism is that the verifier's coins are *private*, i.e., hidden from the prover. At around the same time the class $\mathcal{IP}$ was proposed, a related class was proposed in which the verifier's coins are required to be *public* (still, the verifier does not toss coins until they are needed, so that the prover does not know what coins will be tossed in the future). These are called *Arthur-Merlin* proof systems, where Arthur represents the (polynomial-time) verifier and Merlin the (all-powerful) prover. We again require perfect completeness and bounded soundness (though see Theorems 1 and 2 below). As in the case of $\mathcal{IP}$ one can in general allow polynomially many rounds of interaction. Although it might appear that Arthur-Merlin proofs are (strictly) *weaker* than general interactive proofs, this is not the case [3]. We do not prove this, but an indication of the general technique will be given in Section **??**.

We will consider for now only the Arthur-Merlin classes **MA** and **AM** where there are one or two rounds of interaction. For the class **MA** Merlin talks first, and then Arthur chooses random coins and tries to verify the "proof" that Merlin sent. (We have already seen this type of proof system before when we showed an interactive proof for $\mathcal{BPP}$.) For the class **AM** Arthur talks first but is limited to sending its random coins (so the previous proof of graph non-isomorphism does not satisfy this); then Merlin sends a proof that is supposed to "correspond" to these random coins, and Arthur verifies it. (Arthur does not choose any additional random coins after receiving Merlin's message, although it would not change the class if Arthur did; see Theorem 3, below.) One can also express these in the following definition, which is just a specialization of the general definition of Arthur-Merlin proofs to the above cases:

**Definition 2** *$L \in \mathbf{MA}$ if there exists a deterministic algorithm $\mathbf{V}$ running in polynomial time (in the length of its first input) such that:*

- *If $x \in L$ then $\exists \pi$ such that for all $r$ we have $\mathbf{V}(x, \pi, r) = 1$.*
- *If $x \notin L$ then $\forall \pi$ we have $\Pr_r[\mathbf{V}(x, \pi, r) = 1] \leq 1/2$.*

*$L \in \mathbf{AM}$ if there exists a deterministic algorithm $\mathbf{V}$ running in polynomial time (in the length of its first input) such that:*

- *If $x \in L$ then for all $r$ there exists a $\pi$ such that $\mathbf{V}(x, r, \pi) = 1$.*
- *If $x \notin L$ then $\Pr_r[\exists \pi : \mathbf{V}(x, r, \pi) = 1] \leq 1/2$.*

*In the case of* **MA** *the prover (Merlin) sends $\pi$ and the verifier (Arthur) then chooses random coins $r$, while in the case of* **AM** *the verifier (Arthur) sends random coins $r$ and then the prover (Merlin) responds with $\pi$.*

**MA** can be viewed as a randomized version of $\mathcal{NP}$ (since a fixed proof is verified using randomization) and so a language in **MA** is sometimes said to have "publishable proofs." It is clear that Arthur-Merlin proofs are not more powerful than the class $\mathcal{IP}$ (since an Arthur-Merlin proof system is a particular kind of proof system).

As we have said, **MA** and **AM** do not change if we allow error when $x \in L$. We now prove this. Let $\mathbf{MA}_\varepsilon$ and $\mathbf{AM}_\varepsilon$ denote the corresponding classes when (bounded) two-sided error is allowed.

**Theorem 1** $\mathbf{MA}_\varepsilon = \mathbf{MA}$.

**Proof** Let $L \in \mathbf{MA}_\varepsilon$. Then there is a proof system such that if $x \in L$ then there exists a $\pi$ (that Merlin can send) for which Arthur will accept with high probability (i.e., $\mathbf{V}(x, \pi, r) = 1$ with high probability over choice of $r$), while if $x \notin L$ then for any $\pi$ Arthur will accept only with low probability (i.e., $\mathbf{V}(x, \pi, r) = 1$ with low probability over choice of $r$). For a given $x$ *and* $\pi$, let $S_{x,\pi}$ denote the set of coins $r$ for which $\mathbf{V}(x, \pi, r) = 1$. So if $x \in L$ there exists a $\pi$ for which $S_{x,\pi}$ is "large," while if $x \notin L$ then for *every* $\pi$ the set $S_{x,\pi}$ is "small." Having Merlin send $\pi$ along with a proof that $S_{x,\pi}$ is "large" (exactly as in the $\mathcal{BPP}$ case) gives the desired result. ■

**Theorem 2** $\mathbf{AM}_\varepsilon = \mathbf{AM}$.

**Proof** Say $L \in \mathbf{AM}_\varepsilon$. Using standard error reduction, we thus have a proof system for $L$ in which Arthur sends a random string $r$ of (polynomial) length $\ell$ and the error is less than $1/4\ell$. For a common input $x$, let $S_x$ denote the set of challenges $r$ (that Arthur can send) for which there exists a $\pi$ (that Merlin can send) such that $\mathbf{V}(x, r, \pi) = 1$. By definition of $\mathbf{AM}_\varepsilon$, if $x \in L$ then $|S_x| \geq (1 - \frac{1}{4\ell}) \cdot 2^\ell$ while if $x \notin L$ then $|S_x| \leq 2^\ell/4\ell$. Exactly as in the proof system for $\mathcal{BPP}$ shown previously, this means that we have the following proof system for $L$:

1. Merlin sends $z_1, \ldots, z_\ell \in \{0,1\}^\ell$.
2. Arthur sends random $r' \in \{0,1\}^\ell$.
3. Merlin proves that $r' \in \bigcup_i (S_x \oplus z_i)$ by finding an $i$ such that $r' \oplus z_i \in S_x$, setting $r = r' \oplus z_i$, and then computing the appropriate response $\pi$ to the "challenge" $r$. So Merlin's response is $(i, \pi)$.
4. Arthur runs $\mathbf{V}(x, r' \oplus z_i, \pi)$ and outputs the result.

The above has perfect completeness and soundness error at most 1/4 (we do not go through the analysis since it is the same as in the $\mathcal{BPP}$ case).

The problem is that the above is a three-round proof system (notationally, it shows that $L \in \mathbf{MAM}$)! But we show below that an "**MA**" step can be replaced by an "**AM**" step (while preserving perfect completeness), and so if we apply that here and then combine Merlin's last two messages we get an $\mathbf{AMM} = \mathbf{AM}$ protocol. ■

As promised, we now show that $\mathbf{MA} \subseteq \mathbf{AM}$. More generally, the proof shows that an "**MA**" step can be replaced by an "**AM**" step.

**Theorem 3** $\mathbf{MA} \subseteq \mathbf{AM}$.

**Proof** Say $L \in \mathbf{MA}$. Then we have an **MA** proof system with perfect completeness and soundness error at most 1/2. Say the message $\pi$ sent by Merlin has length $p(|x|)$ for some polynomial $p$. Using error reduction, we can obtain a proof system with perfect completeness and soundness error at most $1/2^{p+1}$; note that the lengths of the messages sent by Merlin do not change (only the lengths of the random coins $r$ used by Arthur increase). So, when $x \in L$ there exists a $\pi$ (call it $\pi^*$) for which $\mathbf{V}(x, \pi^*, r) = 1$ for all $r$ chosen by Arthur, while if $x \notin L$ then for any $\pi$ sent by Merlin the fraction of $r$ for which Arthur accepts is at most $1/2^{p+1}$. Now simply flip the order of messages: first Arthur will choose $r$ and send it to Merlin, and then Merlin replies with a $\pi$ and Arthur verifies exactly as before. If $x \in L$ then Merlin has no problem, and can simply send $\pi^*$. On

the other hand, if $x \notin L$ then what is the probability that there *exists* a $\pi$ that will cause Arthur to accept? Well, for any *fixed* $\pi$ the probability that $\pi$ will work is at most $1/2^{p+1}$. Taking a union bound over *all* $\pi$, we see that the probability that there exists one that works is at most $1/2$. We conclude that $L \in \mathbf{AM}$. ■

As we have said, the same proof shows that an "**MA**" step can be replaced by an "**AM**" step in general. So, $\mathbf{AMA} = \mathbf{AAM} = \mathbf{AM}$ and[2] $\mathbf{MAM} = \mathbf{AMM} = \mathbf{AM}$, and so on. In fact, the above proof technique shows that any Arthur-Merlin proof system with a *constant* number of rounds collapses to exactly **AM** (except for **MA** which may be strictly contained in **AM**). Note that the proof does not extend to proof systems with an arbitrary (non-constant) number of rounds since the communication complexity increases by a multiplicative factor each time an "**MA**" step is replaced by an "**AM**" step (and so if we perform this switch too many times, the communication will no longer be polynomial).

## References


[1] M. Furer, O. Goldreich, Y. Mansour, M. Sipser, and S. Zachos. On Completeness and Soundness in Interactive Proof Systems. In *Advances in Computing Research: A Research Annual*, vol. 5 (Randomness and Computation, S. Micali, ed.), 1989. Available at http://www.wisdom.weizmann.ac.il/~oded/papers.html

[2] O. Goldreich. *Modern Cryptography, Probabilistic Proofs, and Pseudorandomness*. Springer-Verlag, 1998.

[3] S. Goldwasser and M. Sipser. Private Coins vs. Public Coins in Interactive Proof Systems. STOC '86.


[2] The theorem shows that $\mathbf{AMA} \subseteq \mathbf{AAM} = \mathbf{AM}$, but the inclusion $\mathbf{AM} \subseteq \mathbf{AMA}$ is trivial.

**Notes on Complexity Theory** Last updated: October, 2011

Lecture 17

*Jonathan Katz*

# 1 Graph Non-Isomorphism is in AM

The proof system we showed earlier for graph non-isomorphism relied on the fact that the verifier's coins are kept hidden from the prover. Is this inherent? Somewhat surprisingly, we now show a *public-coin* proof for graph non-isomorphism. Before doing so, we take a brief detour to discuss *pairwise-independent hash functions* (which are useful in many other contexts as well).

## 1.1 Pairwise-Independent Hash Functions

Fix some domain $D$ and range $R$. Let $\mathcal{H} = \{h_k\}_{k \in K}$ be a family of functions, where each $k \in K$ defines a function $h_k : D \to R$. We say that $\mathcal{H}$ is[1] *pairwise independent family* if for all distinct $x, x' \in D$ and all (not necessarily distinct) $y, y' \in R$ we have

$$\Pr_{k \leftarrow K}\left[h_k(x) = y \bigwedge h_k(x') = y'\right] = 1/|R|^2 .$$

Put differently, let $D = \{x_1, \ldots, x_\ell\}$ and consider the random variables $Y_i = h_K(x_i)$ (where $K$ is uniform). If $\mathcal{H}$ is pairwise independent then each $Y_i$ is uniformly distributed, and moreover the random variables $Y_1, \ldots, Y_\ell$ are pairwise independent; i.e., for any $i \neq j$ the random variables $Y_i$ and $Y_j$ are independent.

We show a simple construction of a pairwise-independent family for $D = R = \mathbb{F}$, where $\mathbb{F}$ is any finite field. Setting $\mathbb{F} = GF(2^n)$, and viewing strings of length $n$ as field elements, we obtain a construction with $D = R = \{0,1\}^n$. By truncating the output, we obtain a construction with $D = \{0,1\}^n$ and $R = \{0,1\}^\ell$ for any $n \geq \ell$. By padding the input with 0s, we obtain a construction for any $\ell \geq n$.

Fix $D = R = \mathbb{F}$ and let $\mathcal{H} = \{h_{a,b}\}_{a,b \in \mathbb{F}}$ where $h_{a,b}(x) = ax + b$. We claim that $\mathcal{H}$ is pairwise independent. Indeed, fix any distinct $x, x' \in \mathbb{F}$ and any $y, y' \in \mathbb{F}$, and consider the probability (over choice of $a, b$) that

$$\begin{aligned} y &= ax + b \\ y' &= ax' + b. \end{aligned}$$

Using some basic algebra, we see that the above equations are true iff

$$\begin{aligned} a &= (y - y') \cdot (x - x')^{-1} \\ b &= y - (y - y') \cdot (x - x')^{-1} \cdot x. \end{aligned}$$

(Note that the above rely on the fact that $x \neq x'$.) Since $x, x', y, y'$ are fixed, the right-hand sides of the above equations are some fixed elements in $\mathbb{F}$; hence, the probability that $a, b$ satisfy both equations is exactly $1/|\mathbb{F}|^2$ as required.

[1] Frequently, terminology is abused and $h_k \in \mathcal{H}$ is called a pairwise-independent hash function. Formally, it only makes sense to speak about pairwise independent *families* of functions.

For applications, what we actually need are *ways to construct* pairwise-independent families on, say, $\{0,1\}^n$ for some given $n$. In that case we actually want an efficient probabilistic algorithm that, given $n$, outputs a key $k$ that, in turn, defines a function $h_k : \{0,1\}^n \to \{0,1\}^n$ that is efficiently computable. The construction given above satisfies this, though it is not entirely trivial to show this. (In particular, we need to use the fact that we can efficiently generate, and manipulate elements of, $GF(2^n)$.)

## 1.2 An AM Protocol for Graph Non-Isomorphism

We begin by introducing some more notation. For an $n$-vertex graph $G$ (represented as an adjacency matrix), consider the (multi-)set $\mathsf{all}(G) = \{\pi_1(G), \ldots, \pi_{n!}(G)\}$ of all permuted versions of $G$. This is indeed a multi-set (in general) since it is possible that $\pi_i(G) = \pi_j(G)$ even when $\pi_i \neq \pi_j$. For example, consider the 3-vertex graph $G$ in which there is a single edge $(1,2)$. Considering the 6 possible permutations on the labels of the vertices, we see that $\pi = (12)(3)$ maps $G$ to itself, even though $\pi$ is not the identity permutation. On the other hand, $\pi' = (13)(2)$ maps $G$ to a graph isomorphic, but not identical, to $G$.

Let $\mathsf{aut}(G) = \{\pi \mid \pi(G) = G\}$; these are the *automorphisms* of $G$. (Note that $\mathsf{aut}(G)$ is never empty, since the identity permutation is always in $\mathsf{aut}(G)$.) Let $\mathsf{iso}(G)$ be the *set* (not multi-set) $\{\pi(G) \mid \pi \text{ is a permutation}\}$. We claim that for any $n$-vertex graph $G$ we have:

$$|\mathsf{aut}(G)| \cdot |\mathsf{iso}(G)| = n! \; .$$

The reason is that our original multi-set $\mathsf{all}(G)$ has exactly $n!$ elements in it, but each graph in $\mathsf{iso}(G)$ appears exactly $\mathsf{aut}(G)$ times in $\mathsf{all}(G)$ (because $|\mathsf{aut}(G)| = |\mathsf{aut}(\pi(G))|$ for any permutation $\pi$).

We now have the ideas we need to describe the proof system. Given graphs $(G_0, G_1)$, define the set $W$ as follows:

$$W = \left\{ (H,\sigma) \mid \begin{array}{c} H \text{ is isomorphic to either } G_0 \text{ or } G_1 \\ \text{and } \sigma \in \mathsf{aut}(H) \end{array} \right\}.$$

Note that if $G_0 \cong G_1$, then $H$ is isomorphic to $G_0$ iff it is isomorphic to $G_1$; also, the number of automorphisms of any such $H$ is exactly $|\mathsf{aut}(G_0)|$. So the size of $W$ is exactly $|\mathsf{iso}(G_0)| \cdot |\mathsf{aut}(G_0)| = n!$. On the other hand, if $G_0 \ncong G_1$ then the graphs isomorphic to $G_0$ are distinct from those graphs isomorphic to $G_1$. So the size of $W$ in this case is

$$|\mathsf{iso}(G_0)| \cdot |\mathsf{aut}(G_0)| + |\mathsf{iso}(G_1)| \cdot |\mathsf{aut}(G_1)| = 2n! \; .$$

So, $|W \times W| = (n!)^2$ if $G_0 \cong G_1$ and $|W \times W| = 4 \cdot (n!)^2$ if $G_0 \ncong G_1$. Furthermore, it is possible to prove membership in $W$ by giving an isomorphism to either $G_0$ or $G_1$ (the automorphism can be verified in polynomial time).

The above suggests the following proof system:

1. On common input $(G_0, G_1)$, define $W \times W$ as above. (Arthur obviously cannot construct $W \times W$, but all it needs to do is compute the upper bound $4(n!)^2$ on its size.) Let $m = \log 4(n!)^2$, and note that $m$ is polynomial in the input size $n$.

2. Arthur selects a random $h$ from a pairwise-independent family, where $h$ maps strings of the appropriate length (which will become obvious in a minute) to $\{0,1\}^m$. It sends $h$ to Merlin.

3. Merlin finds an $x \in W \times W$ such that $h(x) = 0^m$ (if one exists). It sends this $x$ to Arthur, along with a proof that $x \in W \times W$.

4. Arthur outputs 1 if $x \in W \times W$ and $h(x) = 0^m$.

We now analyze the above. Say $(G_0, G_1)$ are isomorphic. Then $|W \times W| = (n!)^2$ and so

$$\begin{aligned} \Pr_h[\exists x \in W \times W : h(x) = 0^m] &\leq \sum_{x \in W \times W} \Pr_h[h(x) = 0^m] \\ &= (n!)^2 \cdot 2^{-m} = 1/4, \end{aligned}$$

and so Merlin convinces Arthur only with probability at most 1/4. On the other hand, if $G_0 \not\cong G_1$ then $|W \times W| = 4(n!)^2$ and we can bound the desired probability as follows:

$$\begin{aligned} \Pr_h[\exists x \in W \times W : h(x) = 0^m] &\geq \sum_{x \in W \times W} \Pr_h[h(x) = 0^m] \\ &\quad - \frac{1}{2} \cdot \sum_{\substack{x,y \in W \times W \\ x \neq y}} \Pr_h[h(x) = 0^m \wedge h(y) = 0^m] \\ &> 1 - \frac{1}{2} \cdot (4(n!)^2)^2 \cdot (2^{-m})^2 = 1/2\,, \end{aligned}$$

using the inclusion-exclusion principle for the first inequality, and relying on pairwise independence in the second step. (A better bound can be obtained using Chebyshev's inequality.)

The above does not have perfect completeness, but we have seen before that this can be fixed.

### 1.3 Evidence that Graph Isomorphism is not $\mathcal{NP}$-Complete

Let $GI$ be the language of graph isomorphism, and $GNI$ be the language of graph non-isomorphism. In the previous section we showed $GNI \in \mathbf{AM}$. This gives evidence that $GI$ is *not* $\mathcal{NP}$-complete.

**Theorem 1** *If $GI$ is $\mathcal{NP}$-complete, then the polynomial hierarchy collapses (specifically, $\mathsf{PH} = \Sigma_2$).*

**Proof** We first observe that $\mathbf{AM} \subseteq \Pi_2$ (why?). Now, assume $GI$ is $\mathcal{NP}$-complete. Then $GNI$ is $\mathsf{co}\mathcal{NP}$-complete and hence (since $GNI \in \mathbf{AM}$) we have $\mathsf{co}\mathcal{NP} \subseteq \mathbf{AM}$. We show that this implies $\Sigma_2 \subseteq \mathbf{AM} \subseteq \Pi_2$ and hence $\mathsf{PH} = \Sigma_2$.

Say $L \in \Sigma_2$. Then by definition of $\Sigma_2$, there is a language $L' \in \Pi_1 = \mathsf{co}\mathcal{NP}$ such that: (1) if $x \in L$ then there exists a $y$ such that $(x, y) \in L'$, but (2) if $x \notin L$ then for all $y$ we have $(x, y) \notin L'$. This immediately suggests the following proof system for $L$:

1. Merlin sends $y$ to Arthur.

2. Arthur and Merlin then run an **AM** protocol that $(x, y) \in L'$ (this is possible precisely because $L' \in \mathsf{co}\mathcal{NP} \subseteq \mathbf{AM}$).

The above is an **MAM** proof system for $L$. But, as we have seen, this means there is an **AM** proof system for $L$. Since $L \in \Sigma_2$ was arbitrary this means $\Sigma_2 \subseteq \mathbf{AM}$, completing the proof. ■

**Notes on Complexity Theory** Last updated: October, 2011

# Lecture 18

*Jonathan Katz*

## 1 The Power of $\mathcal{IP}$

We have seen a (surprising!) interactive proof for graph non-isomorphism. This begs the question: how powerful is $\mathcal{IP}$?

### 1.1 $\mathsf{co}\mathcal{NP} \subseteq \mathcal{IP}$

As a "warm-up" we show that $\mathsf{co}\mathcal{NP} \subseteq \mathcal{IP}$. We have seen last time that $\mathsf{co}\mathcal{NP}$ is unlikely to have a constant-round interactive proof system (since this would imply[1] that the polynomial hierarchy collapses). For this reason it was conjectured at one point that $\mathcal{IP}$ was not "too much more powerful" than $\mathcal{NP}$. Here, however, we show this intuition wrong: any language in $\mathsf{co}\mathcal{NP}$ has a proof system using a *linear* number of rounds.

We begin by *arithmetizing* a 3CNF formula $\phi$ to obtain a polynomial expression that evaluates to 0 iff $\phi$ has no satisfying assignments. (This powerful technique, by which a "combinatorial" statement about satisfiability of a formula is mapped to an *algebraic* statement about a polynomial, will come up again later in the course.) We then show how to give an interactive proof demonstrating that the expression indeed evaluates to 0.

To arithmetize $\phi$, the prover and verifier proceed as follows: identify 0 with "false" and positive integers with "true." The literal $x_i$ becomes the variable $x_i$, and the literal $\bar{x}_i$ becomes $(1 - x_i)$. We replace "$\wedge$" by multiplication, and "$\vee$" by addition. Let $\Phi$ denote the polynomial that results from this arithmetization; note that this is an $n$-variate polynomial in the variables $x_1, \ldots, x_n$, whose total degree is at most the number of clauses in $\phi$.

Now consider what happens when the $\{x_i\}$ are assigned boolean values: all literals take the value 1 if they evaluate to "true," and 0 if they evaluate to "false." Any clause (which is a disjunction of literals) takes a positive value iff at least one of its literals is true; thus, a clause takes a positive value iff it evaluates to "true." Finally, note that $\Phi$ itself (which is a conjunction of clauses) takes on a positive value iff all of its constituent clauses are positive. We can summarize this as: $\Phi(x_1, \ldots, x_n) > 0$ if $\phi(x_1, \ldots, x_n) = \text{TRUE}$, and $\Phi(x_1, \ldots, x_n) = 0$ if $\phi(x_1, \ldots, x_n) = \text{FALSE}$. Summing over all possible (boolean) settings to the variables, we see that

$$\phi \in \overline{\mathsf{SAT}} \Leftrightarrow \sum_{x_1 \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} \Phi(x_1, \ldots, x_n) = 0.$$

If $\phi$ has $m$ clauses, then $\Phi$ has degree (at most) $m$ (where the [total] degree of a polynomial is the maximum degree on any of its monomials, and the degree of a monomial is the sum of the degrees of its constituent variables). Furthermore, the sum above is at most $2^n \cdot 3^m$. So, if we work

[1] In more detail: a constant-round proof system for $\mathsf{co}\mathcal{NP}$ would imply a constant-round *public-coin* proof system for $\mathsf{co}\mathcal{NP}$, which would in turn imply $\mathsf{co}\mathcal{NP} \subseteq \mathbf{AM}$. We showed last time that the latter implies the collapse of $\mathsf{PH}$.

modulo a prime $q > 2^n \cdot 3^m$ the above is equivalent to:

$$\phi \in \overline{\mathsf{SAT}} \Leftrightarrow \sum_{x_1 \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} \Phi(x_1, \ldots, x_n) = 0 \bmod q.$$

Working modulo a prime (rather than over the integers) confers two advantages: it keeps the numbers from getting too large (since all numbers will be reduced modulo $q$; note that $|q| = \log q$ is polynomial) and it means that we are working over the finite field $\mathbb{F}_q$ (which simplifies the analysis).

We have now reduced the question of whether $\phi$ is unsatisfiable to the question of proving that a particular polynomial expression sums to 0! This already hints at the power of arithmetization: it transforms questions of logic (e.g., satisfiability) into questions of abstract mathematics (polynomials, group theory, algebraic geometry, ...) and we can then use all the powerful tools of mathematics to attack our problem. Luckily, for the present proof the only "deep" mathematical result we need is that a non-zero polynomial of degree $m$ over a field has at most $m$ roots. An easy corollary is that two different polynomials of degree (at most) $m$ can agree on at most $m$ points.

We now show the *sum-check protocol*, which is an interactive proof that $\phi$ is not satisfiable.

- Both prover and verifier have $\phi$. They both generate the polynomial $\Phi$. Note that the (polynomial-time) verifier cannot write out $\Phi$ explicitly, but it suffices for the verifier to be able to evaluate $\Phi$ on any given values of $x_1, \ldots, x_n$. The prover wants to show that $0 = \sum_{x_1 \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} \Phi(x_1, \ldots, x_n)$.
- The prover sends a prime $q$ such that $q > 2^n \cdot 3^m$. The verifier checks the primality of $q$. (The verifier could also generate $q$ itself, and send it to the prover.) All subsequent operations are performed modulo $q$.
- The verifier initializes $v_0 = 0$.
- The following is repeated for $i = 1$ to $n$:
  - The prover sends a polynomial $\hat{P}_i$ (in one variable) of degree at most $m$.
  - The verifier checks that $\hat{P}_i$ has degree at most $m$ and that $\hat{P}_i(0) + \hat{P}_i(1) = v_{i-1}$ (addition is done in $\mathbb{F}_q$). If not, the verifier rejects. Otherwise, the verifier chooses a random $r_i \in \mathbb{F}_q$, computes $v_i = \hat{P}_i(r_i)$, and sends $r_i$ to the prover.
- The verifier accepts if $\Phi(r_1, \ldots, r_n) = v_n \bmod q$ and rejects otherwise. (Note that even though we originally only "cared" about the values $\Phi$ takes when its inputs are *boolean*, nothing stops us from evaluating $\Phi$ at any points in the field.)

**Claim 1** *If $\phi$ is unsatisfiable then a prover can make the verifier accept with probability 1.*

For every $1 \leq i \leq n$ (and given the verifier's choices of $r_1, \ldots, r_{i-1}$) define the degree-$m$ polynomial:

$$P_i(x_i) \stackrel{\text{def}}{=} \sum_{x_{i+1} \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} \Phi(r_1, \ldots, r_{i-1}, x_i, x_{i+1}, \ldots, x_n).$$

We claim that if $\phi$ is unsatisfiable and the prover always sends $\hat{P}_i = P_i$, then the verifier always accepts. In the first iteration ($i = 1$), we have

$$P_1(0) + P_1(1) = \sum_{x_1 \in \{0,1\}} \left( \sum_{x_2 \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} \Phi(x_1, \ldots, x_n) \right) = 0 = v_0,$$

since $\phi$ is unsatisfiable. For $i > 1$ we have:

$$\begin{aligned} P_i(0) + P_i(1) &= \sum_{x_i \in \{0,1\}} \sum_{x_{i+1} \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} \Phi(r_1, \ldots, r_{i-1}, x_i, \ldots, x_n) \\ &= P_{i-1}(r_{i-1}) = v_{i-1}. \end{aligned}$$

Finally, $v_n \stackrel{\text{def}}{=} P_n(r_n) = \Phi(r_1, \ldots, r_n)$ so the verifier accepts.

**Claim 2** *If $\phi$ is satisfiable, then no matter what the prover does the verifier will accept with probability at most $nm/q$.*

The protocol can be viewed recursively, where in iteration $i$ the prover is trying to convince the verifier that

$$v_{i-1} = \sum_{x_i \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} \Phi_i(x_i, \ldots, x_n) \tag{1}$$

for some degree-$m$ polynomial $\Phi_i$ that the verifier can evaluate. (In an execution of the protocol, we have $\Phi_1 = \Phi$; for $i > 1$ we have $\Phi_i(x_i, \ldots, x_n) = \Phi(r_1, \ldots, r_{i-1}, x_i, \ldots, x_n)$.) Each iteration $i$ proceeds as follows: the prover sends some degree-$m$ polynomial $P_i'(x_i)$ (this polynomial is supposed to be equal to $\sum_{x_{i+1} \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} \Phi_i(x_i, \ldots, x_n)$ but may not be if the prover is cheating). The verifier checks that $\sum_{x_i \in \{0,1\}} P_i'(x_i) = v_{i-1}$ and, if so, then chooses a random point $r_i$; the prover then needs to convince the verifier that

$$v_i \stackrel{\text{def}}{=} P_i'(r_i) = \sum_{x_{i+1} \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} \Phi_{i+1}(x_{i+1}, \ldots, x_n),$$

where $\Phi_{i+1}(x_{i+1}, \ldots, x_n) = \Phi_i(r_i, x_{i+1}, \ldots, x_n)$.

Looking now abstractly at Eq. (1), we claim that if Eq. (1) does *not* hold then the prover can make the verifier accept with probability at most $km/q$, where $k = n - i + 1$ is the number of variables we are summing over. The proof is by induction on $k$:

**Base case.** When $k = 1$ we have

$$v_n \neq \sum_{x_n \in \{0,1\}} \Phi_n(x_n) \tag{2}$$

but the prover is trying to convince the verifier otherwise. The prover sends some polynomial $P_n'(x_n)$. If $P_n' = \Phi_n$ the verifier always rejects since, by Eq. (2), $P_n'(0) + P_n'(1) \neq v_n$. If $P_n' \neq \Phi_n$, then the polynomials $P_n'$ and $\Phi_n$ agree on at most $m$ points; since the verifier chooses random $r_n$ and accepts only if $P_n'(r_n) = \Phi_n(r_n)$, the verifier accepts with probability at most $m/q$.

**Inductive step.** Say the claim is true for some value of $k$, and look at Eq. (1) for $k+1$. Renumbering the variables, we have

$$v \neq \sum_{x_1 \in \{0,1\}} \cdots \sum_{x_{k+1} \in \{0,1\}} \Phi(x_1, \ldots, x_{k+1}),$$

but the prover is trying to convince the verifier otherwise. The prover sends some polynomial $P'(x_1)$. Let $\hat{P}(x_1) = \sum_{x_2 \in \{0,1\}} \cdots \sum_{x_{k+1} \in \{0,1\}} \Phi(x_1, \ldots, x_{k+1})$ (this is what the prover is "supposed" to

send). There are again two possibilities: if $P' = \hat{P}$, then $P'(0) + P'(1) \neq v$ and the verifier always rejects. If $P' \neq \hat{P}$ then these polynomials agree on at most $m$ points. So with probability at most $m/q$ the verifier chooses a point $r_1$ for which $P'(r_1) = \hat{P}(r_1)$; if this happens, we will just say the prover succeeds. If this does *not* occur, then

$$v' \stackrel{\text{def}}{=} P'(r_1) \neq \sum_{x_2 \in \{0,1\}} \cdots \sum_{x_{k+1} \in \{0,1\}} \Phi(r_1, x_2, \ldots, x_{k+1}),$$

and we have reduced to a case where we are summing over $k$ variables. By our inductive assumption, the prover succeeds with probability at most $km/q$ in that case. Thus, the overall success probability of the prover is at most $m/q + km/q \leq (k+1)m/q$. This completes the proof.

## 1.2 $\#\mathcal{P} \subseteq \mathcal{IP}$

(We have not yet introduced the class $\#\mathcal{P}$, but we do not use any properties of this class here.) It is relatively straightforward to extend the protocol of the previous section to obtain an interactive proof regarding the *number* of satisfying assignments of some 3CNF formula $\phi$. We need only change the way we do the arithmetization: now we want our arithmetization $\Phi$ to evaluate to *exactly* 1 on any satisfying assignment to $\phi$, and to 0 otherwise. For literals we proceed as before, transforming $x_i$ to $x_i$ and $\bar{x}_i$ to $1 - x_i$. For clauses, we do something different: given clause $a \vee b \vee c$ (where $a, b, c$ are literals), we construct the polynomial:

$$1 - (1 - \hat{a})(1 - \hat{b})(1 - \hat{c}),$$

where $\hat{a}$ represents the arithmetization of $a$, etc. Note that if all of $a, b, c$ are set to "false" (i.e., $\hat{a} = \hat{b} = \hat{c} = 0$) the above evaluates to 0 (i.e., false), while if any of $a, b, c$ are "true" the above evaluates to 1 (i.e., true). Finally, arithmetization of the entire formula $\phi$ (which is the "and" of a bunch of clauses) is simply the product of the arithmetization of its clauses. This gives a polynomial $\Phi$ with the desired properties. Note that the degree of $\Phi$ is now (at most) $3m$, rather than $m$.

Using the above arithmetization, a formula $\phi$ has exactly $K$ satisfying assignments iff:

$$K = \sum_{x_1 \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} \Phi(x_1, \ldots, x_n).$$

Using the exact same protocol as before, except with $q > 2^n$ (since the above summation can now be at most $2^n$) and setting $v_0 = K$ (the claimed number of satisfying assignments), gives an interactive proof for $\#SAT$ with soundness error $3mn/q$.

**Notes on Complexity Theory** Last updated: November, 2011

# Lecture 19

*Jonathan Katz*

## 1 $\mathcal{IP} = \mathsf{PSPACE}$

A small modification of the previous protocol gives an interactive proof for any language in PSPACE, and hence $\mathsf{PSPACE} \subseteq \mathcal{IP}$. Before showing this, however, we quickly argue that $\mathcal{IP} \subseteq \mathsf{PSPACE}$. To see this, fix some proof system $(\mathbf{P}, \mathbf{V})$ for a language $L$ (actually, we really only care about the verifier algorithm $\mathbf{V}$). We claim that $L \in \mathsf{PSPACE}$. Given an input $x \in \{0,1\}^n$, we compute exactly (using polynomial space) the maximum probability with which a prover can make $\mathbf{V}$ accept. (Although the prover is allowed to be all-powerful, we will see that the optimal strategy can be computed in PSPACE and so it suffices to consider PSPACE provers in general.) Imagine a tree where each node at level $i$ (with the root at level 0) corresponds to some sequence of $i$ messages exchanged between the prover and verifier. This tree has polynomial depth (since $\mathbf{V}$ can only run for polynomially many rounds), and each node has at most $2^{n^c}$ children (for some constant $c$), since messages in the protocol have polynomial length. We recursively assign values to each node of this tree in the following way: a leaf node is assigned 0 if the verifier rejects, and 1 if the verifier accepts. The value of an internal node where the prover sends the next message is the *maximum* over the values of that node's children. The value of an internal node where the verifier sends the next message is the (weighted) *average* over the values of that node's children. The value of the root determines the maximum probability with which a prover can make the verifier accept on the given input $x$, and this value can be computed in polynomial space. If this value is greater than $2/3$ then $x \in L$; if it is less than $1/3$ then $x \notin L$.

### 1.1 $\mathsf{PSPACE} \subseteq \mathcal{IP}$

We now turn to the more interesting direction, namely showing that $\mathsf{PSPACE} \subseteq \mathcal{IP}$. We will now work with the PSPACE-complete language TQBF, which (recall) consists of true quantified boolean formulas of the form:

$$\forall x_1 \exists x_2 \cdots Q_n x_n \quad \phi(x_1, \ldots, x_n),$$

where $\phi$ is a 3CNF formula. We begin by arithmetizing $\phi$ as we did in the case of $\#\mathcal{P}$; recall, if $\phi$ has $m$ clauses this results in a degree-$3m$ polynomial $\Phi$ such that, for $x_1, \ldots, x_n \in \{0,1\}$, we have $\Phi(x_1, \ldots, x_n) = 1$ if $\phi(x_1, \ldots, x_n)$ is true, and $\Phi(x_1, \ldots, x_n) = 0$ if $\phi(x_1, \ldots, x_n)$ is false.

We next must arithmetize the quantifiers. Let $\Phi$ be an arithmetization of $\phi$ as above. The arithmetization of an expression of the form $\forall x_n \, \phi(x_1, \ldots, x_n)$ is

$$\prod_{x_n \in \{0,1\}} \Phi(x_1, \ldots, x_n) \stackrel{\text{def}}{=} \Phi(x_1, \ldots, x_{n-1}, 0) \cdot \Phi(x_1, \ldots, x_{n-1}, 1).$$

If we fix values for $x_1, \ldots, x_{n-1}$, then the above evaluates to 1 if the expression $\forall x_n \, \phi(x_1, \ldots, x_n)$ is true, and evaluates to 0 if this expression is false. The arithmetization of an expression of the

form $\exists x_n\, \phi(x_1, \ldots, x_n)$ is

$$\coprod_{x_n \in \{0,1\}} \Phi(x_1, \ldots, x_n) \stackrel{\text{def}}{=} 1 - \big(1 - \Phi(x_1, \ldots, x_{n-1}, 0)\big) \cdot \big(1 - \Phi(x_1, \ldots, x_{n-1}, 1)\big)\,.$$

Note again that if we fix values for $x_1, \ldots, x_{n-1}$ then the above evaluates to 1 if the expression $\exists x_n\, \phi(x_1, \ldots, x_n)$ is true, and evaluates to 0 if this expression is false. Proceeding in this way, a quantified boolean formula $\exists x_1 \forall x_2 \cdots \forall x_n \phi(x_1, \ldots, x_n)$ is true iff

$$1 = \coprod_{x_1 \in \{0,1\}} \prod_{x_2 \in \{0,1\}} \cdots \prod_{x_n \in \{0,1\}} \Phi(x_1, \ldots, x_n). \tag{1}$$

A natural idea is to use Eq. (1) in the protocols we have seen for $\mathsf{co}\mathcal{NP}$ and $\#\mathcal{P}$, and to have the prover convince the verifier that the above holds by "stripping off" operators one-by-one. While this works in principle, the problem is that the *degrees* of the intermediate results are too large. For example, the polynomial

$$P(x_1) = \prod_{x_2 \in \{0,1\}} \cdots \prod_{x_n \in \{0,1\}} \Phi(x_1, \ldots, x_n)$$

may have degree as high as $2^n \cdot 3m$ (note that the degree of $x_1$ doubles each time a $\prod$ or $\coprod$ operator is applied). Besides whatever effect this will have on soundness, this is even a problem for completeness since a polynomially bounded verifier cannot read an exponentially large polynomial (i.e., with exponentially many terms).

To address the above issue, we use a simple[1] trick. In Eq. (1) the $\{x_i\}$ only take on *boolean* values. But for any $k > 0$ we have $x_i^k = x_i$ when $x_i \in \{0, 1\}$. So we can in fact reduce the degree of every variable in any intermediate polynomial to (at most) 1. (For example, the polynomial $x_1^5 x_2^4 + x_1^6 + x_1^7 x_2$ would become $2x_1x_2 + x_1$.) Let $R_{x_i}$ be an operator denoting this "degree reduction" operation applied to variable $x_i$. Then the prover needs to convince the verifier that

$$1 = \coprod_{x_1 \in \{0,1\}} R_{x_1} \prod_{x_2 \in \{0,1\}} R_{x_1} R_{x_2} \coprod_{x_3 \in \{0,1\}} \cdots R_{x_1} \cdots R_{x_{n-1}} \prod_{x_n \in \{0,1\}} R_{x_1} \cdots R_{x_n} \Phi(x_1, \ldots, x_n).$$

As in the previous protocols, we will actually evaluate the above modulo some prime $q$. Since the above evaluates to either 0 or 1, we can take $q$ any size we like (though soundness will depend inversely on $q$ as before).

We can now apply the same basic idea from the previous protocols to construct a new protocol in which, in each round, the prover helps the verifier "strip" one operator from the above expression. Denote the above expression abstractly by:

$$F_\phi = \mathcal{O}_1 \mathcal{O}_2 \cdots \mathcal{O}_\ell\ \Phi(x_1, \ldots, x_n) \bmod q\,,$$

where $\ell = \sum_{i=1}^{n} (i+1)$ and each $O_j$ is one of $\prod_{x_i}$, $\coprod_{x_i}$, or $R_{x_i}$ (for some $i$). At every round $k$ the verifier holds some value $v_k$ and the prover wants to convince the verifier that

$$v_k = \mathcal{O}_{k+1} \cdots \mathcal{O}_\ell\ \Phi_k \bmod q,$$

[1] Of course, it seems simple in retrospect...

where $\Phi_k$ is some polynomial. At the end of the round the verifier will compute some $v_{k+1}$ and the prover then needs to convince the verifier that

$$v_{k+1} = \mathcal{O}_{k+2} \cdots \mathcal{O}_\ell \; \Phi_{k+1} \bmod q,$$

for some $\Phi_{k+1}$. We explain how this is done below. At the beginning of the protocol we start with $v_0 = 1$ and $\Phi_0 = \Phi$ (so that the prover wants to convince the verifier that the given quantified formula is true); at the end of the protocol the verifier will be able to compute $\Phi_\ell$ itself and check whether this is equal to $v_\ell$.

It only remains to describe each of the individual rounds. There are three cases corresponding to the three types of operators (we omit the " mod $q$" from our expressions from now on, for simplicity):

**Case 1:** $\mathcal{O}_{k+1} = \prod_{x_i}$ (for some $i$). Here, the prover wants to convince the verifier that

$$v_k = \prod_{x_i} R_{x_1} \cdots \coprod_{x_{i+1}} \cdots \prod_{x_n} R_{x_1} \cdots R_{x_n} \Phi(r_1, \ldots, r_{i-1}, x_i, \ldots, x_n). \tag{2}$$

(*Technical note*: when we write an expression like the above, we really mean

$$\left( \prod_{x_i} R_{x_1} \cdots \coprod_{x_{i+1}} \cdots \prod_{x_n} R_{x_1} \cdots R_{x_n} \Phi(x_1, \ldots, x_{i-1}, x_i, \ldots, x_n) \right) [r_1, \ldots, r_{i-1}].$$

That is, first the expression is computed symbolically, and then the resulting expression is evaluated by setting $x_1 = r_1, \ldots, x_{i-1} = r_{i-1}$.) This is done in the following way:

- The prover sends a degree-1 polynomial $\hat{P}(x_i)$.
- The verifier checks that $v_k = \prod_{x_i} \hat{P}(x_i)$. If not, reject. Otherwise, choose random $r_i \in \mathbb{F}_q$, set $v_{k+1} = \hat{P}(r_i)$, and enter the next round with the prover trying to convince the verifier that:

$$v_{k+1} = R_{x_1} \cdots \coprod_{x_{i+1}} \cdots \prod_{x_n} R_{x_1} \cdots R_{x_n} \Phi(r_1, \ldots, r_{i-1}, r_i, x_{i+1}, \ldots, x_n). \tag{3}$$

To see completeness, assume Eq. (2) is true. Then the prover can send

$$\hat{P}(x_i) = P(x_i) \stackrel{\text{def}}{=} R_{x_1} \cdots \coprod_{x_{i+1}} \cdots \prod_{x_n} R_{x_1} \cdots R_{x_n} \Phi(r_1, \ldots, r_{i-1}, x_i, \ldots, x_n);$$

the verifier will not reject and Eq. (3) will hold for any choice of $r_i$. As for soundness, if Eq. (2) does *not* hold then the prover must send $\hat{P}(x_i) \neq P(x_i)$ (or else the verifier rejects right away); but then Eq. (3) will not hold except with probability $1/q$.

**Case 2:** $\mathcal{O}_{k+1} = \coprod_{x_i}$ (for some $i$). This case and its analysis are similar to the above and are therefore omitted.

**Case 3:** $\mathcal{O}_{k+1} = R_{x_i}$ (for some $i$). Here, the prover wants to convince the verifier that

$$v_k = R_{x_i} \cdots \prod_{x_n} R_{x_1} \cdots R_{x_n} \Phi(r_1, \ldots, r_j, x_{j+1}, \ldots, x_n), \tag{4}$$

where $j \geq i$. This case is a little different from anything we have seen before. Now:

- The prover sends a polynomial $\hat{P}(x_i)$ of appropriate degree (see below).
- The verifier checks that $\left(R_{x_i}\hat{P}(x_i)\right)[r_i] = v_k$. If not, reject. Otherwise, choose a **new** random $r_i \in \mathbb{F}_q$, set $v_{k+1} = \hat{P}(r_i)$, and enter the next round with the prover trying to convince the verifier that:

$$v_{k+1} = \mathcal{O}_{k+2} \cdots \prod_{x_n} R_{x_1} \cdots R_{x_n} \Phi(r_1, \ldots, r_i, \ldots, r_j, x_{j+1}, \ldots, x_n). \tag{5}$$

Completeness is again easy to see: assuming Eq. (4) is true, the prover can simply send

$$\hat{P}(x_i) = P(x_i) \stackrel{\text{def}}{=} \mathcal{O}_{k+2} \cdots \prod_{x_n} R_{x_1} \cdots R_{x_n} \Phi(r_1, \ldots, r_{i-1}, x_i, r_{i+1}, \ldots, r_j, x_{j+1}, \ldots, x_n)$$

and then the verifier will not reject and also Eq. (5) will hold for any (new) choice of $r_i$. As for soundness, if Eq. (4) does *not* hold then the prover must send $\hat{P}(x_i) \neq P(x_i)$; but then Eq. (5) will not hold except with probability $d/q$ where $d$ is the degree of $\hat{P}$.

This brings us to the last point, which is what the degree of $\hat{P}$ should be. Except for the innermost $n$ reduce operators, the degree of the intermediate polynomial is at most 2; for the innermost $n$ reduce operators, the degree can be up to $3m$.

We may now compute the soundness error of the entire protocol. There is error $1/q$ for each of the $n$ operators of type $\prod$ or $\coprod$, error $3m/q$ for each of the final $n$ reduce operators, and error $2/q$ for all other reduce operators. Applying a union bound, we see that the soundness error is:

$$\frac{n}{q} + \frac{3mn}{q} + \frac{2}{q} \cdot \sum_{i=1}^{n-1} i = \frac{3mn + n^2}{q}.$$

Thus, a polynomial-length $q$ suffices to obtain negligible soundness error.

## Bibliographic Notes

The result that $\mathsf{PSPACE} \subseteq \mathcal{IP}$ is due to Shamir [3], building on [2]. The "simplified" proof given here is from [4]. Guruswami and O'Donnell [1] have written a nice survey of the history behind the discovery of interactive proofs (and the PCP theorem that we will cover in a few lectures).

## References


[1] V. Guruswami and R. O'Donnell. A History of the PCP Theorem. Available at `http://www.cs.washington.edu/education/courses/533/05au/pcp-history.pdf`

[2] C. Lund, L. Fortnow, H.J. Karloff, and N. Nisan. Algebraic Methods for Interactive Proof Systems. *J. ACM* 39(4): 859–868 (1992). The result originally appeared in FOCS '90.

[3] A. Shamir. $\mathcal{IP} = \mathsf{PSPACE}$. *J. ACM* 39(4): 869–877 (1992). Preliminary version in FOCS '90.

[4] A. Shen. $\mathcal{IP} = \mathsf{PSPACE}$: Simplified Proof. *J. ACM* 39(4): 878–880 (1992).

**Notes on Complexity Theory** Last updated: November, 2011

Lecture 20

*Jonathan Katz*

# 1 Zero-Knowledge Proofs

(The following notes just sketch what was covered in class.)

In the complexity-theoretic setting of interactive proofs for some language $L$, we are concerned about a cheating prover who might try to fool a verifier into accepting an incorrect statement $x \notin L$; there are no "security guarantees" for the prover against the (possibly cheating) verifier. In cryptographic settings, however, it may be desirable to ensure some notion of privacy for the prover. Specifically, the prover may be willing to convince the verifier that $x \in L$ but be unwilling to reveal any *additional* information to the verifier.

A "made up" example is the case of a prover who wants to convince a verifier that some mathematical statement is true (and that the prover has a proof of that fact), but does not want to reveal the proof to the verifier for fear that the verifier will then rush to publish the proof on its own. A more realistic example might be the following: say a voter casts a vote by encrypting the vote using the public key $pk$ of some central authority, resulting in some ciphertext $C$. The voter might be required to *prove* that it voted honestly — that is, to prove that $C$ is an encryption of either 0 or 1 — but the voter does not want to divulge its actual vote. To accomplish this, the parties involved can define the language

$$L \stackrel{\text{def}}{=} \{(pk, C) : \exists b \in \{0,1\}, r \in \{0,1\}^n \text{ s.t. } C = \mathsf{Enc}_{pk}(b; r)\};$$

the voter then proves to the verifier that $(pk, C) \in L$.

We discussed the following results:

1. We defined the notion of *honest-verifier* (perfect) zero knowledge, defined $\mathcal{HVPZK}$ as the class of languages having honest-verifier zero-knowledge proofs, and showed that graph isomorphism is in $\mathcal{HVPZK}$. (Note that from a complexity-theoretic viewpoint we may allow the honest prover to be computationally unbounded, but for cryptographic purposes we want the prover to be efficient.)

2. We then introduced the notion of (dishonest-verifier, perfect) zero knowledge, defined $\mathcal{PZK}$ as the class of languages having zero-knowledge proofs, and showed that the same protocol for graph isomorphism satisfies this stronger notion.

3. If we want to reduce the soundness error to $2^{-n}$ by repeating the graph-isomorphism protocol $n$ times, *parallel* repetition preserves honest-verifier zero knowledge, but *sequential* repetition appears necessary if we want to preserve (dishonest-verifier) zero knowledge.

4. $\mathcal{PZK}$ cannot be too powerful; it is known that $\mathcal{PZK} \subseteq \mathbf{AM} \cap \mathsf{coAM}$. (In fact, this holds even if we only require the simulated transcript to be statistically close to the real transcript.) Motivated by this, we consider two relaxations of the notion:

- *Computational* zero knowledge requires the simulated transcript to be (only) computationally indistinguishable from the real transcript. Let $\mathcal{CZK}$ denote the set of languages having computational zero-knowledge proofs. We showed a computational zero-knowledge proof for graph 3-colorability under the assumption that (statistically binding) commitment schemes exist, implying $\mathcal{NP} \subset \mathcal{CZK}$ under the same assumption. In fact, $\mathcal{CZK} = \mathcal{IP}$ under this assumption as well.

  Statistically binding commitment schemes can be constructed from the (minimal) cryptographic assumption of one-way functions.

- A different relaxation is to require perfect zero knowledge as before, but to only demand soundness against *computationally bounded* cheating provers. In this case we refer to *arguments* instead of *proofs*. Statistical zero-knowledge arguments for all of $\mathcal{NP}$ can be based on the existence of statistically hiding commitment schemes. These, in turn, were recently shown to exist based on the assumption of one-way functions.

**Notes on Complexity Theory** Last updated: November, 2011

Lecture 21

*Jonathan Katz*

# 1 Probabilistically Checkable Proofs

Work on *interactive* proof systems motivates further exploration of *non-interactive* proof systems (e.g., the class $\mathcal{NP}$). One specific question is: how many bits of the proof does the verifier need to read? Note that in the usual certificate-based definition of $\mathcal{NP}$, the deterministic "verifier" reads the *entire* certificate, and correctness and soundness hold with probability 1. If we allow the verifier to be probabilistic, and are willing to tolerate non-zero soundness error, is it possible to have the verifier read fewer bits of the proof? (Turning as usual to the analogy with mathematical proofs, this would be like probabilistically verifying the proof of a mathematical theorem by reading only a couple of words of the proof!) Amazingly, we will see that it is possible to have the verifier read only a *constant* number of bits while being convinced with high probability.

Abstracting the above ideas, we define the class $\mathsf{PCP}$ of *probabilistically checkable proofs*:

**Definition 1** *Let $r, q$ be arbitrary functions. We say $L \in \mathsf{PCP}(r(\cdot), q(\cdot))$ if there exists a probabilistic polynomial-time verifier $\mathbf{V}$ such that:*

- *$\mathbf{V}^\pi(x)$ uses $O(r(|x|))$ random coins and reads $O(q(|x|))$ bits of $\pi$.*[1]
- *If $x \in L$ then there exists a $\pi$ such that $\Pr[\mathbf{V}^\pi(x) = 1] = 1$.*
- *If $x \notin L$ then for all $\pi$ we have $\Pr[\mathbf{V}^\pi(x) = 1] < 1/2$.*

Some remarks are in order:

- One can view a probabilistically checkable proof as a form of interactive proof where the (cheating) prover is restricted to committing to its answers *in advance* (rather than choosing them adaptively based on queries in previous rounds). Since the power of the cheating prover is restricted but the abilities of an honest prover are unaffected, $\mathcal{IP} \subseteq \mathsf{PCP}(\mathsf{poly}, \mathsf{poly}) \stackrel{\text{def}}{=} \bigcup_c \mathsf{PCP}(n^c, n^c)$. In particular, $\mathsf{PSPACE} \subseteq \mathsf{PCP}(\mathsf{poly}, \mathsf{poly})$.
- Since $\mathbf{V}$ runs in polynomial time (in $|x|$), the length of $i$ (cf. footnote 1) is polynomial and so it is only meaningful for the length of $\pi$ to be at most exponential in $|x|$. In fact, if the verifier uses $r(n)$ random coins and makes $q(n)$ queries then we may as well assume that any proof $\pi$ for a statement of length $n$ satisfies $|\pi| \leq 2^{r(n)} \cdot q(n)$.
- The soundness error can, as usual, be reduced by repetition. The completeness condition could also be relaxed (as long as there is an inverse polynomial gap between the acceptance probabilities when $x \in L$ and when $x \notin L$). In either case, the parameters $r, q$ may be affected.

[1] Formally, $\mathbf{V}$ has an oracle tape on which it can write an index $i$ and obtain the $i^{\text{th}}$ bit of $\pi$ in the next step.

- The definition allows $\mathbf{V}$ to query $\pi$ *adaptively* (i.e., it may read the $i^{\text{th}}$ bit, and then based on this value determine which index $j$ to read next). We will only consider non-adaptive verifiers. However, any adaptive verifier making a constant number of queries can be converted into a non-adaptive verifier which makes only a (larger) constant number of queries.

## 1.1 Toward Understanding the Power of PCP

An easy observation is that $\mathsf{PCP}(0, \mathsf{poly}) = \mathcal{NP}$. In fact, we have the following stronger result:

**Lemma 1** $\mathsf{PCP}(\log, \mathsf{poly}) = \mathcal{NP}$.

**Proof** Containment of $\mathcal{NP}$ in $\mathsf{PCP}(\log, \mathsf{poly})$ is obvious. For the reverse containment, let $L \in \mathsf{PCP}(\log, \mathsf{poly})$ and let $\mathbf{V}$ be the verifier for $L$. For given $x \in L$, we will show how to construct a witness for $x$; the $\mathcal{NP}$-machine deciding $L$ will follow naturally. Note that we cannot simply use a "good" proof $\pi_x$ (which is guaranteed to exist since $x \in L$) because $\pi_x$ may be exponentially long. However, we *can* use a "compressed" version of $\pi_x$. Specifically, imagine running $\mathbf{V}$ for all possible settings of its $O(\log n)$ random coins (here, $n = |x|$). This results in a set $S$ of only *polynomially many* indices at which $\mathbf{V}$ potentially reads $\pi_x$ (for each setting of its random coins, $\mathbf{V}$ reads polynomially many indices; there are only $2^{O(\log n)} = \mathsf{poly}(n)$ possible settings of $\mathbf{V}$'s random coins). These queries/answers $\{(i, \pi_i)\}_{i \in S}$ will be our $\mathcal{NP}$ witness $w$. Our $\mathcal{NP}$ algorithm for $L$ is simple: on input a witness $w$ of the above form, simulate the computation of $\mathbf{V}$ (in the natural way) for all possible settings of its random coins. (If $\mathbf{V}$ tries to read an index which is not present in $w$, then $\mathbf{V}$ immediately rejects.) Accept only if $\mathbf{V}$ accepts in all those executions. ■

In fact, we have the more general result that $\mathsf{PCP}(r(n), q(n)) \subseteq \text{NTIME}(2^{O(r(n))} \cdot O(q(n)))$.

At the other extreme, if we allow no queries to $\pi$ we obtain $\mathsf{PCP}(\mathsf{poly}, 0) = \mathsf{co}\mathcal{RP}$ (at least if we require perfect completeness, as we do in our definition). This, along with the previous result, shows that we only get something interesting from probabilistically checkable proofs if we consider the power of randomness and proof queries in tandem.

We have the following deep and important result:

**Theorem 2 (The PCP Theorem)** $\mathcal{NP} = \mathsf{PCP}(\log, 1)$.

The number of queries can be taken to be a fixed constant which is the same for all languages $L \in \mathcal{NP}$ (and not, e.g., a constant that depends on the language but not the input length). To see that this follows from the theorem, note that the theorem implies that $\mathsf{SAT}$ has a probabilistically checkable proof where the verifier uses $c \log |\phi|$ random coins and reads $t$ bits when verifying the proof for some 3CNF formula $\phi$. Now, for any $L \in \mathcal{NP}$ we can construct a probabilistically checkable proof where the verifier first applies a Karp reduction to the input to obtain a 3CNF formula $\phi$, and then runs the PCP for $\mathsf{SAT}$ on input $\phi$. If the Karp reduction maps $n$-bit inputs to $n^k$-bit formulae (for some constant $k$), then the verifier for $L$ will use $ck \log |x|$ random coins and reads $t$ bits when verifying the proof that some $x \in L$.

The above characterization is tight under the assumption that $\mathcal{P} \neq \mathcal{NP}$, in the sense that $\mathcal{P} \neq \mathcal{NP}$ is known to imply $\mathcal{NP} \not\subseteq \mathsf{PCP}(o(\log), o(\log))$. Also, although not explicit in the theorem, the PCP theorem also shows how to efficiently convert any witness $w$ for a given $x$ (with respect to a given $\mathcal{NP}$ relation $R$) into a proof $\pi_x$ for which the corresponding PCP verifier always accepts.

For completeness, we also state the following result (that we will not explore further):

**Theorem 3** $\mathsf{PCP}(\mathsf{poly}, \mathsf{poly}) = \mathsf{NEXP} = \mathsf{PCP}(\mathsf{poly}, 1)$.

## 2 PCP and Inapproximability

Assuming $P \neq \mathcal{NP}$, we know that we cannot hope to exactly solve all $\mathcal{NP}$-complete (search) problems in polynomial time. However, we might hope to be able to find an *approximate* solution in polynomial time. The PCP theorem can be used to show limits on the best approximations we can hope to achieve for some specific problems.

### 2.1 Inapproximability of max-SAT

As an example, we show that there exists some constant $\alpha$ such that it is infeasible to approximate (in polynomial time) the maximum number of satisfiable clauses in a 3CNF formula to within a multiplicative factor of $\alpha$. We begin with some definitions.

**Definition 2** *For a formula $\phi$ and an assignment $\mathbf{b}$ to the variables in $\phi$, let $\mathsf{SAT}_{\mathbf{b}}(\phi)$ denote the fraction of clauses satisfied by the given assignment. Let $\mathsf{max\text{-}SAT}(\phi) = \max_{\mathbf{b}}\{\mathsf{SAT}_{\mathbf{b}}(\phi)\}$.*

Note that $\mathsf{max\text{-}SAT}(\phi) = 1$ iff $\phi$ is satisfiable. On the other hand, observe that if $\phi$ has $m$ clauses and is unsatisfiable then it could be the case that $\mathsf{max\text{-}SAT}(\phi) = 1 - 1/m$; in other words, there is no fixed constant $c$ for which $\mathsf{max\text{-}SAT}(\phi) < c$ iff $\phi$ is unsatisfiable. As for a lower bound, it is not hard to show that for any 3CNF formula $\phi$ we have $\mathsf{max\text{-}SAT}(\phi) \geq 7/8$. (*Proof*: A random $\mathbf{b}$ satisfies each clause with probability 7/8, and so satisfies 7/8 of the clauses in expectation. Thus, there must exist a $\mathbf{b}$ that satisfies at least 7/8 of the clauses.)

**Definition 3** *Let $\rho < 1$. A value $k$ is a $\rho$-*approximation for *$\phi$ if*

$$\rho \cdot \mathsf{max\text{-}SAT}(\phi) \leq k \leq \mathsf{max\text{-}SAT}(\phi).$$

*Polynomial-time algorithm $A$ is an $\rho(\cdot)$-*approximation algorithm for 3SAT *if $A(\phi)$ always outputs a $\rho(|\phi|)$-approximation for $\phi$.*

*More generally: for an instance of a* **maximization** *problem where the best solution has value $v$, a $\rho$-approximation ($\rho < 1$) is a value $k$ with $\rho \cdot v \leq k \leq v$. For an instance of a* **minimization** *problem where the best solution has cost $c$, a $\rho$-approximation ($\rho > 1$) is a value $k$ with $c \leq k \leq \rho \cdot c$. A polynomial-time algorithm is a $\rho$-*approximation algorithm *for some problem if it always outputs a $\rho$-approximation to its input instance.*

A 1-approximation algorithm for 3SAT would imply that we could solve 3SAT in polynomial time. By what we have said above, it is trivial to find an 7/8-approximation in polynomial time by always outputting the answer "7/8." Can we do better? Toward showing that there is a limit to how well we can do (assuming $\mathcal{P} \neq \mathcal{NP}$), we introduce the notion of an *amplifying reduction*.

**Definition 4** *Let $c < 1$. A $c$-*amplifying reduction *of* 3SAT *is a polynomial-time function $f$ on 3CNF formulae such that:*

- *If $\phi$ is satisfiable, then $f(\phi)$ is satisfiable. I.e., if $\mathsf{max\text{-}SAT}(\phi) = 1$ then $\mathsf{max\text{-}SAT}(f(\phi)) = 1$.*
- *If $\phi$ is not satisfiable, then every assignment to the variables in $f(\phi)$ satisfies at most a $c$-fraction of the clauses in $f(\phi)$. I.e., if $\mathsf{max\text{-}SAT}(\phi) < 1$ then $\mathsf{max\text{-}SAT}(f(\phi)) < c$.*

*(In particular, an amplifying reduction is a Karp reduction.) We will say that* 3SAT has an amplifying reduction *if it has a $c$-amplifying reduction for some $c < 1$.*

An amplifying reduction for 3SAT implies a hardness-of-approximation result for max-SAT:

**Lemma 4** *Assume $\mathcal{P} \neq \mathcal{NP}$ and that* 3SAT *has a $c$-amplifying reduction. Then there is no $c$-approximation algorithm for* 3SAT.

**Proof** Assume to the contrary that there is a $c$-approximation algorithm $A$ for 3SAT. We can then deterministically solve 3SAT in polynomial time as follows: on input formula $\phi$, run $A(f(\phi))$ to obtain output $k$. If $k \geq c$, output 1; otherwise, output 0. To see correctness of this algorithm, note that when $\phi$ is satisfiable then $\mathsf{max\text{-}SAT}(f(\phi)) = 1$ and so the output $k$ of $A$ must be at least $c$. On the other hand, when $\phi$ is not satisfiable then $\mathsf{max\text{-}SAT}(f(\phi)) < c$ and so the output $k$ of $A$ must satisfy $k < c$. The claim follows. ■

In general, say we have a reduction $f$ that maps, e.g., boolean formula to instances of a maximization (resp., minimization) problem such that

- If $\phi$ is satisfiable, then $f(\phi)$ has value (resp., cost) $\alpha(n)$, where $n$ denotes the size of $f(\phi)$;
- If $\phi$ is not satisfiable, then $f(\phi)$ has value (resp., cost) strictly less than $\beta(n) < \alpha(n)$ (resp., strictly more than $\beta(n) > \alpha(n)$).

Then, assuming $\mathcal{P} \neq \mathcal{NP}$, the maximization (resp., minimization) problem has no $(\beta(n)/\alpha(n))$-approximation algorithm.

To establish the connection between the PCP theorem and inapproximability, we show that the PCP theorem implies the existence of an amplifying reduction for 3SAT. In fact, the implication goes in both directions, thus showing that one way to prove the PCP theorem is to construct an amplifying reduction.

**Lemma 5** $\mathcal{NP} \subseteq \mathsf{PCP}(\log, 1)$ *if and only if* 3SAT *has an amplifying reduction.*

**Proof** One direction is easy. If 3SAT has an amplifying reduction $f$, then we can construct the following PCP system for 3SAT: On input $\phi$, the verifier computes $f(\phi)$. The proof will contain a satisfying assignment for $f(\phi)$ (i.e., position $i$ of the proof contains the assignment to $x_i$). To check the proof, the verifier chooses a random clause in $f(\phi)$, queries for the assignments to the 3 variables of that clause, and then verifies that the clause is satisfied for those settings of the variables. It accepts if and only if that is the case.

If $\phi$ is satisfiable then $f(\phi)$ is satisfiable and so a valid proof (consisting of a satisfying assignment for $f(\phi)$) exists. On the other hand, if $\phi$ is not satisfiable then at most a $c$-fraction of the clauses in $f(\phi)$ are satisfiable (for *any* assignment to the variables), and so the verifier accepts with probability at most $c$ regardless of the proof. Since $c$ is a constant, repeating the above procedure a constant number of times (and accepting only if each procedure leads to acceptance) will give the desired soundness error $1/2$ using an overall constant number of queries. Also, the number of random bits needed to select a random clause is logarithmic in $|\phi|$ since $|f(\phi)|$ is polynomial in $|\phi|$.

The other direction is the more interesting one. Say $\mathsf{SAT} \in \mathsf{PCP}(\log, 1)$, and let $\mathbf{V}$ be a verifier for SAT using $c \log n$ random coins (on input $\phi$ with $|\phi| = n$) and making $t$ queries. We now describe an amplifying reduction $f$. On input a 3CNF formula $\phi$ do:

- For each setting $r$ of the random coins for $\mathbf{V}$, do the following:
  - Determine the $t$ indices $q_1, \ldots q_t$ that $\mathbf{V}(\phi; r)$ would when using random coins $r$ (recall that without loss of generality these indices are chosen non-adaptively).

– Run $\mathbf{V}(\phi; r)$ on all possible settings for these bits of the proof to determine when $\mathbf{V}$ accepts in this case. In this way, one may define a CNF formula $\hat{\phi}_r$ on the variables $x_{q_1}, \ldots, x_{q_t}$ such that $\hat{\phi}_r$ evaluates to true exactly when $\mathbf{V}(\phi; r)$ would accept. (We stress that variables of the type $x_{q_i}$ are the same for the different settings of $r$.) The number of clauses in $\hat{\phi}_r$ is constant since $t$ is constant. Using auxiliary variables (different for each $r$), we may convert $\hat{\phi}_r$ to an equivalent 3CNF formula $\phi_r$. The number of clauses in $\phi_r$ is constant as well.

- Set the output $f(\phi)$ to be $\bigwedge_{r \in \{0,1\}^{c \log n}} \phi_r$.

Note that the above can be implemented in polynomial time and, in particular, both the number of clauses and the number of variables in $f(\phi)$ are polynomial.[2]

We claim that $f$, as given above, is an amplifying reduction. It is not hard to see that if $\phi$ is satisfiable then $f(\phi)$ is (this follows from perfect completeness of the PCP system). On the other hand, assume $\phi$ is not satisfiable. Then for *any* setting of the variables in $f(\phi)$, at least half of the $\{\phi_r\}$ are not satisfied (this follows from soundness of the PCP system). In each unsatisfied $\phi_r$ there is at least one unsatisfied clause. Let $t' = O(1)$ denote the maximum number of clauses in any of the $\{\phi_r\}$. It follows that for any setting of the variables, the fraction of unsatisfied clauses in $f(\phi)$ is at least $\beta = 1/2t'$, and so the fraction of satisfied clauses is at most $1 - \beta$. This means that $f$ is a $c$-amplifying reduction for any $c > 1 - \beta$. ■

An alternate way of viewing the above result is in terms of a promise problem where "yes" instances correspond to satisfiable 3CNF formulae, and "no" instances correspond to 3CNF formulae $\phi$ for which $\mathsf{max\text{-}SAT}(\phi) < c$. The above result implies that this promise problem is $\mathcal{NP}$-hard.

## 2.2 Inapproximability of Other Problems

Different $\mathcal{NP}$-complete problems may behave differently when it comes to how well they can be approximated. We discuss some examples here.

### 2.2.1 Minimum Vertex Cover and Maximum Independent Set

For a given graph $G$, note that a minimum vertex cover is the complement of a maximum independent set; thus, with respect to exact solutions, the problems are identical. However, they behave differently with respect to approximation. (Actually, this should not be too surprising: For an $n$-vertex graph with maximum independent set of size $I = n - O(1)$, an independent set of size $\rho \cdot I$ is a $\rho$-approximation; however, it gives a vertex cover of size $n - \rho I$ which is only an $(n - \rho I)/(n - I) = O(n)$-approximation and so arbitrarily bad as $n$ gets large.)

**Lemma 6** *There is a Karp reduction from 3CNF formulae to graphs such that for every 3CNF formula $\phi$ with $\mathsf{max\text{-}SAT}(\phi) = v$, the graph $G = f(\phi)$ has a maximum independent set of size $\frac{v}{7} \cdot n$ (where $n$ denotes the number of vertices in $G$)*

The above just uses the standard Karp reduction we have seen in class before. Using our previous inapproximability result for $\mathsf{max\text{-}SAT}$, this immediately implies the following:

[2] Note that at most $2^{c \log n} \cdot t$ indices are ever potentially queried by $\mathbf{V}$.

**Corollary 7** *If $\mathcal{P} \neq \mathcal{NP}$ then there are constants $\rho < 1$ and $\rho' > 1$ such that there is no $\rho$-approximation algorithm for Maximum Independent Set, and no $\rho'$-approximation algorithm for Minimum Vertex Cover.*

Actually, by reducing directly to the PCP theorem (rather than to 3SAT) we can get a stronger inapproximability result for Maximum Independent Set:

**Theorem 8** *There is no 1/2-approximation algorithm for Maximum Independent Set.*

**Proof** Let $\alpha(n) = \Theta(n)$ be some function we will fix later. Given an arbitrary $\mathcal{NP}$-complete language $L$, we show a transformation $f$ that takes an input $x$ and outputs a graph $G_x$ such that the following hold:

- If $x \in L$, then $G_x = f(x)$ has a maximum independent set of size $\alpha(n)$ (where $n$ denotes the number of vertices in $G_x$).
- If $x \notin L$ then $G_x = f(x)$ has a maximum independent set of size at most $\alpha(n)/2$.

By the PCP theorem, there exists a probabilistically checkable proof system for $L$ with a polynomial-time verifier $\mathbf{V}$ that on input $x$ make $t$ queries to its proof and uses $\ell = O(\log |x|)$ coin tosses. Let $r_1, \ldots, r_m \in \{0,1\}^\ell$ denote the sequence of all possible coin tosses of $\mathbf{V}$ (note $m = \mathsf{poly}(|x|)$), and let $q_1^i, \ldots, q_t^i$ denote the queries made on random coin tosses $r_i$. (Recall we assume queries are made non-adaptively.) Let $a_1^i, \ldots, a_t^i$ be a sequence of possible answers. Define a graph $G_x$ as follows:

**Vertices** For each set of random coins $r_i$ and each possible set of answers $a_1^i, \ldots, a_t^i$, the tuple

$$\left(r_i, (q_1^i, a_1^i), \ldots, (q_t^i, a_t^i)\right)$$

is a vertex if and only if $\mathbf{V}$ would accept $x$ when using random coins $r_i$ and receiving those answers to its queries.

Since $r_i$ and $x$ uniquely determine the queries, there are at most $m \cdot 2^t$ vertices in $G_x$.

**Edges** Two vertices $v$ and $u$ have an edge between them if and only if they are not consistent. (Two vertices are *not* consistent if they contain different answers to the same query.) Note that if vertices $u, v$ contain the same random tape $r_i$ then they cannot be consistent and so will share an edge.

Finally, add isolated vertices (if necessary) to obtain a graph with exactly $m \cdot 2^t$ vertices.

Define $\alpha(n) \stackrel{\text{def}}{=} n/2^t$, so that $\alpha(m \cdot 2^t) = m$. We show that $G_x$ satisfies our desiderata:

- When $x \in L$, there exists a proof $\pi$ for which $\mathbf{V}$ accepts for every setting of its random tape. This implies the existence of an independent set in $G_x$ of size at least $m$.
- When $x \notin L$, the existence of an independent set with $m/2$ (or more) vertices would imply the existence of a proof that would cause $\mathbf{V}$ to accept with probability at least $1/2$, in contradiction to the soundness of the PCP system. ■

We can further amplify the above results, and show that there is *no* constant-factor approximation algorithm for Maximum Independent Set.

**Theorem 9** *Assume $\mathcal{P} \neq \mathcal{NP}$. Then for* **every** $\rho \in (0, 1]$*, Maximum Independent Set cannot be $\rho$-approximated in polynomial time.*

**Proof** Given a graph $G$ and an integer $k$, define the graph $G^k$ as follows: vertices of $G^k$ correspond to subsets of $k$ vertices of $G$; two vertices $S_1, S_2$ of $G^k$ have no edge between them iff $S_1 \cup S_2$ is an independent set in $G$. Note that $G^k$ can be generated from $G$ in polynomial time. If $G$ has $n$ vertices, then $G^k$ has $\binom{n}{k}$ vertices. If $G$ has an independent set $S$ then the vertices in $G^k$ corresponding to all $k$-size subsets of $S$ form an independent set in $G^k$. Conversely, if there is some independent set $S_1, \ldots, S_\ell$ of vertices in $G^k$, then the vertices in $\cup_i S_i$ form an independent set in $G$. Thus, if the maximum independent set in $G$ has size $|S|$ then the maximum independent set in $G^k$ has size $\binom{|S|}{k}$.

Applying the reduction from Lemma 6, followed by the reduction above (using some fixed, constant value of $k$), we get a reduction $f$ mapping boolean formulae to graphs, such that if $\phi$ is satisfiable then $G^k = f(\phi)$ has a maximum independent set of size $\binom{|S|}{k}$, while if $\phi$ is not satisfiable then $G^k$ has a maximum independent set of size $\binom{\rho \cdot |S|}{k}$ for some $\rho < 1$, where $|S| = n/7$ (and $n$ is the number of nodes in the intermediate graph produced by the reduction from Lemma 6). Taking $k$ sufficiently large, the ratio $\binom{\rho \cdot |S|}{k} / \binom{|S|}{k} \approx \rho^k$ can be made arbitrarily small. ■

**Notes on Complexity Theory** Last updated: November, 2011

Lecture 22

*Jonathan Katz*

# 1 $\mathcal{NP} \subseteq \mathsf{PCP}(\mathsf{poly}, 1)$

We show here a probabilistically checkable proof for $\mathcal{NP}$ in which the verifier reads only a *constant* number of bits from the proof (and uses only polynomially many random bits). In addition to being of independent interest, this result is used as a key step in the proof of the PCP theorem itself.

To show the desired result, we will work with the $\mathcal{NP}$-complete language of *satisfiable quadratic equations*. Instances of this problem consist of a system of $m$ quadratic equations

$$\left\{ \sum_{i,j=1}^{n} c_{i,j}^{(k)} \cdot x_i \, x_j = c^{(k)} \right\}_{k=1}^{m} \tag{1}$$

(over the field $\mathbb{F}_2$) in the $n$ variables $x_1, \ldots, x_n$. (Note that we can assume no linear terms since $x_i = x_i \cdot x_i$ in $\mathbb{F}_2$ and the summations above include the case $i = j$.) A system of the above form is said to be *satisfiable* if there is an assignment to the $\{x_i\}$ for which every equation is satisfied.

It is obvious that this problem is in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete we reduce an instance of 3SAT to an instance of the above. Given a 3SAT formula $\phi$ on $n$ variables, using arithmetization we can express each of its clauses as a cubic equation. (One way to do this is as follows: arithmetize the literal $x_j$ by the term $1 - x_j$ and the literal $\bar{x}_j$ by the term $x_j$; a clause $\ell_1 \vee \ell_2 \vee \ell_3$ is arithmetized by the product of the arithmetization of its literals. Then ask whether there is an assignment under which the arithmetization of all the clauses of $\phi$ equal 0.) To reduce the degree to quadratic, we introduce the "dummy" variables $\{x_{i,j}\}_{i,j=1}^{n}$ and then: (1) replace monomials of the form $x_i x_j x_k$ with a monomial of the form $x_{i,j} x_k$, and (2) introduce $n^2$ new equations of the form $x_{i,j} - x_i x_j = 0$.

We remark that there is no hope of reducing the degree further (unless $\mathcal{P} = \mathcal{NP}$) since a system of linear equations can be solved using standard linear algebra.

## 1.1 A PCP for Satisfiable Quadratic Equations: An Overview

For the remainder of these notes, we will assume a system of $m$ equations in the $n$ variables $\{x_i\}$, as in Eq. (1). The proof string $\pi$ will be a boolean string $\pi \in \{0,1\}^{n^2}$ that we index by a binary vector $\vec{v}$ of length $n^2$. Equivalently, we will view $\pi$ as a function $\pi : \{0,1\}^{n^2} \to \{0,1\}$. For a given system of satisfiable quadratic equations, $\pi$ should be such that

$$\pi(\vec{v}) \stackrel{\text{def}}{=} \sum_{i,j=1}^{n} a_i a_j v_{i,j}$$

for some satisfying assignment $(a_1, \ldots, a_n)$, where $\vec{v} = (v_{1,1}, \ldots, v_{1,n}, \ldots, v_{n,1}, \ldots, v_{n,n})$ Note that with $(a_1, \ldots, a_n)$ fixed, $\pi$ is a linear function of $\vec{v}$; i.e., $\pi$ is just the dot product of the input with the fixed string $(a_{1,1}, \ldots, a_{n,n})$.

Roughly speaking, given access to a proof string $\pi$ we will have the verifier check three things: (1) that the proof string encodes a linear function; i.e.,

$$\pi(\vec{v}) = \sum_{i,j=1}^{n} \lambda_{i,j}\, v_{i,j}$$

for some $\{\lambda_{i,j}\}$; (2) that the coefficients of the linear function encoded by the proof string are *consistent*; namely, that $\lambda_{i,j} = \lambda_{i,i} \cdot \lambda_{j,j}$ for all $i, j$; and (3) that the assignment defined by setting $a_i = \lambda_{i,i}$ is indeed a satisfying assignment. (Note that these all hold for a "good" proof $\pi$ when the system of equations is satisfiable.) Because the verifier is restricted to making a very small number of queries, the verifier will be unable to verify any of the above with certainty, but it will be able to verify these conditions probabilistically. In these notes, we focus only on achieving a constant probability of rejection when the system of equations is unsatisfiable; we aim for the simplest proof and make no attempt to optimize the constants. Of course, by using a constant number of independent repetitions we can then reduce the error probability to $1/2$ (while still reading only a constant number of bits from $\pi$ and using polynomially many random bits).

We discuss in turn the tests used to verify each of the properties above, and then show how they can be combined to yield the desired PCP system.

## 2 The Linearity Test

A function $f : \{0,1\}^N \to \{0,1\}$ is linear if there exists an $r \in \{0,1\}^N$ such that $f(x) = \langle x, r \rangle$, i.e.,

$$f(x_1 \cdots x_N) = \sum_{i=1}^{N} r_i \cdot x_i \,.$$

In this section we show how to test whether a function $\pi : \{0,1\}^N \to \{0,1\}$ is (close to) linear.

Let us first define a notion of distance for functions. Two functions $f, g : \{0,1\}^N \to \{0,1\}$ have distance $\delta$ if they disagree on a $\delta$ fraction of their inputs; that is, if $\Pr_x[f(x) \neq g(x)] = \delta$ (where $x$ is chosen uniformly from $\{0,1\}^N$). Viewing a boolean function over $\{0,1\}^N$ as a binary string of length $2^N$, two functions have distance $\delta$ if their Hamming distance is $\delta \cdot 2^N$. We say a function $f$ is distance at least $\delta$ from linear if for all linear functions $g$ the distance between $f$ and $g$ is at least $\delta$, and define "distance $\delta$ from linear" and "distance at most $\delta$ from linear" similarly.

The following test allows a verifier, given access to $\pi$, to check whether $\pi$ is "close" to linear:

- Choose random $x, y \in \{0,1\}^N$.
- Query $\pi(x), \pi(y)$, and $\pi(x+y)$.
- Accept if and only if $\pi(x) + \pi(y) = \pi(x+y)$, where addition is in $\mathbb{F}_2$ component-wise.

Note that if $\pi$ is linear, then the verifier always accepts since

$$\begin{aligned} \pi(x+y) &= \sum_{i=1}^{N} r_i \cdot (x_i + y_i) \\ &= \left(\sum_{i=1}^{N} r_i\, x_i\right) + \left(\sum_{i=1}^{N} r_i\, y_i\right) = \pi(x) + \pi(y). \end{aligned}$$

The interesting part is to show that when $\pi$ is "far" from linear then the verifier rejects with high probability. In the following section we prove:

**Theorem 1** *If $\pi$ has distance $\varepsilon$ from linear, the linearity test rejects with probability at least $\varepsilon$.*

Of course, by repeating the test a constant number of times we can increase the rejection probability to any constant less than 1.

In the next section we give a proof of Theorem 1 based on Fourier analysis. (A proof that does not use Fourier analysis is also possible but, conceptually speaking, Fourier analysis is exactly the right tool for this setting.) The proof is unnecessary for understanding the rest of the PCP construction, and so the reader willing to take Theorem 1 on faith can skip directly to Section 3.

## 2.1 Proof of Theorem 1

The first thing we will do is view $\pi$ as a function from $\{-1,1\}^N$ to $\{-1,1\}$, by mapping each bit $b$ of the input and output to the value $(-1)^b$. Given this notational switch, the linearity test chooses random $x, y \in \{-1,1\}^N$, and accepts if and only if $\pi(x) \cdot \pi(y) \cdot \pi(x \circ y) = 1$, where "$\circ$" denotes component-wise product.

The proof relies on some basic Fourier analysis; we provide some background first. View the set of functions from $\{-1,1\}^N$ to the reals as a vector space (over the reals). This is a vector space of dimension $2^N$, with one basis given by the functions $\{I_v : \{-1,1\}^N \to \mathbb{R}\}_{v \in \{-1,1\}^N}$ where

$$I_v(v') \stackrel{\text{def}}{=} \begin{cases} 1 & v' = v \\ 0 & \text{otherwise} \end{cases}.$$

To confirm that this is a basis, note that any function $\pi : \{-1,1\}^N \to \mathbb{R}$ can be expressed as:

$$\pi = \sum_{v \in \{-1,1\}^N} \pi(v) \cdot I_v.$$

We will also define an inner product $\langle \cdot, \cdot \rangle$ on this vector space, via:

$$\langle f, g \rangle \stackrel{\text{def}}{=} \frac{1}{2^N} \cdot \sum_v f(v) \cdot g(v) = \mathbf{Exp}_v[f(v) \cdot g(v)].$$

(Note that this inner product is bilinear.) We see that the basis given above is orthogonal.

The basis described above is the "standard" one. In our context, however, there is another basis that works even better: the Fourier basis $\{\chi_v\}_{S \subseteq [N]}$ where

$$\chi_S(v') = \prod_{i \in S} v'_i$$

(with the empty product when $S = \emptyset$ interpreted as a '1'). One can check that these functions are orthogonal and hence, since there are $2^N$ such functions, this is indeed a basis; in fact it is an *orthonormal* basis. This means that we can write any function $f : \{-1,1\} \to \mathbb{R}$ as

$$f = \sum_{S \subseteq [N]} \hat{f}(S) \cdot \chi_S$$

with $\hat{f}(S) \in \mathbb{R}$; by orthonormality, we have

$$\langle f, \chi_S \rangle = \left\langle \sum_{S \subseteq [N]} \hat{f}(S) \cdot \chi_S,\ \chi_S \right\rangle = \hat{f}(S).$$

The first hint that the Fourier basis might be useful for our purposes is the following observation: If $f, g$ are functions from $\{-1,1\}^N$ to $\{-1,1\}$, then

$$\langle f, g \rangle = \frac{1}{2^N} \cdot \Big( \big|\{x \mid f(x) = g(x)\}\big| - \big|\{x \mid f(x) \neq g(x)\}\big| \Big) = 1 - 2 \cdot \Pr_x[f(x) \neq g(x)];$$

in other words, if $f$ is distance $\delta$ from $g$, then $\langle f, g \rangle = 1 - 2\delta$. Note that each $\chi_S$ is a linear function (except that everything has been translated from $\{0,1\}$ to $\{-1,1\}$), and so the Fourier basis includes *all* linear functions. Thus, *to find the linear function closest to $\pi$ we simply need to find $S$ for which $\langle \chi_S, \pi \rangle$ is maximized.* Furthermore, *$\pi$ is far from linear if and only if $\langle \chi_S, \pi \rangle$ is small for* **all** $S$. We will use this in the proof below.

Before turning to the proof of the linearity test, we state two facts that follow from standard linear algebra.

- The inner product $\langle f, g \rangle$ of any two functions $f, g$ is given by the sum of the product of the coefficients of $f$ and $g$ in any orthonormal basis. Thus, in particular, we have

  $$\langle f, g \rangle = \sum_{S \subseteq N} \hat{f}(S) \cdot \hat{g}(S)\,.$$

  This is known as *Plancherel's theorem.*

- It follows from the above that $\langle f, f \rangle = \sum_S \hat{f}(S)^2$. If the range of $f$ is $\{-1,1\}$, then (by definition of the inner product)

  $$\langle f, f \rangle = \frac{1}{2^N} \sum_v f(v)^2 = 1.$$

  We thus conclude that when $f$ maps onto $\{-1,1\}$, we have $\sum_S \hat{f}(S)^2 = 1$. This is known as *Parseval's theorem.*

We can now prove the following result:

**Theorem 2** *If $\pi$ has distance $\varepsilon$ from linear, the linearity test rejects with probability at least $\varepsilon$.*

We begin with a lemma. Amazingly, the lemma is pretty powerful although its proof involves nothing more than grinding through some algebraic manipulations.

**Lemma 3** Pr[linearity test accepts $\pi$] $= \frac{1}{2} + \frac{1}{2} \cdot \sum_S \hat{\pi}(S)^3$.

**Proof** In our notation, the linearity test chooses random $x, y \in \{-1,1\}^N$, and accepts iff $\pi(x) \cdot \pi(y) \cdot \pi(x \circ y) = 1$. Since $\pi$ is boolean (so has range $\{-1,1\}$), we have $\pi(x) \cdot \pi(y) \cdot \pi(x \circ y) \in \{-1,1\}$.

So, $I \stackrel{\text{def}}{=} \frac{1}{2} + \frac{1}{2}\pi(x) \cdot \pi(y) \cdot \pi(x \circ y)$ is an indicator random variable for the event that the linearity test accepts. Thus:

$$\begin{aligned} \Pr[\text{linearity test accepts}] &= \mathbf{Exp}_{x,y}[I] \\ &= \frac{1}{2} + \frac{1}{2} \cdot \mathbf{Exp}_{x,y}[\pi(x) \cdot \pi(y) \cdot \pi(x \circ y)]. \end{aligned} \tag{2}$$

Expanding $\pi$ in terms of its Fourier coefficients gives

$$\begin{aligned} &\mathbf{Exp}_{x,y}[\pi(x) \cdot \pi(y) \cdot \pi(x \circ y)] = \\ &\quad \mathbf{Exp}_{x,y}\left[\left(\sum_S \hat{\pi}(S)\,\chi_S(x)\right) \cdot \left(\sum_{S'} \hat{\pi}(S')\,\chi_{S'}(y)\right) \cdot \left(\sum_{S''} \hat{\pi}(S'')\,\chi_{S''}(x \circ y)\right)\right] \\ &= \mathbf{Exp}_{x,y}\left[\sum_{S,S',S''} \hat{\pi}(S)\,\hat{\pi}(S')\,\hat{\pi}(S'')\,\chi_S(x)\,\chi_{S'}(y)\,\chi_{S''}(x \circ y)\right] \\ &= \sum_{S,S',S''} \hat{\pi}(S)\,\hat{\pi}(S')\,\hat{\pi}(S'') \cdot \mathbf{Exp}_{x,y}\left[\chi_S(x)\,\chi_{S'}(y)\,\chi_{S''}(x \circ y)\right]. \end{aligned} \tag{3}$$

By definition of $\chi_S$, for any fixed $S, S', S''$ we have:

$$\begin{aligned} \mathbf{Exp}_{x,y}\left[\chi_S(x)\,\chi_{S'}(y)\,\chi_{S''}(x \circ y)\right] &= \mathbf{Exp}_{x,y}\left[\prod_{i \in S} x_i \cdot \prod_{i \in S'} y_i \cdot \prod_{i \in S''} x_i y_i\right] \\ &= \mathbf{Exp}_{x,y}\left[\prod_{i \in S \triangle S''} x_i \cdot \prod_{i \in S' \triangle S''} y_i\right] \\ &= \mathbf{Exp}_x\left[\prod_{i \in S \triangle S''} x_i\right] \cdot \mathbf{Exp}_y\left[\prod_{i \in S' \triangle S''} y_i\right], \end{aligned} \tag{4}$$

where $A \triangle B$ denotes the symmetric difference between sets $A$ and $B$. (I.e., $i \in A \triangle B$ iff $i$ is in exactly one of $A$ or $B$.) Above, the second equality uses the fact that $x_i^2 = y_i^2 = 1$ (since $x_i, y_i \in \{-1, 1\}$), and the third equality relies on the fact that $x$ and $y$ are independent.

Evaluating $\mathbf{Exp}_x\left[\prod_{i \in S \triangle S''} x_i\right]$ is easy: if $S = S''$ then the product is empty and so evaluates to 1 regardless of $x$. On the other hand, if $S \neq S''$ then each $x_i$ is equally likely to be 1 or $-1$ and so the expected value of the product is 0. We thus see from Equation (4) that $\mathbf{Exp}_{x,y}\left[\chi_S(x)\,\chi_{S'}(y)\,\chi_{S''}(x \circ y)\right] = 0$ unless $S = S' = S''$, in which case the expression evaluates to 1. Working back through Equations (3) and (2) gives the claimed result. ■

Given Lemma 3 we can prove Theorem 2 in just a few lines. We have:

$$\begin{aligned} \Pr[\text{linearity test accepts}] &= \frac{1}{2} + \frac{1}{2} \cdot \sum_S \hat{\pi}(S)^3 \\ &\leq \frac{1}{2} + \frac{1}{2} \cdot \max_S\{\hat{\pi}(S)\} \cdot \sum_S \hat{\pi}(S)^2 \\ &= \frac{1}{2} + \frac{1}{2} \cdot \max_S\{\hat{\pi}(S)\}, \end{aligned}$$

using Parseval's theorem. If $\pi$ is distance $\varepsilon$ from linear, this means that $\max_v\{\hat{\pi}(v)\} = 1 - 2\varepsilon$. We conclude that Pr[linearity test accepts] $\leq 1 - \varepsilon$, proving the theorem.

# 3 The Consistency Test

We return to our notation from Section 1.1, where $\vec{v}$ represents a boolean vector of length $n^2$ and we index it using two indices each ranging from 1 to $n$.

Assume $\pi$ is within distance $1/48$ from linear. This means there exists a unique[1] linear function $f$ within distance $1/48$ from $\pi$; we can write $f$ as

$$f(\vec{v}) = \sum_{i,j=1}^{n} \lambda_{i,j} \cdot v_{i,j}$$

for some $\{\lambda_{i,j} \in \{0,1\}\}$. We now want a way to check that these $\{\lambda_{i,j}\}$ are *consistent*; i.e., that $\lambda_{i,j} = \lambda_{i,i} \cdot \lambda_{j,j}$ for all $i, j$. A useful way to view this is to put the $\{\lambda_{i,j}\}$ in an $n \times n$ matrix $M$; i.e.,

$$M \stackrel{\text{def}}{=} \begin{pmatrix} \lambda_{1,1} & \lambda_{1,2} & \cdots & \lambda_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ \lambda_{n,1} & \lambda_{n,2} & \cdots & \lambda_{n,n} \end{pmatrix}.$$

Let $\vec{\lambda} \stackrel{\text{def}}{=} (\lambda_{1,1}, \ldots, \lambda_{n,n})$ (all our vectors will be row vectors). Then consistency is equivalent to:

$$M = \vec{\lambda}^T \vec{\lambda}.$$

(Note once again that $\lambda_{i,i}^2 = \lambda_{i,i}$ since we are working in $\mathbb{F}_2$.) We first show an efficient way to test equality of matrices, and then show how the test can be implemented using access to $\pi$.

**Claim 4** *Let $M, M'$ be two unequal $n \times n$ matrices over $\mathbb{F}_2$. Then*

$$\Pr_{\vec{x},\vec{y} \in \{0,1\}^n}[\vec{x}M\vec{y}^T = \vec{x}M'\vec{y}^T] \leq \frac{3}{4}.$$

**Proof** Note that $\vec{x}M\vec{y}^T - \vec{x}M'\vec{y}^T = \vec{x}(M - M')\vec{y}^T$ and $M - M'$ is a non-zero matrix. So we are interested in the probability that $\vec{x}M''\vec{y}^T = 0$ for non-zero matrix $M''$.

The probability that $M''\vec{y}^T = \vec{0}$ is at most $1/2$. Assuming this does not occur, the probability that $\vec{x}(M''\vec{y}^T) = 0$ is exactly $1/2$. So, the probability that $\vec{x}M''\vec{y}^T = 0$ is at most $3/4$. ■

How can we evaluate $\vec{x}M\vec{y}^T$ and $\vec{x}(\vec{\lambda}^T\vec{\lambda})\vec{y}^T$ given access to $\pi$? Let us assume we have access to $f$, and show how to correct for this later. Given access to $f$, it is easy to compute $\vec{x}M\vec{y}^T$ since

$$\vec{x}M\vec{y}^T = \sum_{i,j=1}^{n} \lambda_{i,j} x_i y_j .$$

Setting $v_{i,j} = x_i y_j$ and querying $f(\vec{v})$ thus gives the desired answer. For the second computation, note that

$$\vec{x}(\vec{\lambda}^T\vec{\lambda})\vec{y}^T = (\vec{x}\vec{\lambda}^T)(\vec{\lambda}\vec{y}^T).$$

[1] That $f$ is unique follows from the fact that any two distinct linear functions are distance $1/2$ from each other.

Setting $v_{i,i} = x_i$ (and $v_{i,j} = 0$ when $i \neq j$), we see that $f(\vec{v}) = \vec{x}\vec{\lambda}^T$; the value $\vec{\lambda}\vec{y}^T$ is computed similarly.

The above assumes we have access to $f$ — but we only have access to $\pi$! However, we said that $\pi$ was within distance $1/48$ from $f$. So we can compute $f(\vec{v})$ (for any $\vec{v}$) by choosing a random "shift" $\vec{r} \in \{0,1\}^{n^2}$ and computing $\tilde{f}(\vec{v}) = \pi(\vec{r}) + \pi(\vec{r} + \vec{v})$. Note that as long as $\pi(\vec{r}) = f(\vec{r})$ and $\pi(\vec{r}+\vec{v}) = f(\vec{r}+\vec{r})$, then $\hat{f}(\vec{v}) = f(\vec{v})$. Thus, for any $\vec{v}$ we compute the correct value of $f(\vec{v})$ except with probability $2/48 = 1/24$. This technique is called *self-correction*.

### 3.1 In Summary

To summarize, we perform the following *consistency test*:

1. Choose random $\vec{x}, \vec{y}$.
2. Using self-correction and the approach described above, compute $\vec{x}M\vec{y}^T$.
3. Using self-correction and the approach described above, compute $\vec{x}(\vec{\lambda}^T\vec{\lambda})\vec{y}^T$.
4. Accept if and only if the two values thus computed are equal.

Note that step 2 requires one call to $f$, so it requires two calls to $\pi$. Step 3 requires two calls to $f$, so it requires four calls to $\pi$.

We may now state the main result of this section (again, we have not tried to optimize constants):

**Theorem 5** *Assume $\pi$ is within distance $1/48$ of a linear function $f$. If $f$ is not consistent (in the sense described above), then the consistency test rejects with probability at least $1/8$.*

**Proof** Assuming the test correctly computes $\vec{x}M\vec{y}^T$ and $\vec{x}(\vec{\lambda}^T\vec{\lambda})\vec{y}^T$, the test will accept with probability at most $3/4$ (by Claim 4). The probability that one of the six calls to $\pi$ results in an incorrect value for $f$ is at most $6/48 = 1/8$ (using the fact that $\pi$ and $f$ disagree on at most a $1/48$ fraction of their points, and applying a union bound). So, the probability of acceptance is at most $3/4 + 1/8$ and the theorem follows. ■

## 4 The Satisfiability Test

Assume $\pi$ is within distance $1/48$ from the linear function

$$f(\vec{v}) = \sum_{i,j=1}^{n} \lambda_{i,j} v_{i,j}$$

and furthermore that $f$ is consistent (i.e., the $\{\lambda_{i,j}\}$ satisfy $\lambda_{i,j} = \lambda_{i,i} \cdot \lambda_{j,j}$ for all $i, j$). We view $\pi$ an encoding an assignment $\vec{a} = (\lambda_{1,1}, \lambda_{2,2}, \ldots, \lambda_{n,n})$. We now want to check that this assignment is a satisfying assignment for the given system of equations. (Indeed, note that until this point everything we have done has been independent of the system of equations whose satisfiability we are interested in!)

Our set of equations (cf. Eq (1)) can be written as:

$$\left\{ c^{(k)} + \sum_{i,j=1}^{n} c_{i,j}^{(k)} \cdot x_i x_j = 0 \right\}_{k=1}^{m},$$

and so we want to verify whether

$$y_k \stackrel{\text{def}}{=} c^{(k)} + \sum_{i,j=1}^{n} c_{i,j}^{(k)} \cdot a_i a_j$$

is equal to 0 for all $k \in [1, m]$. If we let $\vec{y} \stackrel{\text{def}}{=} (y_1, \ldots, y_m)$, then we want to check whether $\vec{y}$ is the 0-vector. We can't check every position individually since this will require too many queries to $\pi$. What we will do instead is to look at the dot product of $\vec{y}$ with a random vector: if $\vec{y} = \vec{0}$ then this dot product will always be 0, but if $\vec{y} \neq 0$ then the dot product will be 1 with probability 1/2.

Taking the dot product of $\vec{y}$ with a random vector is equivalent to choosing a random subset $S \subseteq [m]$ and looking at the sum. That is,

$$\begin{aligned}
\sum_{k \in S} y_k &= \sum_{k \in S} \left( c^{(k)} + \sum_{i,j=1}^{n} c_{i,j}^{(k)} \cdot a_i a_j \right) \\
&= \sum_{k \in S} c^{(k)} + \sum_{k \in S} \sum_{i,j=1}^{n} c_{i,j}^{(k)} \cdot a_i a_j \\
&= \sum_{k \in S} c^{(k)} + \sum_{i,j=1}^{n} a_i a_j \cdot \left( \sum_{k \in S} c_{i,j}^{(k)} \right).
\end{aligned}$$

We can evaluate the first term on our own, and will obtain the second term by evaluating $f(\vec{v})$ where

$$v_{i,j} = \sum_{k \in S} c_{i,j}^{(k)}.$$

To obtain this value $f(\vec{v})$, we will again use self-correction as in the previous section.

In total, we make two queries to $\pi$ and achieve the following (again, constants have not been optimized):

**Theorem 6** *Assume $\pi$ is within distance 1/48 of a linear function $f$ and that $f$ is consistent (as defined previously). Then if the system of equations is not satisfiable, the satisfiability test rejects with probability at least 1/8.*

**Proof** If the test correctly computes $f(\vec{v})$, it accepts with probability 1/2. The probability that one of the two calls to $\pi$ results in an incorrect value for $f$ is at most 2/48 (as in the previous theorem). So, the probability of acceptance is at most $1/2 + 2/48$ and the theorem follows. ■

## 5 Putting it all Together

We summarize the PCP system. Given a system of equations and access to an oracle $\pi$, the verifier proceeds as follows:

- Perform the linearity test (3 queries to $\pi$).
- Perform the consistency test (6 queries to $\pi$).
- Perform the satisfiability test (2 queries to $\pi$).
- Accept only if all the above tests succeed.

If the system of equations is satisfiable, then there exists a proof string $\pi$ for which the above test accepts with probability 1. We claim that if the system of equations is *not* satisfiable, then the test will reject with probability at least 1/48 (for any $\pi$). There are three cases: (1) $\pi$ is not within distance 1/48 of a linear function; (2) $\pi$ is within distance 1/48 of a (unique) linear function $f$, but $f$ is not consistent; or (3) $\pi$ is within distance 1/48 of a consistent, linear function $f$, but $f$ does not encode a satisfying assignment. In case (1) the linearity test will reject with probability at least 1/48; in case (2) the consistency test will reject with probability at least 1/8; and in case (3) the satisfiability test will reject with probability at least 1/8.

## Bibliographic Notes

For more on Fourier analysis of boolean functions, see O'Donnell's lecture notes [2], or the online textbook he is currently writing [3]. For a reasonably self-contained proof of the full PCP theorem (including also a proof that $\mathcal{NP} \subseteq \mathsf{PCP}(\mathsf{poly}, 1)$), see Harsha's thesis [1].

## References


[1] P. Harsha. Robust PCPs of Proximity and Shorter PCPs. PhD thesis, MIT, 2004. Available at `http://www.tcs.tifr.res.in/~prahladh/papers/#theses`

[2] R. O'Donnell. Lecture notes for 15-859S: Analysis of Boolean Functions. Available at `http://www.cs.cmu.edu/~odonnell/boolean-analysis`.

[3] R. O'Donnell. `http://www.contrib.andrew.cmu.edu/~ryanod/?page_id=2`

**Notes on Complexity Theory** Last updated: November, 2011

Lecture 23

*Jonathan Katz*

# 1 The Complexity of Counting

## 1.1 The Class $\#\mathcal{P}$

$\mathcal{P}$ captures problems where we can efficiently find an answer; $\mathcal{NP}$ captures problems where we can efficiently verify an answer. *Counting* the number of answers gives rise to the class $\#\mathcal{P}$.

Recall that $L \in \mathcal{NP}$ if there is a (deterministic) Turing machine $M$ running in time polynomial in its first input such that

$$x \in L \Leftrightarrow \exists w\ M(x,w) = 1. \tag{1}$$

The corresponding counting problem is: given $x$, determine the *number* of strings $w$ for which $M(x,w) = 1$. (Note that $x \in L$ iff this number is greater than 0.) An important point is that for a given $L$, there might be several (different) machines for which Eq. (1) holds; when specifying the counting problem, we need to fix not only $L$ but also a specific machine $M$. Sometimes, however, we abuse notation when there is a "canonical" $M$ for some $L$.

We let $\#\mathcal{P}$ denote the class of counting problems corresponding to polynomial-time $M$ as above. The class $\#\mathcal{P}$ can be defined as a function class or a language class; we will follow the book and speak about it as a function class. Let $M$ be a (two-input) Turing machine $M$ that halts on all inputs, and say $M$ runs in time $t(n)$ where $n$ denotes the length of its first input. Let $\#M(x) \stackrel{\text{def}}{=} \left|\left\{w \in \{0,1\}^{t(|x|)} \mid M(x,w) = 1\right\}\right|$. Then:[1]

**Definition 1** *A function $f : \{0,1\}^* \to \mathbb{N}$ is in $\#\mathcal{P}$ if there is a Turing machine $M$ running in time polynomial in its first input such that $f(x) = \#M(x)$.*

We let $\mathcal{FP}$ denote the class of functions computable in polynomial time; this corresponds to the language class $\mathcal{P}$.

Any $f \in \#\mathcal{P}$ defines a natural language $L \in \mathcal{NP}$: letting $M$ be the Turing machine for which $f(x) = \#M(x)$, we can define

$$L = \{x \mid f(x) > 0\}.$$

This view can be used to show that $\#\mathcal{P}$ is at least as hard as $\mathcal{NP}$. Consider, for example, the problem $\#\mathsf{SAT}$ of counting the number of satisfying assignments of a boolean formula. It is easy to see that $\#\mathsf{SAT} \in \#\mathcal{P}$, but $\#\mathsf{SAT}$ is not in $\mathcal{FP}$ unless $\mathcal{P} = \mathcal{NP}$ (since being able to count the number of solutions clearly implies the ability to determine existence of a solution). Interestingly, it is also possible for a counting problem to be hard even when the corresponding decision problem is easy. (Actually, it is trivial to come up with "cooked up" examples where this is true. What is

[1] For completeness, we also discuss how $\#\mathcal{P}$ can be defined as a language class. For the purposes of this footnote only, let $\#\mathcal{FP}$ denote the function class (as defined above). Then language class $\#\mathcal{P}$ can be defined as: $L \in \#\mathcal{P}$ if there is a Turing machine $M$ running in time polynomial in its first input such that $L = \{(x,k) \mid \#M(x) \leq k\}$. We use inequality rather than equality in this definition to ensure that $\#\mathcal{P} = \mathcal{P}^{\#\mathcal{FP}}$ and $\#\mathcal{FP} = \mathcal{FP}^{\#\mathcal{P}}$.

interesting is that there are many natural examples.) For example, let #cycle be the problem of counting the number of cycles in a directed graph. Note that #cycle $\in \#\mathcal{P}$, and the corresponding decision problem is in $\mathcal{P}$. But:

**Claim 1** *If $\mathcal{P} \neq \mathcal{NP}$, then #cycle $\notin \mathcal{FP}$.*

**Proof** (Sketch) If #cycle $\in \mathcal{FP}$ then we can detect the existence of Hamiltonian cycles in polynomial time. (Deciding Hamiltonicity is a classic $\mathcal{NP}$-complete problem.) Given a graph $G$, form a new graph $G'$ by replacing each edge $(u, v)$ with a "gadget" that introduces $2^{n \log n}$ paths from $u$ to $v$. If $G$ has a Hamiltonian cycle, then $G'$ has at least $\left(2^{n \log n}\right)^n = n^{n^2}$ cycles; if $G$ does not have a Hamiltonian cycle then its longest cycle has length at most $n-1$, and it has at most $n^{n-1}$ cycles; thus, $G'$ has at most $\left(2^{n \log n}\right)^{n-1} \cdot n^{n-1} < n^{n^2}$ cycles. ■

There are two approaches, both frequently encountered, that can be used to define (different notions of) $\#\mathcal{P}$-completeness. We say a function $g \in \#\mathcal{P}$ is *$\#\mathcal{P}$-complete under parsimonious reductions* if for every $f \in \#\mathcal{P}$ there is a polynomial-time computable function $\phi$ such that $f(x) = g(\phi(x))$ for all $x$. (A more general, but less standard, definition would allow for two polynomial-time computable functions $\phi, \phi'$ such that $f(x) = \phi'(g(\phi(x)))$.) This is roughly analogous to a Karp reduction. An alternative definition is that $g \in \#\mathcal{P}$ is *$\#\mathcal{P}$-complete under oracle reductions* if for every $f \in \#\mathcal{P}$ there is a polynomial-time Turing machine $M$ such that $f$ is computable by $M^g$. (In other words, $\#\mathcal{P} \subseteq \mathcal{FP}^g$.) This is analogous to a Cook reduction.

Given some $g \in \#\mathcal{P}$, denote by $L_g$ the $\mathcal{NP}$-language corresponding to $g$ (see above). It is not hard to see that if $g$ is $\#\mathcal{P}$-complete under parsimonious reductions then $L_g$ is $\mathcal{NP}$-complete. As for the converse, although no general result is known, one can observe that most Karp reductions are parsimonious; in particular, #SAT is $\#\mathcal{P}$-complete under parsimonious reductions. $\#\mathcal{P}$-completeness under oracle reductions is a much more liberal definition; as we will see in the next section, it is possible for $g$ to be $\#\mathcal{P}$-complete under Cook reductions even when $L_g \in \mathcal{P}$.

### 1.2 $\#\mathcal{P}$-Completeness of Computing the Permanent

Let $A = \{a_{i,j}\}$ be an $n \times n$ matrix over the integers. The *permanent* of $A$ is defined as:

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

where $S_n$ is the set of all permutations on $n$ elements. This formula is very similar to the formula defining the *determinant* of a matrix; the difference is that in the case of the determinant there is an extra factor of $(-1)^{\mathrm{sign}(\pi)}$. Nevertheless, although the determinant can be computed in polynomial time, computing the permanent (even of boolean matrices) is $\#\mathcal{P}$-complete.

We should say a word about why computing the permanent is in $\#\mathcal{P}$ (since it does not seem to directly correspond to a counting problem). The reason is that computing the permanent is equivalent to (at least) two other problems on graphs. For the case when $A$ is a boolean matrix, we may associate $A$ with a bipartite graph $G_A$ having $n$ vertices in each component, where there is an edge from vertex $i$ (in the left component) to vertex $j$ (in the right component) iff $a_{i,j} = 1$. Then $\mathrm{perm}(A)$ is equal to the number of perfect matchings in $G_A$. For the case of a general integer matrices, we may associate any such matrix $A$ with an $n$-vertex weighted, directed graph $G_A$ (allowing self-loops) by viewing $A$ as a standard adjacency matrix. A *cycle cover* in $G_A$ is a set

of edges such that each vertex has exactly one incoming and outgoing edge in this set. (Any cycle cover corresponds to a permutation $\pi$ on $[n]$ such that $(i, \pi(i))$ is an edge for all $i$.) The *weight* of a cycle cover is the product of the weight of the edges it contains. Then $\text{perm}(A)$ is equal to the sum of the weights of the cycle covers of $G_A$. (For boolean matrices, $\text{perm}(A)$ is just the number of cycle covers of $G_A$.)

Determining *existence* of a perfect matching, or of a cycle cover, can be done in polynomial time; it is *counting* the number of solutions that is hard:

**Theorem 2** *Permanent for boolean matrices is $\#\mathcal{P}$-complete under oracle reductions.*

The proof is quite involved and so we skip it; a full proof can be found in [1, Section 17.3.1].

## References

[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

**Notes on Complexity Theory** Last updated: November, 2011

# Lecture 24

*Jonathan Katz*

## 1 The Complexity of Counting

We explore three results related to hardness of counting. Interestingly, at their core each of these results relies on a simple — yet powerful — technique due to Valiant and Vazirani.

### 1.1 Hardness of Unique-SAT

Does SAT become any easier if we are guaranteed that the formula we are given has at most *one* solution? Alternately, if we are guaranteed that a given boolean formula has a unique solution does it become any easier to find it? We show here that this is not likely to be the case.

Define the following promise problem:

$$\begin{aligned} \mathsf{USAT} &\stackrel{\text{def}}{=} \{\phi : \phi \text{ has exactly one satisfying assignment}\} \\ \overline{\mathsf{USAT}} &\stackrel{\text{def}}{=} \{\phi : \phi \text{ is unsatisfiable}\}. \end{aligned}$$

Clearly, this problem is in promise-$\mathcal{NP}$. We show that if it is in promise-$\mathcal{P}$, then $\mathcal{NP} = \mathcal{RP}$. We begin with a lemma about pairwise-independent hashing.

**Lemma 1** *Let $S \subseteq \{0,1\}^n$ be an arbitrary set with $2^m \leq |S| \leq 2^{m+1}$, and let $H_{n,m+2}$ be a family of pairwise-independent hash functions mapping $\{0,1\}^n$ to $\{0,1\}^{m+2}$. Then*

$$\Pr_{h \in H_{n,m+2}}[\textit{there is a unique } x \in S \textit{ with } h(x) = 0^{m+2}] \geq 1/8.$$

**Proof** Let $\mathbf{0} \stackrel{\text{def}}{=} 0^{m+2}$, and let $p \stackrel{\text{def}}{=} 2^{-(m+2)}$. Let $N$ be the random variable (over choice of random $h \in H_{n,m+2}$) denoting the number of $x \in S$ for which $h(x) = \mathbf{0}$. Using the inclusion/exclusion principle, we have

$$\begin{aligned} \Pr[N \geq 1] &\geq \sum_{x \in S} \Pr[h(x) = \mathbf{0}] - \frac{1}{2} \cdot \sum_{x \neq x' \in S} \Pr[h(x) = h(x') = \mathbf{0}] \\ &= |S| \cdot p - \binom{|S|}{2} p^2, \end{aligned}$$

while $\Pr[N \geq 2] \leq \sum_{x \neq x' \in S} \Pr[h(x) = h(x') = \mathbf{0}] = \binom{|S|}{2} p^2$. So

$$\Pr[N = 1] = \Pr[N \geq 1] - \Pr[N \geq 2] \geq |S| \cdot p - 2 \cdot \binom{|S|}{2} p^2 \geq |S|p - |S|^2 p^2 \geq 1/8,$$

using the fact that $|S| \cdot p \in [\frac{1}{4}, \frac{1}{2}]$. ■

**Theorem 2 (Valiant-Vazirani)** *If* $(\mathsf{USAT}, \overline{\mathsf{USAT}})$ *is in promise-*$\mathcal{RP}$*, then* $\mathcal{NP} = \mathcal{RP}$*.*

**Proof** If $(\mathsf{USAT}, \overline{\mathsf{USAT}})$ is in promise-$\mathcal{RP}$, then there is a probabilistic polynomial-time algorithm $A$ such that

$$\begin{aligned} \phi \in \mathsf{USAT} &\Rightarrow \Pr[A(\phi) = 1] \geq 1/2 \\ \phi \in \overline{\mathsf{USAT}} &\Rightarrow \Pr[A(\phi) = 1] = 0. \end{aligned}$$

We design a probabilistic polynomial-time algorithm $B$ for $\mathsf{SAT}$ as follows: on input an $n$-variable boolean formula $\phi$, first choose uniform $m \in \{0, \ldots, n-1\}$. Then choose random $h \leftarrow H_{n,m+2}$. Using the Cook-Levin reduction, rewrite the expression $\psi(x) \stackrel{\text{def}}{=} \left(\phi(x) \wedge \left(h(x) = 0^{m+2}\right)\right)$ as a boolean formula $\phi'(x, z)$, using additional variables $z$ if necessary. (Since $h$ is efficiently computable, the size of $\phi'$ will be polynomial in the size of $\phi$. Furthermore, the number of satisfying assignments to $\phi'(x, z)$ will be the same as the number of satisfying assignments of $\psi$.) Output $A(\phi')$.

If $\phi$ is not satisfiable then $\phi'$ is not satisfiable, so $A$ (and hence $B$) always outputs 0. If $\phi$ is satisfiable, with $S$ denoting the set of satisfying assignments, then with probability $1/n$ the value of $m$ chosen by $B$ is such that $2^m \leq |S| \leq 2^{m+1}$. In that case, Lemma 1 shows that with probability at least $1/8$ the formula $\phi'$ will have a unique satisfying assignment, in which case $A$ outputs 1 with probability at least $1/2$. We conclude that when $\phi$ is satisfiable then $B$ outputs 1 with probability at least $1/16n$. ■

## 1.2 Approximate Counting, and Relating $\#\mathcal{P}$ to $\mathcal{NP}$

$\#\mathcal{P}$ is clearly not weaker than $\mathcal{NP}$, since if we can count solutions then we can certainly tell if any exist. Although $\#\mathcal{P}$ is (in some sense) "harder" than $\mathcal{NP}$, we show that any problem in $\#\mathcal{P}$ can be probabilistically *approximated* in polynomial time using an $\mathcal{NP}$ oracle. (This is reminiscent of the problem of reducing search to decision, except that here we are reducing *counting* the number of witness to the decision problem of whether or not a witness exists. Also, we are only obtaining an approximation, and we use randomization.) We focus on the $\#\mathcal{P}$-complete problem $\#\mathsf{SAT}$. Let $\#\mathsf{SAT}(\phi)$ denote the number of satisfying assignments of a boolean formula $\phi$. We show that for any polynomial $p$ there exists a PPT algorithm $A$ such that

$$\Pr\left[\#\mathsf{SAT}(\phi) \cdot \left(1 - \frac{1}{p(|\phi|)}\right) \leq A^{\mathcal{NP}}(\phi) \leq \#\mathsf{SAT}(\phi) \cdot \left(1 + \frac{1}{p(|\phi|)}\right)\right] \geq 1 - 2^{-p(|\phi|)}; \tag{1}$$

that is, $A$ approximates $\#\mathsf{SAT}(\phi)$ (the number of satisfying assignments to $\phi$) to within a factor $(1 \pm \frac{1}{p(|\phi|)})$ with high probability.

The first observation is that it suffices to obtain a constant-factor approximation. Indeed, say we have an algorithm $B$ such that

$$\frac{1}{64} \cdot \#\mathsf{SAT}(\phi) \leq B^{\mathcal{NP}}(\phi) \leq 64 \cdot \#\mathsf{SAT}(\phi). \tag{2}$$

(For simplicity we assume $B$ always outputs an approximation satisfying the above; any failure probability of $B$ propagates in the obvious way.) We can construct an algorithm $A$ satisfying (1) as follows: on input $\phi$, set $q = \log 64 \cdot p(|\phi|)$ and compute $t = B(\phi')$ where

$$\phi' \stackrel{\text{def}}{=} \textstyle\bigwedge_{i=1}^{q} \phi(x_i)\,,$$

and the $x_i$ denote independent sets of variables. $A$ then outputs $t^{1/q}$.

Letting $N$ (resp., $N'$) denote the number of satisfying assignments to $\phi$ (resp., $\phi'$), note that $N' = N^q$. Since $t$ satisfies $\frac{1}{64} \cdot N' \leq t \leq 64 \cdot N'$, the output of $A$ lies in the range

$$\left[2^{-1/p(|\phi|)} \cdot N, \quad 2^{1/p(|\phi|)} \cdot N\right] \subseteq \left[\left(1 - \frac{1}{p(|\phi|)}\right) \cdot N, \quad \left(1 + \frac{1}{p(|\phi|)}\right) \cdot N\right],$$

as desired. In the last step, we use the following inequalities which hold for all $x \geq 1$:

$$\left(\frac{1}{2}\right)^{1/x} \geq \left(1 - \frac{1}{x}\right) \quad \text{and} \quad 2^{1/x} \leq \left(1 + \frac{1}{x}\right).$$

The next observation is that we can obtain a constant-factor approximation by solving the promise problem $(\Pi_Y, \Pi_N)$ given by:

$$\begin{aligned} \Pi_Y &\stackrel{\text{def}}{=} \{(\phi, k) \mid \#\mathsf{SAT}(\phi) > 8k\} \\ \Pi_N &\stackrel{\text{def}}{=} \{(\phi, k) \mid \#\mathsf{SAT}(\phi) < k/8\}. \end{aligned}$$

Given an algorithm $C$ solving this promise problem, we can construct an algorithm $B$ satisfying (2) as follows. (Once again, we assume $C$ is deterministic; if $C$ errs with non-zero probability we can handle it in the straightforward way.) On input $\phi$ do:

- Set $i = 0$.
- While $M((\phi, 8^i)) = 1$, increment $i$.
- Return $8^{i-\frac{1}{2}}$.

Let $i^*$ be the value of $i$ at the end of the algorithm, and set $\alpha = \log_8 \#\mathsf{SAT}(\phi)$. In the second step, we know that $M((\phi, 8^i))$ outputs 1 as long as $\#\mathsf{SAT}(\phi) > 8^{i+1}$ or, equivalently, $\alpha > i+1$. So we end up with an $i^*$ satisfying $i^* \geq \alpha - 1$. We also know that $M((\phi, 8^i))$ will output 0 whenever $i > \alpha + 1$ and so the algorithm above must stop at the first (integer) $i$ to satisfy this. Thus, $i^* \leq \alpha + 2$. Putting this together, we see that our output value satisfies:

$$\#\mathsf{SAT}(\phi)/64 < 8^{i^* - \frac{1}{2}} < 64 \cdot \#\mathsf{SAT}(\phi),$$

as desired. (Note that we assume nothing about the behavior of $M$ when $(\phi, 8^i) \notin \Pi_Y \cup \Pi_N$.)

Finally, we show that we can probabilistically solve $(\Pi_Y, \Pi_N)$ using an $\mathcal{NP}$ oracle. This just uses another application of the Valiant-Vazirani technique. Here we rely on the following lemma:

**Lemma 3** *Let $H_{n,m}$ be a family of pairwise-independent hash functions mapping $\{0,1\}^n$ to $\{0,1\}^m$, and let $\varepsilon > 0$. Let $S \subseteq \{0,1\}^n$ be arbitrary with $|S| \geq \varepsilon^{-3} \cdot 2^m$. Then:*

$$\Pr_{h \in H_{n,m}}\left[(1-\varepsilon) \cdot \frac{|S|}{2^m} \leq |\{x \in S \mid h(x) = 0^m\}| \leq (1+\varepsilon) \cdot \frac{|S|}{2^m}\right] > 1 - \varepsilon.$$

**Proof** Define for each $x \in S$ an indicator random variable $\delta_x$ such that $\delta_x = 1$ iff $h(x) = 0^m$ (and 0 otherwise). Note that the $\delta_x$ are pairwise independent random variables with expectation $2^{-m}$ and variance $2^{-m} \cdot (1 - 2^{-m})$. Let $Y \stackrel{\text{def}}{=} \sum_{x \in S} \delta_x = |\{x \in S \mid h(x) = 0^m\}|$. The expectation of $Y$ is $|S|/2^m$, and its variance is $\frac{|S|}{2^m} \cdot (1 - 2^{-m})$ (using pairwise independent of the $\delta_x$). Using Chebychev's inequality, we obtain:

$$\begin{aligned} \Pr\left[(1-\varepsilon) \cdot \mathbf{Exp}[Y] \leq Y \leq (1+\varepsilon) \cdot \mathbf{Exp}[Y]\right] &= \Pr\left[|Y - \mathbf{Exp}[Y]| \leq \varepsilon \cdot \mathbf{Exp}[Y]\right] \\ &\geq 1 - \frac{\mathbf{Var}[Y]}{(\varepsilon \cdot \mathbf{Exp}[Y])^2} \\ &= 1 - \frac{(1-2^{-m}) \cdot 2^m}{\varepsilon^2 \cdot |S|}, \end{aligned}$$

which is greater than $1 - \varepsilon$ for $|S|$ as stated in the proposition. ■

The algorithm solving $(\Pi_Y, \Pi_N)$ is as follows. On input $(\phi, k)$ with $k > 1$ (note that a solution is trivial for $k = 1$), set $m = \lfloor \log k \rfloor$, choose a random $h$ from $H_{n,m}$, and then query the $\mathcal{NP}$ oracle on the statement $\phi'(x) \stackrel{\text{def}}{=} (\phi(x) \wedge (h(x) = 0^m))$ and output the result. An analysis follows.

**Case 1:** $(\phi, k) \in \Pi_Y$, so $\#\mathsf{SAT}(\phi) > 8k$. Let $S_\phi = \{x \mid \phi(x) = 1\}$. Then $|S_\phi| > 8k \geq 8 \cdot 2^m$. So:

$$\begin{aligned} \Pr\left[\phi' \in \mathsf{SAT}\right] &= \Pr\left[\{x \in S_\phi : h(x) = 0^m\} \neq \emptyset\right] \\ &\geq \Pr\left[\,|\{x \in S_\phi : h(x) = 0^m\}| \geq 4\right] \;\geq\; \frac{1}{2}, \end{aligned}$$

which we obtain by applying Lemma 3 with $\varepsilon = \frac{1}{2}$.

**Case 2:** $(\phi, k) \in \Pi_N$, so $\#\mathsf{SAT}(\phi) < k/8$. Let $S_\phi$ be as before. Now $|S_\phi| < k/8 \leq 2^m/4$. So:

$$\begin{aligned} \Pr\left[\phi' \in \mathsf{SAT}\right] &= \Pr\left[\{x \in S_\phi : h(x) = 0^m\} \neq \emptyset\right] \\ &\leq \sum_{x \in S_\phi} \Pr\left[h(x) = 0^m\right] \\ &< \frac{2^m}{4} \cdot 2^{-m} \;=\; \frac{1}{4}, \end{aligned}$$

where we have applied a union bound in the second step. We thus have a constant gap in the acceptance probabilities when $\phi \in \Pi_Y$ vs. when $\phi \in \Pi_N$; this gap can be amplified as usual.

### 1.3 Toda's Theorem

The previous section may suggest that $\#\mathcal{P}$ is not "much stronger" than $\mathcal{NP}$, in the sense that $\#\mathcal{P}$ can be closely approximated given access to an $\mathcal{NP}$ oracle. Here, we examine this more closely, and show the opposite: while *approximating* the number of solutions may be "easy" (given an $\mathcal{NP}$ oracle), determining the *exact* number of solutions appears to be much more difficult.

Toward this, we first introduce the class $\oplus\mathcal{P}$ ("parity $\mathcal{P}$"):

**Definition 1** *A function $f : \{0,1\}^* \to \{0,1\}$ is in $\oplus\mathcal{P}$ if there is a Turing machine $M$ running in time polynomial in its first input such that $f(x) = \#M(x) \bmod 2$.*

Note that if $f \in \oplus\mathcal{P}$ then $f$ is just the least-significant bit of some function $\bar{f} \in \#\mathcal{P}$. The class $\oplus\mathcal{P}$ does not represent any "natural" computational problem. Nevertheless, it is natural to study

it because (1) it nicely encapsulates the difficulty of computing functions in $\#\mathcal{P}$ *exactly* (i.e., down to the least-significant bit), and (2) it can be seen as a generalization of the unique-SAT example discussed previously (where the difficulty there is determining whether a boolean formula has 0 solutions or 1 solution).

A function $g \in \oplus\mathcal{P}$ is $\oplus\mathcal{P}$-complete (under parsimonious reductions) if for every $f \in \#\mathcal{P}$ there is a polynomial-time computable function $\phi$ such that $f(x) = g(\phi(x))$ for all $x$. If $\bar{g} \in \#\mathcal{P}$ is $\#\mathcal{P}$-complete under parsimonious reductions, then the least-significant bit of $\bar{g}$ is $\oplus\mathcal{P}$-complete under parsimonious reductions. For notational purposes it is easier to treat $\oplus\mathcal{P}$ as a language class, in the natural way. (In particular, if $f \in \oplus\mathcal{P}$ as above then we obtain the language $L_f = \{x : f(x) = 1\}$.) In this sense, $\oplus\mathcal{P}$-completeness is just the usual notion of a Karp reduction. Not surprisingly,

$$\oplus\mathsf{SAT} \stackrel{\text{def}}{=} \{\phi : \phi \text{ has an odd number of satisfying assignments}\}$$

is $\oplus\mathcal{P}$-complete. Note that $\phi \in \oplus\mathsf{SAT}$ iff $\sum_x \phi(x) = 1 \bmod 2$ (where we let $\phi(x) = 1$ if $x$ satisfies $\phi$, and $\phi(x) = 0$ otherwise).

A useful feature of $\oplus\mathcal{P}$ is that it can be "manipulated" arithmetically in the following sense:

- $(\phi \in \oplus\mathsf{SAT}) \bigwedge (\phi' \in \oplus\mathsf{SAT}) \Leftrightarrow \phi \wedge \phi' \in \oplus\mathsf{SAT}$. This follows because

  $$\sum_{x,x'} \phi(x) \wedge \phi'(x') = \sum_{x,x'} \phi(x) \cdot \phi'(x') = \left(\sum_x \phi(x)\right) \cdot \left(\sum_{x'} \phi'(x')\right),$$

  and hence the number of satisfying assignments of $\phi \wedge \phi'$ is the product of the number of satisfying assignments of each of $\phi, \phi'$.

- Let $\phi, \phi'$ be formulas, where without loss of generality we assume they both have the same number $n$ of variables (this can always be enforced, without changing the number of satisfying assignments, by "padding" with additional variables that are forced to be 0 in any satisfying assignment). Define the formula $\phi + \phi'$ on $n+1$ variables as follows:

  $$(\phi + \phi')(z, x) = \big((z = 0) \wedge \phi(x)\big) \vee \big((z = 1) \wedge \phi'(x)\big).$$

  Note that the number of satisfying assignments of $\phi + \phi'$ is the sum of the number of satisfying assignments of each of $\phi, \phi'$. In particular, $(\phi + \phi') \in \oplus\mathsf{SAT}$ iff *exactly one* of $\phi, \phi' \in \oplus\mathsf{SAT}$.

- Let '1' stand for some canonical boolean formula that has exactly one satisfying assignment. Then $\phi \notin \oplus\mathsf{SAT} \Leftrightarrow (\phi + 1) \in \oplus\mathsf{SAT}$.

- Finally, $(\phi \in \oplus\mathsf{SAT}) \bigvee (\phi' \in \oplus\mathsf{SAT}) \Leftrightarrow (\phi + 1) \bigwedge (\phi' + 1) + 1 \in \oplus\mathsf{SAT}$.

We use the above tools to prove the following result:

**Theorem 4 (Toda's theorem)** $\mathsf{PH} \subseteq \mathcal{P}^{\#\mathcal{P}}$.

The proof of Toda's theorem proceeds in two steps, each of which is a theorem in its own right.

**Theorem 5** *Fix any $c \in \mathbb{N}$. There is a probabilistic polynomial-time algorithm $A$ such that for any quantified boolean formula $\psi$ with $c$ alternations, the following holds:*

$$\begin{aligned} \psi \text{ is true} &\Rightarrow \Pr[A(1^m, \psi) \in \oplus\mathsf{SAT}] \geq 1 - 2^{-m} \\ \psi \text{ is false} &\Rightarrow \Pr[A(1^m, \psi) \in \oplus\mathsf{SAT}] \leq 2^{-m}. \end{aligned}$$

*As a corollary,* $\mathsf{PH} \subseteq \mathcal{BPP}^{\oplus\mathcal{P}}$.

**Proof** It suffices to consider quantified boolean formulae beginning with an '$\exists$' quantifier. Indeed, say we have some algorithm $A'$ that works in that case. If $\psi$ begins with a '$\forall$' quantifier then $\neg\psi$ can be written as a quantified boolean formula beginning with an '$\exists$' quantifier; moreover, $\psi$ is true iff $\neg\psi$ is false. Thus, defining $A(1^m, \psi)$ to return $A'(1^m, \neg\psi) + 1$ gives the desired result.

The proof is by induction on $c$. For $c = 1$ we apply the Valiant-Vazirani result plus amplification. Specifically, let $\psi$ be a statement with only a single $\exists$ quantifier. The Valiant-Vazirani technique gives us a probabilistic polynomial-time algorithm $B$ such that:

$$\begin{aligned} \psi \text{ is true} &\Rightarrow \Pr[B(\psi) \in \oplus\mathsf{SAT}] \geq 1/8n \\ \psi \text{ is false} &\Rightarrow \Pr[B(\psi) \in \oplus\mathsf{SAT}] = 0, \end{aligned}$$

where $n$ is the number of variables in $\psi$. Algorithm $A(1^m, \psi)$ runs $B(\psi)$ a total of $\ell = O(mn)$ times to obtain formulae $\phi_1, \ldots, \phi_\ell$; it then outputs the formula $\Phi = 1 + \bigwedge_i (\phi_i + 1)$. Note that $\bigvee_i (\phi_i \in \oplus\mathsf{SAT}) \Leftrightarrow \Phi \in \oplus\mathsf{SAT}$; hence

$$\begin{aligned} \psi \text{ is true} &\Rightarrow \Pr[A(1^m, \psi) \in \oplus\mathsf{SAT}] \geq 1 - 2^{-m} \\ \psi \text{ is false} &\Rightarrow \Pr[A(1^m, \psi) \in \oplus\mathsf{SAT}] = 0. \end{aligned}$$

In fact, it can be verified that the above holds even if $\psi$ has some free variables $x$. In more detail, let $\psi_x$ be a statement (with only a single $\exists$ quantifier) that depends on free variables $x$.[1] The Valiant-Vazirani technique gives us a probabilistic polynomial-time algorithm $B$ outputting a statement $\phi_x$ (with free variables $x$) such that, for each $x$:

$$\begin{aligned} x \text{ is such that } \psi \text{ is true} &\Rightarrow \Pr[\phi_x \in \oplus\mathsf{SAT}] \geq 1/8n \\ x \text{ is such that } \psi \text{ is false} &\Rightarrow \Pr[\phi_x \in \oplus\mathsf{SAT}] = 0. \end{aligned}$$

Repeating this $O(n \cdot (m + |x|))$ times and proceeding as before gives a formula $\Phi_x$ where, for *all* $x$,

$$\begin{aligned} x \text{ is such that } \psi \text{ is true} &\Rightarrow \Pr[\Phi_x \in \oplus\mathsf{SAT}] \geq 1 - 2^{-m} \\ x \text{ is such that } \psi \text{ is false} &\Rightarrow \Pr[\Phi_x \in \oplus\mathsf{SAT}] = 0. \end{aligned}$$

For the inductive step, write $\psi = \exists x : \psi'_x$, where $\psi'_x$ is a quantified boolean formula with $c - 1$ alternations having $n$ free variables $x$. Applying the inductive hypothesis, we can transform $\psi'_x$ into a boolean formula $\Phi'_x$ such that, for all $x$:

$$x \text{ is such that } \psi'_x \text{ is true} \Rightarrow \Phi'_x \in \oplus\mathsf{SAT} \tag{3}$$

$$x \text{ is such that } \psi'_x \text{ is false} \Rightarrow \Phi'_x \notin \oplus\mathsf{SAT} \tag{4}$$

except with probability at most $2^{-(m+1)}$. We assume the above hold for the rest of the proof.

The key observation is that the Valiant-Vazirani technique applies here as well. We can output, in polynomial time, a boolean formula $\beta$ such that with probability at least $1/8n$,

$$\begin{aligned} \exists x : \psi'_x &\Rightarrow \exists x : \Phi'_x \in \oplus\mathsf{SAT} \Rightarrow \left|\left\{x : \left(\Phi'_x \in \oplus\mathsf{SAT}\right) \wedge \beta(x)\right\}\right| = 1 \bmod 2 \\ \nexists x : \psi'_x &\Rightarrow \nexists x : \Phi'_x \notin \oplus\mathsf{SAT} \Rightarrow \left|\left\{x : \left(\Phi'_x \in \oplus\mathsf{SAT}\right) \wedge \beta(x)\right\}\right| = 0 \bmod 2. \end{aligned}$$

[1] E.g., $\psi_x$ may be of the form "$\exists z : (z \vee \bar{x}) \wedge x$", in which case $\psi_0$ is false and $\psi_1$ is true.

Assume $\beta$ is such that the above hold. Let $[P]$ evaluate to 1 iff predicate $P$ is true. Then $\exists x : \psi'_x$ implies

$$\begin{aligned} 1 &= \sum_x \left[ \left(\Phi'_x \in \oplus\mathsf{SAT}\right) \wedge \beta(x) \right] \bmod 2 \\ &= \sum_x \left[ \left( 1 = \sum_z \Phi'_x(z) \bmod 2 \right) \wedge \beta(x) \right] \bmod 2 \\ &= \sum_x \left[ 1 = \sum_z \left( \beta(x) \wedge \Phi'_x(z) \right) \bmod 2 \right] \bmod 2 \\ &= \sum_{x,z} \left( \beta(x) \wedge \Phi'_x(z) \right) \bmod 2, \end{aligned}$$

and similarly $\nexists x : \psi'_x$ implies

$$0 = \sum_{x,z} \left( \beta(x) \wedge \Phi'_x(z) \right) \bmod 2.$$

Letting $\phi(x,z) \stackrel{\text{def}}{=} \beta(x) \wedge \Phi'_x(z)$ (note $\phi$ has no free variables), we conclude that

$$\exists x : \psi'_x \Leftrightarrow \phi \in \oplus\mathsf{SAT}.$$

The above all holds with probability at least $1/8n$. But we may amplify as before to obtain $\Phi$ such that

$$\begin{aligned} \exists x : \psi'_x &\Rightarrow \Pr[\Phi \in \oplus\mathsf{SAT}] \geq 1 - 2^{-(m+1)} \\ \nexists x : \psi'_x &\Rightarrow \Pr[\Phi \in \oplus\mathsf{SAT}] \leq 2^{-(m+1)}. \end{aligned}$$

Taking into account the error from Equations (3) and (4), we get a total error probability that is bounded by $2^{-m}$. ■

The second step of Toda's theorem shows how to derandomize the above reduction, given access to a $\#\mathcal{P}$ oracle.

**Theorem 6** *$\mathcal{BPP}^{\oplus\mathcal{P}} \subseteq \mathcal{P}^{\#\mathcal{P}}$.*

**Proof** We prove a weaker result, in that we consider only probabilistic Karp reductions to $\oplus\mathcal{P}$. (This suffices to prove Toda's theorem, since the algorithm from the preceding theorem shows that $\mathsf{PH}$ can be solved by such a reduction.) For simplicity, we also only consider derandomization of the specific algorithm $A$ from the previous theorem.

The first observation is that there is a (deterministic) polynomial-time computable transformation $T$ such that if $\phi' = T(\phi, 1^\ell)$ then

$$\begin{aligned} \phi \in \oplus\mathsf{SAT} &\Rightarrow \#\mathsf{SAT}(\phi') = -1 \bmod 2^{\ell+1} \\ \phi \notin \oplus\mathsf{SAT} &\Rightarrow \#\mathsf{SAT}(\phi') = 0 \bmod 2^{\ell+1}. \end{aligned}$$

(See [1, Lemma 17.22] for details.)

Let now $A$ be the randomized reduction from the previous theorem (fixing $m = 2$), so that

$$\begin{aligned} \psi \text{ is true} &\Rightarrow \Pr[A(\psi) \in \oplus\mathcal{P}] \geq 3/4 \\ \psi \text{ is false} &\Rightarrow \Pr[A(\psi) \in \oplus\mathcal{P}] \leq 1/4, \end{aligned}$$

where $\psi$ is a quantified boolean formula. Say $A$ uses $t = t(|\psi|)$ random bits. Let $T \circ A$ be the (deterministic) function given by

$$T \circ A(\psi, r) = T(A(\psi; r), 1^t).$$

Finally, consider the polynomial-time predicate $R$ given by

$$R(\psi, (r, x)) = 1 \text{ iff } x \text{ is a satisfying assignment for } T \circ A(\psi, r).$$

Now:

1. If $\psi$ is true then for at least $3/4$ of the values of $r$ the number of satisfying assignments to $T \circ A(\psi, r)$ is equal to $-1$ modulo $2^{t+1}$, and for the remaining values of $r$ the number of satisfying assignments is equal to 0 modulo $2^{t+1}$. Thus

$$\left|\{(r, x) \mid R(\psi, (r, x)) = 1\}\right| \in \{-2^t, \ldots, -3 \cdot 2^t/4\} \bmod 2^{t+1}.$$

2. If $\psi$ is false then for at least $3/4$ of the values of $r$ the number of satisfying assignments to $T \circ A(\psi, r)$ is equal to 0 modulo $2^{t+1}$, and for the remaining values of $r$ the number of satisfying assignments is equal to $-1$ modulo $2^{t+1}$. Thus

$$\left|\{(r, x) \mid R(\psi, (r, x)) = 1\}\right| \in \{-2^t/4, \ldots, 0\} \bmod 2^{t+1}.$$

We can distinguish the two cases above using a single call to the $\#\mathcal{P}$ oracle (first applying a parsimonious reduction from $R(\psi, \cdot)$ to a boolean formula $\phi(\cdot)$). ■

## References


[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

**Notes on Complexity Theory** Last updated: November, 2011

Lecture 25

*Jonathan Katz*

# 1 Time-Bounded Derandomization

Randomization provides unconditional benefits in many settings; examples include cryptography (where random keys are used to provide protection against an adversary) and distributed computing (where randomness can be used as a means to break symmetry between parties). Randomness also appears to help in algorithm design. But is it possible that, from a complexity-theoretic perspective, randomness does *not* help? E.g., might it be the case that every problem that can be solved in randomized polynomial time can also be solved in deterministic polynomial time? (That is, is $\mathcal{P} = \mathcal{BPP}$?) Historically, guided by progress in designing efficient randomized algorithms, most researchers believed that randomness does help. Research over the past 25 years on (time-bounded) *derandomization* has now led many to change their views; the consensus nowadays is that randomization does not help.[1]

One natural approach to derandomize algorithms is to use a *pseudorandom generator* (PRG) that expands a small, truly random input into a larger, random-looking output. In the next section we define PRGs and then describe their application to derandomization. The remainder of these notes will focus on constructing a PRG based on a (plausible) complexity assumption.

# 2 Pseudorandom Generators

A pseudorandom generator $G$ is a deterministic algorithm that expands a short input (often called a "seed") into a larger output. The output of $G$ should "look random"; formally, $G(s)$ (for $s$ chosen uniformly) should be indistinguishable from a uniform string of length $|G(s)|$. We give a formal definition next. (A word on notation: When we write $G : \{0,1\}^{\ell(t)} \to \{0,1\}^t$ we mean that for every integer $t$ and every $s \in \{0,1\}^{\ell(t)}$, we have $|G(s)| = t$.)

**Definition 1** *A function $G : \{0,1\}^{\ell(t)} \to \{0,1\}^t$ is a* (complexity-theoretic) pseudorandom generator *if $G$ can be computed in exponential time (i.e., $G(s)$ can be computed in time $2^{O(|s|)}$) and if for all sufficiently large $t$ the following holds: for any distinguisher (i.e., circuit) $C$ of size at most $t$,*

$$\left| \Pr_{r \leftarrow \{0,1\}^t} [C(r) = 1] - \Pr_{s \leftarrow \{0,1\}^{\ell(t)}} [C(G(s)) = 1] \right| < 1/t.$$

It is worth pointing out several differences between the above definition and that of *cryptographic* pseudorandom generators. (Those who have not seen cryptographic PRGs can skip to the next section.) The primary difference is with respect to the *running time of the PRG* vs. the *running time*

[1] Note that even if randomness "does not help" from a complexity-theoretic standpoint, it may still be the case that it helps from an algorithmic standpoint. Namely, even if $\mathcal{P} = \mathcal{BPP}$ there may exist problems whose solution requires, say, deterministic quadratic time but randomized linear time.

*of the distinguisher*. Simplifying things a bit, in the cryptographic setting honest parties evaluate the PRG and an adversary plays the role of the distinguisher; we would like to keep the running time of the honest parties as small as possible, while simultaneously protecting them against the most powerful class of adversaries possible. In particular, we certainly want to offer protection against adversaries who are at least as powerful as the honest parties; thus, when defining a cryptographic PRG we will always want to consider distinguishers that run in time greater than the time to evaluate the PRG itself. In contrast, we will see that this is not needed for complexity-theoretic applications; thus, it is still meaningful in our context to consider PRGs where the distinguisher runs in less time than is required to evaluate the PRG.

We mention a few other differences; the reason for these differences should become clear in the following section:

- Here we only require that the distinguisher cannot distinguish the pseudorandom distribution from the uniform distribution "too well", i.e., with advantage better than $1/t$. In the cryptographic setting we require the distinguisher's advantage to be negligible.
- A complexity-theoretic PRG may require *exponential* time (in the input length) to compute. In the cryptographic setting, as noted above, evaluating the PRG should be as efficient as possible; we at least require the PRG to be computable in *polynomial* time.
- In the present definition we consider non-uniform distinguishers, while in the usual cryptographic setting one considers only uniform distinguishers.

With the exception of the last point, complexity-theoretic PRGs are weaker than cryptographic PRGs; thus they can be constructed from milder assumptions.

## 2.1 Using a PRG to Derandomize Algorithms

We now show how a complexity-theoretic PRG can be used to derandomize algorithms.

**Theorem 1** *If there is a (complexity-theoretic) pseudorandom generator* $G : \{0,1\}^{\ell(t)} \to \{0,1\}^t$, *then* $\text{BPTIME}(t(n)) \subseteq \text{TIME}(2^{O(\ell(t^2(n)))})$.

**Proof** Fix a language $L \in \text{BPTIME}(t(n))$, and a probabilistic algorithm $A$ running in time $T(n) = O(t(n))$ for which

$$\begin{aligned} x \in L &\Rightarrow \Pr[A(x) = 1] \geq 2/3 \\ x \notin L &\Rightarrow \Pr[A(x) = 1] \leq 1/3. \end{aligned}$$

We construct a deterministic algorithm $B$ deciding $L$. We focus on input lengths $n$ sufficiently large; the behavior of $B$ on a finite number of shorter inputs can be hard-coded into the algorithm.

On input $x \in \{0,1\}^n$, algorithm $B$ sets $t = t(n)$ and does:

1. For each $s \in \{0,1\}^{\ell(t^2)}$, compute $A(x; G(s))$. (Note that $A$ uses at most $T(n)$ random bits, which is less than $|G(s)| = t^2(n)$ for $n$ sufficiently large.)
2. Output the majority value computed in the previous step.

Each iteration of step 1 takes time[2] at most $2^{O(\ell(t^2(n)))} + T(n) = 2^{O(\ell(t^2(n)))}$, and there are $2^{\ell(t^2(n))}$ iterations; thus, $B$ runs in time $2^{O(\ell(t^2(n)))}$. We now show that $B$ correctly decides $L$.

Correctness of $B$ follows once we show that

$$\begin{aligned} x \in L &\Rightarrow \Pr[A(x; G(s)) = 1] > 1/2 \\ x \notin L &\Rightarrow \Pr[A(x; G(s)) = 1] < 1/2, \end{aligned}$$

where the probabilities are over random choice of $s \in \{0,1\}^{\ell(t^2)}$. Fix $x \in \{0,1\}^n$ with $x \in L$. (The argument if $x \notin L$ is analogous.) Consider the distinguisher $C(\cdot) = A(x; \cdot)$. Since $A$ runs in time $T(n)$, there is a circuit of size $o(T^2) = o(t^2)$ computing $C$. But then for $n$ sufficiently large

$$\begin{aligned} \Pr[A(x; G(s)) = 1] &= \Pr[C(G(s)) = 1] \\ &> \Pr[C(r) = 1] - 1/t^2 \qquad \text{(by pseudorandomness of } G) \\ &= \Pr[A(x) = 1] - 1/t^2 \;\geq\; 2/3 - 1/t^2 > 1/2. \end{aligned}$$

This completes the proof. ■

It is worth noting that non-uniformity comes into play in the preceding proof because we want $B$ to be correct on *all* inputs; if there exists an input $x$ on which $B$ is incorrect then we can "hard-wire" $x$ into the distinguisher $C$. The theorem would hold even if we only required $G$ to be indistinguishable for $T = O(t)$-time algorithms taking $n \leq T$ bits of advice. In a different direction, if we only required $B$ to be correct for *efficiently sampleable* inputs then we could work with a uniform notion of PRGs.

**Corollary 2** *If there is a (complexity-theoretic) pseudorandom generator $G : \{0,1\}^{\ell(t)} \to \{0,1\}^t$ with $\ell(t) = O(\log t)$, then $\mathcal{P} = \mathcal{BPP}$.*

**Proof** Take arbitrary $L \in \mathcal{BPP}$. Then $L \in \text{BPTIME}(n^c)$ for some constant $c$. By the previous theorem, $L \in \text{TIME}(2^{O(\ell(n^{2c}))}) = \text{TIME}(2^{O(\log n)}) = \text{TIME}(n^{O(1)}) \subset \mathcal{P}$. ■

# 3 The Nisan-Wigderson PRG

## 3.1 Some Preliminaries

We collect here some results that are used in the next section, but are tangential to the main thrust.

The first lemma follows by a standard hybrid argument.

**Lemma 3** *Fix $G : \{0,1\}^\ell \to \{0,1\}^t$ and suppose there is a circuit $C$ of size at most $t$ such that*

$$\left| \Pr_{r \leftarrow \{0,1\}^t} [C(r) = 1] - \Pr_{x \leftarrow \{0,1\}^\ell} [C(G(x)) = 1] \right| \geq 1/t.$$

*Then there exists an $i \in \{1, \ldots, t\}$ and a circuit $C'$ of size at most $t$ such that*

$$\Pr_{x \leftarrow \{0,1\}^\ell} [C'\left(G(x)_1 \cdots G(x)_{i-1}\right) = G(x)_i] - \frac{1}{2} \geq 1/t^2.$$

*I.e., $C'$ can predict the $i$th bit of the output of $G$.*

The next lemma is a standard fact we have seen before.

**Lemma 4** *Any function $f : \{0,1\}^k \to \{0,1\}$ can be computed by a circuit of size at most $2^k$.*

[2]Note that $\ell(t) = \Omega(\log t)$ (so $2^{O(\ell(t^2))} = \Omega(T)$); otherwise, there is a trivial distinguisher and $G$ is not a PRG.

### 3.2 From Hardness to Randomness

We will construct a PRG starting from any "suitably hard" computational problem. The starting point here is simple: if a boolean function $f$ is hard to compute (on average) for algorithms running in time $t$ — we formalize this below — then, by definition, $x \| f(x)$ "looks random" to any $t$-time algorithm given $x$. This does not yet give a PRG, but at least indicates the intuition. We first formally define what it means for a function to be hard.

**Definition 2** *A function $f : \{0,1\}^m \to \{0,1\}$ is* $S$-hard *if for all circuits $C$ of size at most $S$,*

$$\left|\Pr_{x \leftarrow \{0,1\}^m}[C(x) = f(x)] - 1/2\right| < 1/S\,.$$

The key to the construction of a PRG is a combinatorial object called a *design*.

**Definition 3** *Fix integers $k, m, \ell$. A collection of sets $\{S_1, \ldots, S_t\}$ with $S_i \subset \{1, \ldots, \ell\}$ is a* $(k, m)$-design *if (1) $|S_i| = m$ for all $i$, and (2) $|S_i \cap S_j| \leq k$ for all $i \neq j$.*

We can specify a set system $\{S_1, \ldots, S_t\}$ with $S_i \subseteq \{1, \ldots, \ell\}$ by a $t \times \ell$ matrix, where row $i$ of the matrix is the characteristic vector for $S_i$. We say such a matrix $A$ is a $(k, m)$-design if the corresponding set system is.

Given a function $f : \{0,1\}^m \to \{0,1\}$, an $\ell$-bit string $x = x_1 \cdots x_\ell$, and a set $S = \{i_1, \ldots, i_m\} \subset \{1, \ldots, \ell\}$, define $f_S(x) = f(x_{i_1} \cdots x_{i_m})$. Given a $t \times \ell$ matrix $A$ corresponding to a set system $\{S_1, \ldots, S_t\}$ with $S_i \subset \{1, \ldots, \ell\}$, define $f_A : \{0,1\}^\ell \to \{0,1\}^t$ as

$$f_A(x) = f_{S_1}(x) \cdots f_{S_t}(x).$$

In the following theorem we construct a "PRG" $G : \{0,1\}^\ell \to \{0,1\}^t$ for some fixed values of $\ell, t$. (It is not quite a PRG since it is not yet a construction for arbitrary outputs length $t$.) We will observe later that, as $t$ varies, the construction is computable in exponential time as required by Definition 2.

**Theorem 5** *Fix integers $t, m, \ell$. Suppose $f : \{0,1\}^m \to \{0,1\}$ is $t^2$-hard, and let $A$ be a $t \times \ell$ matrix that is a $(\log t, m)$-design. Let $f_A : \{0,1\}^\ell \to \{0,1\}^t$ be as above. Then for all circuits $C$ of size at most $t$ we have*

$$\left|\Pr_{r \leftarrow \{0,1\}^t}[C(r) = 1] - \Pr_{x \leftarrow \{0,1\}^\ell}\left[C\left(f_A(x)\right) = 1\right]\right| < 1/t. \tag{1}$$

**Proof** Denote the design corresponding to $A$ by $\{S_1, \ldots, S_t\}$. Fix a circuit $C$ of size at most $t$, and assume toward a contradiction that (1) does not hold. By Lemma 3, this implies the existence of an $i \in \{1, \ldots, t\}$ and a circuit $C'$ of size at most $t$ for which

$$\Pr_{x \leftarrow \{0,1\}^\ell}[C'\left(f_{S_1}(x) \cdots f_{S_{i-1}}(x)\right) = f_{S_i}(x)] - \frac{1}{2} \geq 1/t^2. \tag{2}$$

That is, $C'$ can predict $f_{S_i}(x)$ given $f_{S_1}(x), \ldots, f_{S_{i-1}}(x)$. We construct a circuit $D$ of size at most $t^2$ that computes $f$ with probability better than $1/2 + 1/t^2$, contradicting the assumed hardness of $f$.

For notational convenience, let us assume that $S_i = \{1, \ldots, m\}$. Rewriting (2), we have

$$\Pr_{x_1, \ldots, x_\ell \leftarrow \{0,1\}}[C'\left(f_{S_1}(x) \cdots f_{S_{i-1}}(x)\right) = f(x_1 \cdots x_m)] - \frac{1}{2} \geq 1/t^2\,,$$

where $x = x_1 \cdots x_\ell$. By a standard averaging argument, this implies that there exist some fixed values $\bar{x}_{m+1}, \ldots, \bar{x}_\ell$ for the variables $x_{m+1}, \ldots, x_\ell$ for which

$$\Pr_{x_1,\ldots,x_m \leftarrow \{0,1\}}[C'\left(f_{S_1}(x) \cdots f_{S_{i-1}}(x)\right) = f(x_1 \cdots x_m)] - \frac{1}{2} \geq 1/t^2\,,$$

where now $x = x_1 \cdots x_m \bar{x}_{m+1} \cdots \bar{x}_\ell$. We can express $C'$ as a function $D$ of $x_1, \ldots, x_m$ only by defining $D(x_1 \cdots x_m) = C'\left(f_{S_1}(x) \cdots f_{S_{i-1}}(x)\right)$ (with $x$ as just defined). The size of $D$ is at most the size of $C'$ plus the sizes of the circuits required to compute all the $f_{S_j}(x)$. To get an upper bound on the latter, we use the fact that $A$ is a $(\log t, m)$-design. This implies that each $f_{S_j}(x)$ is a function of at most $\log t$ of the bits $x_1, \ldots, x_m$ (since $S_j$ intersects $S_i = \{1, \ldots, m\}$ in at most $\log t$ positions). Thus, by Lemma 4, each $f_{S_i}$ can be computed using at most $t$ gates; hence $D$ requires at most $t + (t-1) \cdot t = t^2$ gates. This gives the desired contradiction, and completes the proof. ■

To finish the construction we need only show how to build a design with the required parameters. Treating the hard function $f$ as being given, we have some fixed $t, m$ and we want a $(\log t, m)$-design $\{S_1, \ldots, S_t\}$ with $S_i \subset \{1, \ldots, \ell\}$ and where $\ell = \ell(t, m)$ is as small as possible. For $\log t \leq m \leq t$, designs with $\ell = O(m^2)$ exist; see [2]. For $m = O(\log t)$, a better construction is possible.

**Theorem 6** *Fix a constant $c$. There is an algorithm that, given $t$, constructs a $(\log t, c \log t)$-design $\{S_1, \ldots, S_t\}$ with $S_i \subset \{1, \ldots, \ell\}$ and $\ell = O(\log t)$. The algorithm runs in time $\mathsf{poly}(t)$.*

**Proof** Define $m = c \log t$, and set $\ell = d \log t$ where the exact constant $d$ can be derived from the analysis that follows. A greedy algorithm, where we exhaustively search for the next set $S_i$ (with the required properties) given $S_1, \ldots, S_{i-1}$, works. To see this, look at the worst case where $S_1, \ldots, S_{t-1}$ have all been fixed. Consider a random subset $S_t \subset \{1, \ldots, \ell\}$ of size $m$. The expected number of points in which $S_t$ and, say, $S_1$ intersect is $m^2/\ell = O(\log t)$; setting $d$ appropriately, the probability that they intersect in more than $\log t$ points is at most $1/t$. A union bound over the $t-1$ sets that have already been fixed shows that the probability that any one of the existing sets intersects $S_t$ in more than $\log t$ points is less than 1; thus, there exists a set $S_t$ that intersects each of the fixed sets in at most $\log t$ points.

Each set can be chosen in time $\mathsf{poly}(t)$ (because $\ell = O(\log t)$, we can enumerate all $m$-size subsets of $\{1, \ldots, \ell\}$ in time $\mathsf{poly}(t)$); since we choose $t$ sets, the overall running time of the algorithm is polynomial in $t$. ■

The only remaining piece is to let $t$ vary, and observe conditions under which the above pseudorandom generator can be computed in the required time. We say a function $f : \{0,1\}^* \to \{0,1\}$ is $S(n)$-hard if for all sufficiently large $n$ the function $f$ restricted to inputs of size $n$ is $S(n)$-hard.

**Corollary 7** *Suppose there is a function $f$ that can be computed in time $2^{\alpha n}$ but is $2^{\beta n}$-hard for some constants $\alpha, \beta$. Then there is a pseudorandom generator $G : \{0,1\}^{\ell(t)} \to \{0,1\}^t$ with $\ell(t) = O(\log t)$, and $\mathcal{P} = \mathcal{BPP}$.*

**Proof** Set $c = 2/\beta$, and let $\ell(t) = O(\log t)$ be as obtained using Theorem 6 for this value of $c$.

Define $G$ as follows: On input $x \in \{0,1\}^\ell$, where $\ell = \ell(t)$ for some $t$, set $m = c \log t$ and let $f^{(m)}$ denote the restriction of $f$ to inputs of length $m$. Do:

1. Construct a $(\log t, m)$-design $\{S_1, \ldots, S_t\}$ with $S_i \subset \{1, \ldots, \ell\}$. Let $A$ be the matrix corresponding to this design.

2. Compute $f_A^{(m)}(x)$.

By the assumptions of the theorem, for all sufficiently large $t$ the function $f^{(m)}$ is $t^2$-hard. Thus, Theorem 5 implies that the output of $G$ is pseudorandom and all that is left is to analyze the running time of $G$. By Theorem 6, step 1 can be done in $\mathsf{poly}(t) = 2^{O(\ell)}$ time. Step 2 requires $t = 2^{O(\ell)}$ evaluations of $f^{(m)}$, and the assumptions of the theorem imply that each such evaluation can be done in time $2^{O(m)} = 2^{O(\ell)}$. We thus see that the entire computation of $G$ can be done in time $2^{O(\ell)}$, as required. ■

## Bibliographic Notes

The results here are due to Nisan and Wigderson [2], whose paper is very readable. A good starting point for subsequent developments in this area is [1, Chapter 20].

## References


[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*, Cambridge University Press, 2009.

[2] N. Nisan and A. Wigderson. Hardness vs. Randomness. *J. Computer & System Sciences* 49(2):149–167, 1994. Preliminary version in FOCS '88.

**Notes on Complexity Theory** Last updated: December, 2011

# Lecture 27

*Jonathan Katz*

## 1 Space-Bounded Derandomization

We now discuss derandomization of space-bounded algorithms. Here non-trivial results can be shown *without making any unproven assumptions*, in contrast to what is currently known for derandomizing time-bounded algorithms. We show first that $\mathcal{BPL} \subseteq \text{SPACE}(\log^2 n)$ and then improve the analysis and show that[1] $\mathcal{BPL} \subseteq \text{TIMESPC}(\mathsf{poly}(n), \log^2 n) \subseteq \mathcal{SC}$. (Note: we already know

$$\mathcal{RL} \subseteq \mathsf{NL} \subseteq \text{SPACE}(\log^2 n)$$

but this does not by itself imply $\mathcal{BPL} \subseteq \text{SPACE}(\log^2 n)$.)

With regard to the first result, we actually prove something more general:

**Theorem 1** *Any randomized algorithm (with two-sided error) that uses space $S = \Omega(\log n)$ and $R$ random bits can be converted to one that uses space $\mathcal{O}(S \log R)$ and $\mathcal{O}(S \log R)$ random bits.*

Since any algorithm using space $S$ uses time at most $2^S$ (by our convention regarding probabilistic machines) and hence at most this many random bits, the following is an immediate corollary:

**Corollary 2** *For $S = \Omega(\log n)$ it holds that $\text{BPSPACE}(S) \subseteq \text{SPACE}(S^2)$.*

**Proof** Let $L \in \text{BPSPACE}(S)$. Theorem 1 shows that $L$ can be decided by a probabilistic machine with two-sided error using $\mathcal{O}(S^2)$ space and $\mathcal{O}(S^2)$ random bits. Enumerating over all random bits and taking majority, we obtain a deterministic algorithm that uses $\mathcal{O}(S^2)$ space. ■

## 2 $\mathcal{BPL} \subseteq \text{SPACE}(\log^2 n)$

We now prove Theorem 1. Let $M$ be a probabilistic machine running in space $S$ (and time $2^S$), using $R \leq 2^S$ random bits, and deciding a language $L$ with two-sided error. (Note that $S, R$ are functions of the input length $n$, and the theorem requires $S = \Omega(\log n)$.) We will assume without loss of generality that $M$ always uses exactly $R$ random bits on all inputs; recall also that $M$ has *read-once* access to its random bits. Fixing an input $x$ and letting $\ell$ be some parameter, we will view the computation of $M_x$ as a random walk on a multi-graph in the following way: our graph will have $R/\ell + 1$ layers, with each layer containing $N \stackrel{\text{def}}{=} 2^{\mathcal{O}(S)}$ nodes that correspond to possible configurations of $M_x$. There is an edge from node $a$ (in some layer $i$) to node $b$ (in some layer $i+1$) labeled by the string $r \in \{0,1\}^\ell$ iff $M_x$ moves from configuration $a$ to configuration $b$ after reading $r$ as its next $\ell$ random bits. Computation of $M(x)$ is then equivalent to a random walk of length $R/\ell$ on this graph, beginning from the node corresponding to the initial configuration of $M_x$ (in

[1] $\mathcal{SC}$ captures computation that *simultaneously* uses polynomial time and polylogarithmic space.

layer 0). If $x \in L$ then the probability that this random walk ends up in the accepting state is at least 2/3, while if $x \notin L$ then the probability that this random walk ends up in the accepting state is at most 1/3.

It will be convenient to represent this process using an $N \times N$ *transition matrix* $Q_x$, where the entry in column $i$, row $j$ is the probability that $M_x$ moves from configuration $i$ to configuration $j$ after reading $\ell$ random bits. Vectors of length $N$ whose entries are non-negative and sum to 1 correspond to probability distributions over the configurations of $M_x$ in the natural way. If we let $\mathbf{s}$ denote the probability distribution that places probability 1 on the initial configuration of $M_x$ (and 0 elsewhere), then $Q_x^{R/\ell} \cdot \mathbf{s}$ corresponds to the probability distribution over the final configuration of $M_x$; thus, if we let $i$ denote the accepting configuration of $M_x$:

$$\begin{aligned} x \in L &\Rightarrow \left(Q_x^{R/\ell} \cdot \mathbf{s}\right)_i \geq 3/4 \\ x \notin L &\Rightarrow \left(Q_x^{R/\ell} \cdot \mathbf{s}\right)_i \leq 1/4. \end{aligned}$$

The statistical difference between two vectors/probability distributions $\mathbf{s}, \mathbf{s}'$ is

$$\mathsf{SD}(\mathbf{s}, \mathbf{s}') \stackrel{\text{def}}{=} \frac{1}{2} \cdot \left\|\mathbf{s} - \mathbf{s}'\right\|_1 = \frac{1}{2} \cdot \sum_i |\mathbf{s}_i - \mathbf{s}'_i|.$$

If $Q, Q'$ are two transition matrices — meaning that all entries are non-negative, and the entries in each column sum to 1 — then we abuse notation and define

$$\mathsf{SD}(Q, Q') \stackrel{\text{def}}{=} \max_{\mathbf{s}}\{\mathsf{SD}(Q\mathbf{s}, Q'\mathbf{s})\},$$

where the maximum is taken over all $\mathbf{s}$ that correspond to probability distributions. Note that if $Q, Q'$ are $N \times N$ transition matrices and $\max_{i,j}\{|Q_{i,j} - Q'_{i,j}|\} \leq \varepsilon$, then $\mathsf{SD}(Q, Q') \leq N\varepsilon/2$.

## 2.1 A Useful Lemma

The pseudorandom generator we construct will use a family $H$ of pairwise-independent functions as a building block. It is easy to construct such a family $H$ whose functions map $\ell$-bit strings to $\ell$-bit strings and such that (1) $|H| = 2^{2\ell}$ (and so choosing a random member of $H$ is equivalent to choosing a random $2\ell$-bit string) and (2) functions in $H$ can be evaluated in $\mathcal{O}(\ell)$ space.

For $S \subseteq \{0,1\}^\ell$, define $\rho(S) \stackrel{\text{def}}{=} |S|/2^\ell$. We define a useful property and then show that a function chosen from a pairwise-independent family satisfies the property with high probability.

**Definition 1** *Let $A, B \subseteq \{0,1\}^\ell$, $h : \{0,1\}^\ell \to \{0,1\}^\ell$, and $\varepsilon > 0$. We say $h$ is $(\varepsilon, A, B)$-**good** if:*

$$\left| \Pr_{x \in \{0,1\}^\ell}\left[x \in A \bigwedge h(x) \in B\right] - \Pr_{x,y \in \{0,1\}^\ell}\left[x \in A \bigwedge y \in B\right] \right| \leq \varepsilon.$$

**Lemma 3** *Let $A, B \subseteq \{0,1\}^\ell$, $H$ be a family of pairwise-independent functions, and $\varepsilon > 0$. Then:*

$$\Pr_{h \in H}[h \text{ is not } (\varepsilon, A, B)\text{-good}] \leq \frac{\rho(A)\rho(B)}{2^\ell \varepsilon^2}.$$

**Proof** The proof is fairly straightforward. We want to bound the fraction of functions in $H$ for which $\left|\Pr_{x\in\{0,1\}^\ell}[x \in A \bigwedge h(x) \in B] - \rho(A)\cdot\rho(B)\right| > \varepsilon$ or, equivalently,

$$\left|\Pr_{x\in A}[h(x) \in B] - \rho(B)\right| > \varepsilon/\rho(A).$$

For fixed $x$, let $\delta_{h(x)\in B}$ be an indicator random variable (over random choice of $h \in H$) that is 1 iff $h(x) \in B$. Then we are interested in the fraction of $h \in H$ for which

$$\left|\sum_{x\in A}\delta_{h(x)\in B} - |A|\cdot\rho(B)\right| > \varepsilon\cdot|A|/\rho(A).$$

Using Chebyshev's inequality and pairwise independence of $H$, we obtain

$$\Pr_{h\in H}\left[\left|\sum_{x\in A}\delta_{h(x)\in B} - |A|\cdot\rho(B)\right| > \varepsilon\cdot|A|/\rho(A)\right] \leq \frac{|A|\cdot\rho(B)\cdot\rho(A)^2}{\varepsilon^2|A|^2} = \frac{\rho(A)\rho(B)}{2^\ell\varepsilon^2}.$$

■

## 2.2 The Pseudorandom Generator and Its Analysis

### 2.2.1 The Basic Step

We first show how to reduce the number of random bits by roughly half. Let $H$ denote a pairwise-independent family of functions, and fix an input $x$. Let $Q$ denote the transition matrix corresponding to transitions in $M_x$ after reading $\ell$ random bits; that is, the $(j,i)$th entry of $Q$ is the probability that $M_x$, starting in configuration $i$, moves to configuration $j$ after reading $\ell$ random bits. So $Q^2$ is a transition matrix denoting the probability that $M_x$, starting in configuration $i$, moves to configuration $j$ after reading $2\ell$ random bits. Fixing $h \in H$, let $Q_h$ be a transition matrix where the $(j,i)$th entry in $Q_h$ is the probability that $M_x$, starting in configuration $i$, moves to configuration $j$ after reading the $2\ell$ "random bits" $r\|h(r)$ (where $r \in \{0,1\}^\ell$ is chosen uniformly at random). Put differently, $Q^2$ corresponds to taking two uniform and independent steps of a random walk, whereas $Q_h$ corresponds to taking two steps of a random walk where the first step (given by $r$) is random and the second step (namely, $h(r)$) is a deterministic function of the first. We now show that these two transition matrices are "very close". Specifically:

**Definition 2** *Let $Q, Q_h, \ell$ be as defined above, and $\varepsilon \geq 0$. We say $h \in H$ is $\varepsilon$-**good for** $Q$ if*

$$\mathsf{SD}(Q_h, Q^2) \leq \varepsilon/2\,.$$

**Lemma 4** *Let $H$ be a pairwise-independent function family, and let $Q$ be an $N \times N$ transition matrix where transitions correspond to reading $\ell$ random bits. For any $\varepsilon > 0$ we have:*

$$\Pr_{h\in H}[h \text{ is not } \varepsilon\text{-good for } Q] \leq \frac{N^6}{\varepsilon^2 2^\ell}\,.$$

**Proof** For $i, j \in [N]$ (corresponding to configurations in $M_x$), define

$$B_{i,j} \stackrel{\text{def}}{=} \{r \in \{0,1\}^\ell \mid r \text{ defines a transition from } i \text{ to } j\}.$$

For any fixed $i, j, k$, we know from Lemma 3 that the probability that $h$ is not $(\varepsilon/N^2, B_{i,j}, B_{j,k})$-good is at most $N^4\rho(B_{i,j})\rho(B_{j,k})/\varepsilon^2 2^\ell \leq N^4\rho(B_{i,j})/\varepsilon^2 2^\ell$. Applying a union bound over all $N^3$ triples $i, j, k \in [N]$, and noting that for any $i$ we have $\sum_j \rho(B_{i,j}) = 1$, we have that $h$ is $(\varepsilon/N^2, B_{i,j}, B_{j,k})$-good for *all* $i, j, k$ except with probability at most $N^6/\varepsilon^2 2^\ell$.

We show that whenever $h$ is $(\varepsilon/N^2, B_{i,j}, B_{j,k})$-good for all $i, j, k$, then $h$ is $\varepsilon$-good for $Q$. Consider the $(k, i)$th entry in $Q_h$; this is given by: $\sum_{j\in[N]} \Pr_{r\in\{0,1\}^\ell}[r \in B_{i,j} \wedge h(r) \in B_{j,k}]$. On the other hand, the $(k, i)$th entry in $Q^2$ is: $\sum_{j\in[N]} \rho(B_{i,j}) \cdot \rho(B_{j,k})$. Since $h$ is $(\varepsilon/N^2, B_{i,j}, B_{j,k})$-good for every $i, j, k$, the absolute value of their difference is

$$\begin{aligned}
&\left| \sum_{j\in[N]} \Big( \Pr[r \in B_{i,j} \wedge h(r) \in B_{j,k}] - \rho(B_{i,j}) \cdot \rho(B_{j,k}) \Big) \right| \\
&\quad \leq \sum_{j\in[N]} \Big| \Pr[r \in B_{i,j} \wedge h(r) \in B_{j,k}] - \rho(B_{i,j}) \cdot \rho(B_{j,k}) \Big| \\
&\quad \leq \sum_{j\in[N]} \varepsilon/N^2 \;=\; \varepsilon/N.
\end{aligned}$$

It follows that $\mathsf{SD}(Q_h, Q^2) \leq \varepsilon/2$ as desired. ■

The lemma above gives us a pseudorandom generator that reduces the required randomness by (roughly) half. Specifically, define a pseudorandom generator $G_1 : \{0,1\}^{2\ell+R/2} \to \{0,1\}^R$ via:

$$G_1(h; r_1, \ldots, r_{R/2\ell}) = r_1 \,\|\, h(r_1) \,\|\, \cdots \,\|\, r_{R/2\ell} \,\|\, h(r_{R/2\ell}), \tag{1}$$

where $h \in H$ (so $|h| = 2\ell$) and $r_i \in \{0,1\}^\ell$. Assume $h$ is $\varepsilon$-good for $Q$. Running $M_x$ using the output of $G_1(h; \cdots)$ as the "random tape" generates the probability distribution

$$\overbrace{Q_h \cdots Q_h}^{R/2\ell} \cdot \mathbf{s}$$

for the final configuration, where $\mathbf{s}$ denotes the initial configuration of $M_x$ (i.e., $\mathbf{s}$ is the probability distribution that places probability 1 on the initial configuration of $M_x$, and 0 elsewhere). Running $M_x$ on a truly random tape generates the probability distribution

$$\overbrace{Q^2 \cdots Q^2}^{R/2\ell} \cdot \mathbf{s}$$

for the final configuration. Since $\mathsf{SD}(Q_h, Q^2) \leq \varepsilon/2$, we have

$$\mathsf{SD}(\overbrace{Q_h \cdots Q_h}^{R/2\ell} \cdot \mathbf{s},\ \overbrace{Q^2 \cdots Q^2}^{R/2\ell} \cdot \mathbf{s}) \leq \frac{R}{2\ell} \cdot \frac{\varepsilon}{2}.$$

This means that the behavior of $M_x$ when run using the output of the pseudorandom generator is very close to the behavior of $M_x$ when run using a truly random tape: in particular, if $x \notin L$ then $M_x$ in the former case accepts with probability at most

$$\Pr[\text{accepts} \wedge h \text{ is } \varepsilon\text{-good for } Q] + \Pr[h \text{ is not } \varepsilon\text{-good for } Q] \;\leq\; \left(1/4 + R\varepsilon/4\ell\right) + N^6/\varepsilon^2 2^\ell;$$

similarly, if $x \in L$ then $M_x$ in the former case accepts with probability at least $3/4 - R\varepsilon/4\ell - N^6/\varepsilon^2 2^\ell$. Summarizing (and slightly generalizing):

**Corollary 5** *Let $H$ be a pairwise-independent function family, let $Q$ be an $N \times N$ transition matrix where transitions correspond to reading $\ell$ random bits, let $k > 0$ be an integer, and let $\varepsilon > 0$. Then except with probability at most $N^6/\varepsilon^2 2^\ell$ over choice of $h \in H$ we have:*

$$\mathsf{SD}\left( \overbrace{Q_h \cdots Q_h}^{k}, \overbrace{Q^2 \cdots Q^2}^{k} \right) \leq k\varepsilon/2.$$

### 2.2.2 Recursing

Fixing $h_1 \in H$, note that $Q_{h_1}$ is a transition matrix and so we can apply Corollary 5 to *it* as well. Moreover, if $Q$ uses $R$ random bits then $Q_{h_1}$ uses $R/2$ random bits (treating $h_1$ as fixed). Continuing in this way for $I \stackrel{\text{def}}{=} \mathcal{O}(\log(R/\ell))$ iterations, we obtain a transition matrix $Q_{h_1,\ldots,h_I}$. Say *all $h_i$ are $\varepsilon$-good* if $h_1$ is $\varepsilon$-good for $Q$, and for each $i > 1$ it holds that $h_i$ is $\varepsilon$-good for $Q_{h_1,\ldots,h_{i-1}}$. By Corollary 5 we have:

- All $h_i$ are $\varepsilon$-good except with probability at most $N^6 I/\varepsilon^2 2^\ell$.
- If all $h_i$ are $\varepsilon$-good then

$$\mathsf{SD}(Q_{h_1,\ldots,h_I}, \overbrace{Q^2 \cdots Q^2}^{R/2\ell}) \;\leq\; \frac{\varepsilon}{2} \cdot \sum_{i=1}^{I} \frac{R}{2^i \ell} \;=\; \mathcal{O}(\varepsilon R/\ell).$$

Equivalently, we obtain a pseudorandom generator

$$G_I(h_1, \ldots, h_I; r) \stackrel{\text{def}}{=} G_{I-1}(h_1, \ldots, h_{I-1}; r) \,\|\, G_{I-1}(h_1, \ldots, h_{I-1}; h_I(r)),$$

where $G_1$ is as in Equation (1).

### 2.2.3 Putting it All Together

We now easily obtain the desired derandomization. Recall $N = 2^{\mathcal{O}(s)}$. Set $\varepsilon = 2^{-S}/10$, and set $\ell = \Theta(S)$ so that $\frac{N^6 S}{\varepsilon^2 2^\ell} \leq 1/20$. Then the number of random bits used (as input to $G_I$ from the previous section) is $\mathcal{O}(\ell \cdot \log(R/\ell) + \ell) = \mathcal{O}(S \log R)$ and the space used is bounded by that as well (using the fact that each $h \in H$ can be evaluated using space $\mathcal{O}(\ell) = \mathcal{O}(S)$). All $h_i$ are good except with probability at most $N^6 \log(R/\ell)/\varepsilon^2 2^\ell \leq N^6 S/\varepsilon^2 2^\ell \leq 1/20$; assuming all $h_i$ are good, the statistical difference between an execution of the original algorithm and the algorithm run with a pseudorandom tape is bounded by $2^{-S}/20 \cdot R \leq 1/20$. Theorem 1 follows easily.

### 2.3 $\mathcal{BPL} \subseteq \mathcal{SC}$

A deterministic algorithm using space $\mathcal{O}(\log^2 n)$ might potentially run for $2^{\mathcal{O}(\log^2 n)}$ steps; in fact, as described, the algorithm from the proof of Corollary 2 uses this much time. For the particular pseudorandom generator we have described, however, it is possible to do better. The key observation is that instead of just choosing the $h_1, \ldots, h_I$ at random and simply hoping that they are all $\varepsilon$-good, we will instead deterministically search for $h_1, \ldots, h_I$ which *are* each $\varepsilon$-good. This can be done in polynomial time (when $S = \mathcal{O}(\log n)$) because: (1) for a given transition matrix $Q_{h_1,\ldots,h_{i-1}}$ and candidate $h_i$, it is possible to determine in polynomial time and polylogarithmic space whether $h_i$ is $\varepsilon$-good for $Q_{h_1,\ldots,h_{i-1}}$ (this relies on the fact that the number of configurations $N$ is polynomial in $n$); (2) there are only a polynomial number of possibilities for each $h_i$ (since $\ell = \Theta(S) = \mathcal{O}(\log n)$).

Once we have found the good $\{h_i\}$, we then cycle through all possible choices of the seed $r \in \{0,1\}^\ell$ and take majority (as before). Since there are a polynomial number of possible seeds (again using the fact that $\ell = \Theta(S) = \mathcal{O}(\log n)$), the algorithm as a whole runs in polynomial time.

(For completeness, we discuss the case of general $S = \Omega(\log n)$ assuming $R = 2^S$. Checking whether a particular $h_i$ is $\varepsilon$-good requires time $2^{\mathcal{O}(S)}$. There are $2^{\mathcal{O}(S)}$ functions to search through at each stage, and $\mathcal{O}(S)$ stages altogether. Finally, once we obtain the good $\{h_i\}$ we must then enumerate through $2^{\mathcal{O}(S)}$ seeds. The end result is that $\text{BPSPACE}(S) \subseteq \text{TIMESPC}(2^{\mathcal{O}(S)}, S^2)$.)

## 3 Applications to Error Reduction

Interestingly, the same pseudorandom generator we have constructed can also be used for efficient error reduction. Before discussing this application, we briefly discuss error reduction in general. (For simplicity, we focus here on the case of error reduction for randomized algorithms with *one-sided* error; all results described here can be generalized for the case of two-sided error.)

For concreteness, say we have an algorithm $A$ for some language $L$ such that

$$\begin{aligned} x \in L &\Rightarrow \Pr[A(x) = 1] \geq 1/2 \\ x \notin L &\Rightarrow \Pr[A(x) = 1] = 0. \end{aligned}$$

Say $A$ uses $\ell$ random bits. (The time/space complexity of $A$ is not relevant to this discussion.) A naïve approach to error reduction would be to run $A$ on a given input $k$ times using independent random tapes $r_1, \ldots, r_k$, outputting 1 iff any of these runs returns 1. This uses $k \cdot \ell$ random bits, requires running $A$ for $k$ times, and achieves error $2^{-k}$.

A different approach (due to Chor-Goldreich) is to let the $\{r_i\}$ be *pairwise independent* rather than completely independent. That is, choose random $h \in H$ and set $r_i = h(i) \in \{0,1\}^\ell$; then run $A$ for $k$ times using the random coins $r_1, \ldots, r_k$. This uses $\mathcal{O}(\ell)$ random bits (the only randomness is the choice of $h$) and $k$ executions of $A$ as before, but only achieves error $\mathcal{O}(1/k)$. (The proof follows directly from Chebyshev's inequality.)

A better approach uses (a small modification of) the pseudorandom generator from the previous section. Define $G_1(h_1; r) = r \,\|\, h_1(r)$ and, inductively,

$$G_I(h_1, \ldots, h_I; r) \stackrel{\text{def}}{=} G_{I-1}(h_1, \ldots, h_{I-1}; r) \,\|\, G_{I-1}(h_1, \ldots, h_{I-1}; h_I(r)).$$

(The difference from before is that now the output length of $G_I$ grows; specifically, the output length of $G_I$ is $\left(\{0,1\}^\ell\right)^{2^I}$.) Our algorithm will now be to run $A$ on each of the $k \stackrel{\text{def}}{=} 2^I$ strings

output by $G_I$; we output 1, as before, iff $A$ outputs 1 in one of those executions. Now we use $\mathcal{O}(\ell \cdot \log k)$ random bits (and $k$ executions of $A$); the error is given by the following theorem.

**Theorem 6** *If $x \in L$, the probability that $A$ always outputs 0 when run on the $k = 2^I$ random strings output by $G_I$ is at most $2^{-k} + (\log k + 2) \cdot 2^{-\ell/3}$.*

**Proof** Setting $\varepsilon = 2^{-\ell/3}$, Lemma 3 shows that for any $A, B \subseteq \{0,1\}^\ell$ we have

$$\Pr_{h \in H}[h \text{ is not } (\varepsilon, A, B)\text{-good}] \leq \varepsilon\,.$$

Thus, all $h_i$ are $(\varepsilon, A, B)$-good (for any $A, B$) except with probability at most $\varepsilon \cdot \log k$.

Assuming all $h_i$ are $(\varepsilon, A, B)$-good, we prove by induction on $I$ that the probability (over choice of $r \in \{0,1\}^\ell$) that $A$ always outputs 0 when run on the output of $G_I(h_1, \ldots, h_I; r)$ is at most $2^{-2^I} + 2\varepsilon$. For $I = 0$ this is immediate. We prove it holds for $I$, assuming it holds for $I - 1$.

Let

$$A = B = \{r \mid A \text{ always outputs 0 when run on the output of } G_{I-1}(h_1, \ldots, h_{I-1}; r)\}.$$

By our inductive step, $\rho(A) = \rho(B) \leq 2^{-2^{I-1}} + 2\varepsilon$. Furthermore, the probability that $A$ always outputs 0 when run on the output of $G_I(h_1, \ldots, h_I; r)$ is exactly the probability that $r \in A$ and $h_I(r) \in B$. Since $h_I$ is $(\varepsilon, A, B)$-good we have

$$\begin{aligned}
\Pr_{r \in \{0,1\}^\ell}[r \in A \bigwedge h_I(r) \in B] &\leq \Pr_{r, r' \in \{0,1\}^\ell}[r \in A \bigwedge r' \in B] + \varepsilon \\
&\leq \left(2^{-2^{I-1}} + 2\varepsilon\right)^2 + \varepsilon \\
&\leq 2^{-2^I} + 2\varepsilon.
\end{aligned}$$

This completes the proof. ■

## Bibliographic Notes

The results of Section 2 are due to [3, 4], both of which are very readable. See also [1, Lecture 16] for a slightly different presentation. Section 3 is adapted from the Luby-Wigderson survey on pairwise independence [2].

## References


[1] O. Goldreich. Introduction to Complexity Theory (July 31, 1999).

[2] M. Luby and A. Wigderson. *Pairwise Independence and Derandomization.* Foundations and Trends in Theoretical Computer Science. Now Publishers Inc., 2006. (Available freely online.)

[3] N. Nisan. Pseudorandom Generators for Space-Bounded Computation. *STOC* '90.

[4] N. Nisan. *RL* $\subseteq$ *SC*. *Computational Complexity* 4: 1–11, 1994. (Preliminary version in *STOC* '92.)

**Notes on Complexity Theory** Last updated: December, 2011

# Lecture 28

*Jonathan Katz*

## 1 Circuit Lower Bounds

Recall that one motivation for studying non-uniform computation is the hope that it might be *easier* to prove lower bounds in that setting. (This is somewhat paradoxical, as non-uniform algorithms are more powerful than uniform algorithms; nevertheless, since circuits are more "combinatorial" in nature than uniform algorithms, there may still be justification for such hope.) The ultimate goal here would be to prove that $\mathcal{NP} \not\subset \mathcal{P}_{/\mathsf{poly}}$, which would imply $\mathcal{P} \neq \mathcal{NP}$. Unfortunately, after over two decades of attempts we are unable to prove anything close to this. Here, we show one example of a lower bound that we *have* been able to prove; we then discuss one "barrier" that partly explains why we have been unable to prove stronger bounds.

## 2 Parity Cannot be Solved by $\mathsf{AC}^0$ Circuits

Recall that $\mathsf{AC}^0$ is the set of languages/problems decided by *constant-depth*, polynomial-size circuits (with gates of unbounded fan-in). We consider the basis consisting of AND, OR, and NOT gates, though we do not count NOT gates when measuring the depth or size of the circuit. The parity function is given by $f(x_1 \cdots x_n) = x_1 \oplus \cdots \oplus x_n$. In this lecture we give the "polynomial proof" that parity cannot be computed in $\mathsf{AC}^0$. We will actually prove something stronger:

**Theorem 1** *For sufficiently large $n$, any depth-$d$ circuit that computes parity on $n$-bit inputs must have at least $\frac{1}{50} \cdot 2^{0.5 \cdot n^{1/2d}}$ gates.*

Thus, for any fixed depth-bound $d$, any circuit family computing parity grows as $2^{n^\varepsilon}$ for some $\varepsilon > 0$.

**Proof** Fix a circuit $C$ of depth $d$ that computes parity on inputs of length $n$. Let $x_1, \ldots, x_n$ denote the inputs to the circuit. We will assume that $C$ has only OR gates and NOT gates; this assumption is without loss of generality since we may convert any AND gate to a combination of OR and NOT gates using De Morgan's laws (by setting $\bigwedge_i a_i = \neg \bigvee_i (\neg a_i)$) without affecting the size or depth of the circuit.

Let $\mathbb{F}_3 = \{-1, 0, 1\}$ be the field of size 3. Say a polynomial $p \in \mathbb{F}_3[x_1, \ldots, x_n]$ is *proper* if $p(x_1, \ldots, x_n) \in \{0, 1\}$ whenever $x_1, \ldots, x_n \in \{0, 1\}$. Note that any proper polynomial can be viewed as a boolean function in the natural way.

The proof hinges on two lemmas: we first show that any circuit in $\mathsf{AC}^0$ can be approximated fairly well by a (proper) low-degree polynomial, and then show that parity cannot be approximated well by any low-degree polynomial.

**Lemma 2** *For every integer $t > 0$, there exists a (proper) polynomial of total degree $(2t)^d$ that differs with $C$ on at most $\mathsf{size}(C) \cdot 2^{n-t}$ inputs.*

**Proof** We will associate a proper polynomial with each wire of the circuit, and then bound the error introduced. Begin at the input wires and associate the monomial $x_i$ to the input $x_i$. Now consider the output wire of some gate $g$, all of whose input wires have already been associated with polynomials. Then:

- If $g$ is a NOT gate, and its input wire is associated with the polynomial $p$, then associate its output wire with $1 - p$. Note that this does not increase the degree of the polynomial.
- If $g$ is an OR gate with $k$ input wires associated with the polynomials $p_1, \ldots, p_k$, then do the following:

  Choose sets $S_1, \ldots, S_t \subseteq [k]$ (see below for how these sets are chosen), and define $q_i = \left(\sum_{j \in S_i} p_j\right)^2$ for $i = 1, \ldots, t$. Then set $p = 1 - \prod_{i=1}^{t}(1 - q_i)$.

  (Note that $p$ is just the OR of the $q_i$.) If the maximum (total) degree of the $\{p_i\}$ is $b$, then the (total) degree of polynomial $p$ is at most $2tb$. Note further that $p$ is proper.

For a given wire with associated polynomial $p$, an *error* is an input $x_1, \ldots, x_n$ on which the value of the wire and the value of $p$ differ. We now bound the fraction of errors in the polynomial $p^*$ associated with the output wire of the circuit. No errors are introduced at input wires or at NOT gates. Looking at any OR gate with $k$ input wires associated with the polynomials $p_1, \ldots, p_k$, we claim that there is *some* choice of subsets $S_1, \ldots, S_t \subseteq [k]$ that will not introduce too many errors. On any input where all the $p_i$'s evaluate to 0, the resulting polynomial $p$ will also evaluate to 0. Consider any input where at least one of the $p_i$'s evaluates to 1, and let $S_1, \ldots, S_t$ be random subsets of $[k]$. With probability at least half over choice of subset $S_j$, polynomial $q_j$ will evaluate to 1. If *any* of the $q_j$ evaluate to 1 then so does $p$. So the probability that $p$ does not evaluate to 1 is at most $2^{-t}$. By an averaging argument, this implies the *existence* of some collection of subsets that introduces errors on at most a $2^{-t}$ fraction of the inputs at this gate.

Taking a union bound, we conclude that $p^*$ is a polynomial of degree at most $(2t)^d$ having at most $\mathsf{size}(C) \cdot 2^{n-t}$ errors with respect to $C$. ■

Setting $t = n^{1/2d}/2$ we get a polynomial of degree at most $\sqrt{n}$ that differs from $C$ on at most $\mathsf{size}(C) \cdot 2^{n-t}$ inputs.

**Lemma 3** *Let $p \in \mathbb{F}_3[x_1, \ldots, x_n]$ be a proper polynomial of degree at most $\sqrt{n}$. Then for sufficiently large $n$ the polynomial $p$ differs from the parity function on at least $2^n/50$ inputs.*

**Proof** Consider the "translated" parity function $\mathsf{parity}' : \{-1, 1\}^n \to \{-1, 1\}$ defined as

$$\mathsf{parity}'(x_1, \ldots, x_n) = \prod_i x_i.$$

Since $\mathsf{parity}'(x_1, \ldots, x_n) = \mathsf{parity}(x_1 - 1, \ldots, x_n - 1) + 1$, there exists a polynomial $p'$ of degree at most $\sqrt{n}$ that agrees with $\mathsf{parity}'$ on the same number of inputs for which $p$ agrees with $\mathsf{parity}$.

Let $S \subseteq \{-1, 1\}^n$ be the set of inputs on which $p'$ and $\mathsf{parity}'$ agree, and let $\mathcal{F}$ denote the set of *all* functions from $S$ to $\mathbb{F}_3$. Note that $|\mathcal{F}| = 3^{|S|}$. Now, for every function $f \in \mathcal{F}$ we can associate a polynomial $p_f \in \mathbb{F}_3[x_1, \ldots, x_n]$ that agrees with $f$ for all $x \in S$: just set

$$p_f(x_1, \ldots, x_n) = -\sum_{y \in S} f(y) \cdot \prod_{i=1}^{n} (y_i \, x_i + 1) \, .$$

Although $p_f$, as constructed, has degree 1 in each input variable, the total degree of $p_f$ may be as large as $n$. We claim that, in fact, we can associate with each $f$ a polynomial $\hat{p}_f$ whose degree is at most $n/2 + \sqrt{n}$. To see this, fix $f$ and $p_f$ and look at some monomial $\pm \prod_{i \in T} x_i$ appearing in $p_f$ where $|T| > n/2 + \sqrt{n}$. For any $x \in S \subseteq \{-1, 1\}^n$ we have

$$\begin{aligned} \pm \prod_{i \in T} x_i &= \pm \prod_{i=1}^{n} x_i \cdot \prod_{i \notin T} x_i \\ &= \pm p'(x) \cdot \prod_{i \notin T} x_i \,. \end{aligned}$$

Since $p'$ has degree at most $\sqrt{n}$, we see that we can re-write $p_f$ as a polynomial $\hat{p}_f$ that agrees with $p_f$ on $S$ and has degree at most $n/2 + \sqrt{n}$.

The number of monomials whose total degree is at most $n/2 + \sqrt{n}$ is $\sum_{i=0}^{n/2+\sqrt{n}} \binom{n}{i}$, which is less than $49 \cdot 2^n/50$ for large enough $n$. So the total number of polynomials whose degree is at most $n/2 + \sqrt{n}$ is upper bounded by $3^{49 \cdot 2^n/50}$. Given that $|\mathcal{F}| = 3^{|S|}$, this means we must have $|S| \leq 49 \cdot 2^n/50$ as claimed. ■

To complete the proof, we just combine the two lemmas. The first lemma gives a polynomial $p$ of degree at most $\sqrt{n}$ that differs from parity on at most $\mathsf{size}(C) \cdot 2^{n - n^{1/2d}/2}$ inputs. The second lemma tells us that, for large enough $n$, we must have $\mathsf{size}(C) \cdot 2^{n - n^{1/2d}/2} \geq 2^n/50$. We conclude that $\mathsf{size}(C) \geq \frac{1}{50} \cdot 2^{0.5 \cdot n^{1/2d}}$, completing the proof. ■

# 3 Limits of Proving Lower Bounds: Natural Proofs

The lower bound proved in the previous section is, to a rough approximation, about the best we are able to show. Why is that? We explore one reason here.

We formalize a notion of proving lower bounds using "natural" proof techniques. We then show that natural proofs *cannot* be used to prove strong lower bounds assuming some (widely believed) cryptographic assumptions hold.[1]

## 3.1 Defining Natural Proofs

Let $\mathcal{C}_0, \mathcal{C}_1$ denote classes of predicates. ($\mathcal{C}_0, \mathcal{C}_1$ need to satisfy certain technical restrictions, but these conditions are not very restrictive.) For example, $\mathcal{C}_0$ or $\mathcal{C}_1$ could be $\mathsf{AC}^0$ or $\mathcal{P}_{/\mathsf{poly}}$, or even something more specific like "depth-5 circuits having at most $n^2$ gates". Say we want to show that some function $f : \{0,1\}^* \to \{0,1\}$ is not in $\mathcal{C}_0$. One way to prove this would be to define a "hardness predicate" $P$ on functions such that, for $n$ large enough, $P(f_n) = 1$ (where $f_n$ denotes the restriction of $f$ to $n$-bit inputs) but for any $g \in \mathcal{C}_0$ and $n$ large enough, $P(g_n) = 0$. (Here we view $P$ as taking the *truth table* of $f_n$ as input. Note that the truth table of $f_n$ has size $2^n$, and we measure complexity of $P$ in terms of that length.) A proof of this form is called a $\mathcal{C}_1$-*natural against* $\mathcal{C}_0$ if $P$ satisfies two conditions: (1) $P$ is in $\mathcal{C}_1$ (this is called the *constructiveness* condition),

[1] This is an interesting situation, as it means that one way to make progress on proving lower bounds (i.e., to prove that some function is "hard") would be to show better upper bounds (namely, to show that certain cryptographic problems are actually "easy".)

and (2) for $n$ sufficiently large, a random predicate $g_n : \{0,1\}^n \to \{0,1\}$ satisfies $P(g_n) = 1$ with probability at least $1/n$ (this is called the *largeness* condition).

**Why are these properties "natural"?** There are two types of answers here. The first is just to observe that (almost?) all known lower bounds do, in fact, satisfy these properties. For example, in the lower bound for parity that we showed in the previous section, the predicate $P$ corresponds informally to

$$P(f_n) = 0 \Leftrightarrow f_n \text{ can be "well approximated" by a "low-degree" polynomial.}$$

A random function cannot be approximated by a low-degree polynomial, with high probability. Constructiveness is more difficult to show, and actually requires us to consider a slightly different predicate $P'$ (for which the largeness condition still holds); we refer to [4] for details. All-in-all, it is possible to show that the parity lower bound is $\mathsf{NC}^2$-natural against $\mathsf{AC}^0$. (Interestingly, previous lower bounds for parity were $\mathsf{AC}^0$-natural against $\mathsf{AC}^0$.)

The second argument in favor of our definition of "natural" is to appeal to intuition. This argument is somewhat harder to make (which gives hope that we can circumvent the barrier imposed by natural proofs, and find new proof techniques for showing lower bounds). Constructiveness is motivated by the fact that, as part of any lower-bound proof of the above form, we must show that $P(f_n) = 1$; it seems that any "constructive" proof of this fact should involve some "efficient" computation involving the truth table of $f_n$. The largeness condition is perhaps more natural. For starters, random functions are typically hardest. Moreover, every proof that a given function $f_n : \{0,1\}^n \to \{0,1\}$ cannot be computed by a circuit of $S$ also proves that at least half the functions on $n$ bits cannot be computed by a circuit of size $\approx S/2$. If not, then write

$$f_n(x) = (f_n \oplus g_n) \oplus g_n$$

for a random function $g_n : \{0,1\}^n \to \{0,1\}$; with probability strictly greater than 0, both $g_n$ and $f_n \oplus g_n$ can be computed by a circuit of size $\approx S/2$ (note that both $g_n$ and $f_n \oplus g_n$ are random functions), but then $f_n$ can be computed by a circuit of size $S$.

### 3.2 Ruling Out Natural Proofs of Lower Bounds

For simplicity, let us fix some $n$ instead of working asymptotically. Then we can view $\mathcal{C}_0, \mathcal{C}_1$ as classes of functions on $n$-bit and $2^n$-bit inputs, respectively. Say we have a function $f : \{0,1\}^n \to \{0,1\}$ and want to prove that $f$ is not in $\mathcal{C}_0$ using a $\mathcal{C}_1$-natural proof. This means that we want to define a predicate $P \in \mathcal{C}_1$ such that $P(f) = 1$ but $P(g) = 0$ for all $g \in \mathcal{C}_0$.

Let $\mathcal{F}$ denote a keyed function mapping inputs of length $n$ to boolean outputs, and having a key of length $m$, i.e., $\mathcal{F} = \{F_k : \{0,1\}^n \to \{0,1\}\}_{k \in \{0,1\}^m}$. We say that $\mathcal{F}$ *can be computed in* $\mathcal{C}_0$ if, for any key $k \in \{0,1\}^m$, the function $F_k : \{0,1\}^n \to \{0,1\}$ is in $\mathcal{C}_0$. (Note in particular that the function $F(k,x) \stackrel{\text{def}}{=} F_k(x)$ may have complexity "higher" than $\mathcal{C}_0$.) Informally, $\mathcal{F}$ is a *pseudorandom function* (PRF) if it is hard to distinguish whether a given function $h : \{0,1\}^n \to \{0,1\}$ is equal to $F_k$ for a random key $k \in \{0,1\}^m$, or whether $h$ is a random boolean function on $n$-bit inputs. Our notion of distinguishing will be very strong: rather than considering distinguishers that make oracle queries to $h$, we simply provide the distinguisher with the entire truth table of $h$.[2] Formally,

[2]For those used to thinking about polynomial-time distinguishers of cryptographic PRFs this may seem strange. If it helps, one can think of $m$ as the security parameter and consider, for example, the case where $n = O(\log^2 m)$. Then we are just requiring security against slightly super-polynomial distinguishers running in time $\mathsf{poly}(2^n) = m^{O(\log m)}$.

then, we say that $\mathcal{F}$ is *pseudorandom against* $\mathcal{C}_1$ if for every distinguisher $D \in \mathcal{C}_1$ we have

$$\left| \Pr_{k \leftarrow \{0,1\}^m}[D(F_k) = 1] - \Pr_{h \leftarrow \mathsf{Func}_n}[D(h) = 1] \right| < 1/n,$$

where $\mathsf{Func}_n$ denote the space of all predicates on $n$-bit inputs.

We can now state the main result:

**Theorem 4** *Assume there exists an $\mathcal{F}$ that can be computed in $\mathcal{C}_0$ and is pseudorandom against $\mathcal{C}_1$. Then there is no $\mathcal{C}_1$-natural proof against $\mathcal{C}_0$.*

**Proof** A $\mathcal{C}_1$ natural proof against $\mathcal{C}_0$ would imply the existence of a predicate $P \in \mathcal{C}_1$ such that $P(g) = 0$ for all $g \in \mathcal{C}_0$, while $\Pr_{h \leftarrow \mathsf{Func}_n}[P(h) = 1] \geq 1/n$. But then $P$ acts as a distinguisher for $\mathcal{F}$, using the fact that $F_k \in \mathcal{C}_0$ for every key $k$. ■

Under suitable cryptographic hardness assumptions (e.g., the assumption that the discrete logarithm problem has hardness $2^{n^\varepsilon}$ for some $\varepsilon > 0$, or an analogous assumption for hardness of subset sum) there are pseudorandom functions that can be computed in $\mathsf{NC}^1$ (or even[3] $\mathsf{TC}^0$) and are pseudorandom against $\mathcal{P}$. Thus, if these cryptographic conjectures are true, there are no $\mathcal{P}$-natural proofs even against weak classes like $\mathsf{NC}^1$ or $\mathsf{TC}^0$. This helps explain why current circuit lower bounds are "stuck" at $\mathsf{AC}^0$ and[4] $\mathsf{ACC}^0$.

## Bibliographic Notes

This proof given here that parity is not in $\mathsf{AC}^0$ is due to Razborov [3] and Smolensky [5] (who proves a more general result); earlier proofs (using a different approach) were given by Furst, Saxe, and Sipser, by Yao, and by Håstad. Good surveys of circuit lower bounds include the (old) article by Boppana and Sipser [1] and the (new) book by Jukna [2]. Natural proofs were introduced by Razborov and Rudich [4].

## References


[1] R. Boppana and M. Sipser. The Complexity of Finite Functions. In *Handbook of Theoretical Computer Science, vol. A: Algorithms and Complexity*, J. van Leeuwen, ed., MIT Press, 1990.

[2] S. Jukna. *Boolean Function Complexity: Advances and Frontiers*. Springer, 2012.

[3] A. Razborov. Lower Bounds on the Size of Bounded Depth Networks Over a Complete Basis with Logical Addition. *Matematicheskie Zametki* 41:598–607, 1987.

[4] A. Razborov and S. Rudich. Natural Proofs. *J. Computer and System Sciences* 55(1): 24–35, 1997.

[5] R. Smolensky. Algebraic Methods in the Theory of Lower Bounds for Boolean Circuit Complexity. STOC 1987.


[3] $\mathsf{TC}^0$ is the class of predicates that can be computed by constant-depth circuits of polynomial size, over a basis of unbounded fan-in threshold gates. Note that $\mathsf{AC}^0 \subseteq \mathsf{TC}^0 \subseteq \mathsf{NC}^1$.

[4] $\mathsf{ACC}^0$ is the class of predicates that can be computed by constant-depth circuits of polynomial size, over a basis of unbounded fan-in AND, OR, and mod $m$ gates (for any fixed constant $m$). Note that $\mathsf{AC}^0 \subseteq \mathsf{ACC}^0 \subseteq \mathsf{TC}^0$.