CS 810: Complexity Theory	1/22/2007
Lecture 1: Introduction	
Instructor: Dieter van Melkebeek	Scribe: Jeff Kinne

Today we begin a review of basic Complexity Theory material that is typically covered in an undergraduate theory of computing course. In this first lecture, we introduce the course, define the model of computation we use to formalize our intuitive notion of a computer, and explore issues that arise concerning the computational model.

1 Course Overview

This course provides a graduate-level introduction to computational complexity theory, the study of the power and limitations of efficient computation.

In the first part of the course we focus on the standard setting, in which one tries to realize a given mapping of inputs to outputs in a time- and space-efficient way. We develop models of computation that represent the various capabilities of digital computing devices, including parallelism, randomness, quantum effects, and non-uniformity. We also introduce models based on the notions of nondeterminism, alternation, and counting, which precisely capture the power needed to efficiently compute important types of mappings. The meat of this part of the course consists of intricate relationships between these models, as well as some separation results.

In the second part we study other computational processes that arise in diverse areas of computer science, each with their own relevant efficiency measures. Specific topics include:

- proof complexity, interactive proofs, and probabilistically checkable proofs – motivated by verification,
- pseudorandomness and zero-knowledge – motivated by cryptography and security,
- computational learning theory – motivated by artificial intelligence,
- communication complexity – motivated by distributed computing,
- query complexity – motivated by databases.

All of these topics have grown into substantial research areas in their own right. We will cover the main concepts and key results from each one.

2 The Standard Setting

2.1 Machine Model

The deterministic Turing machine is the model of computation we use to capture our intuitive notion of a computer. A Turing machine is depicted in Figure 1. The finite control has a finite number states that it can be in at any time, and can read from and/or write to the various memory tapes. Based on the current state and the contents of the tapes, the finite control changes its state and alters the contents of the tapes.

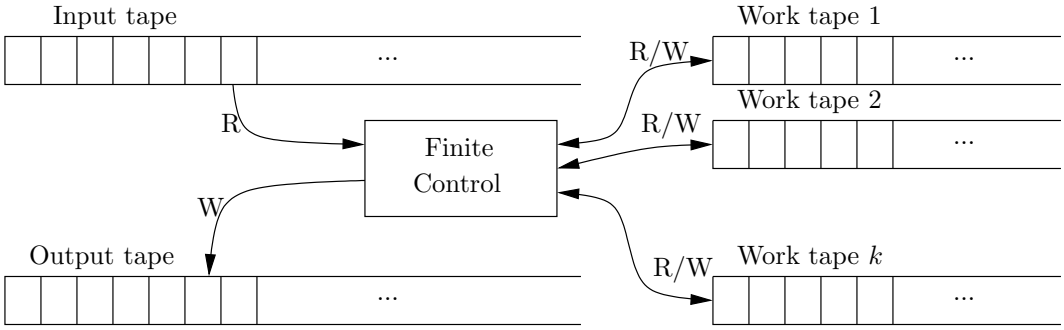


Figure 1: An illustration of a deterministic Turing machine. The finite control has read-only access to the input tape, write-only access to a one-way output tape, and read-write access to a constant k many work tapes.

Definition 1 (sequential access Turing machine)**.** *A* sequential access deterministic Turing machine *M is defined by a tuple: $M = (Q, \Sigma, \Gamma, q_0, q_{halt}, \delta)$, where Q is a finite set of possible states of the finite control, Σ is a finite input and output alphabet, Γ is a finite work-tape alphabet, q_{start} is the start state, q_{halt} is the halt state, and δ is the finite control's transition function.*

The transition function has the form

$$\delta : Q \setminus \{q_{halt}\} \times \Sigma \times \Gamma^k \to Q \times \{\Sigma \cup \{\epsilon\}\} \times \Gamma^k \times \{L, R\} \times \{L, R\}^k.$$

The input to δ represents the current state, the current symbol being scanned on the input tape, and the current symbol being scanned on each work tape. The output represents the next state of the finite control, a symbol to write to the output tape (possibly empty), symbols to write to each work tape, and which direction to move the head on the input and work tapes.

We typically use the binary alphabet for all tapes, so that $\Sigma = \Gamma = \{0, 1\}$.

We think of the use of a Turing machine as consisting of three steps. First, the machine is initialized as follows: place input on input tape with tape head on first symbol of input, work tapes are empty with their tape heads on the left-most cell of the tape, the output tape is empty, and the finite control is set to q_{start}. Second, the machine is allowed to run one step at a time by repeatedly applying the transition function δ. Third, if the machine ever halts by entering q_{halt}, the computation is finished and we can read the output from the output tape. We use $M(x)$ to denote the output of M on input x when this computation halts.

At first sight, the definition of a Turing machine may seem to be too restrictive to correspond to our intuitive notion of computing. However, the Turing machine has been shown to be just as powerful and roughly as efficient as traditional computers (we discuss this more later in this lecture).

The model of Turing machine we use in this course is a modification of the definition given above. The definition above is a bit too restrictive as it does not allow indirect memory addressing, which real computers rely on.

Definition 2 (random access Turing machine)**.** *A* random access Turing machine *is a Turing machine that functions as in Definition 1 with regards to its output and sequential access work tapes. The input tape and any fixed number of work tapes may be* random access tapes *rather than sequential access tapes. Each random access tape has an associated sequential access* index work

tape. *The tape head of a random access tape is moved by a special jump operation that moves its tape head to the location specified by its index tape. The tape head position of a random access tape is not altered by the transition function, but the contents of the memory cell are read and written by the transition function.*

Notice that each tape is either a sequential tape or a random access tape but not both. We think of the sequential access tapes as performing operations that are usually performed in registers on modern computers such as arithmetic, while the random access tapes are used for storage. We choose the random access Turing machine as our basic model of computation for this course as it more closely models modern computers than sequential access Turing machines.

2.2 Computing a Relation

The main setting in the first part of the course is the Turing machine's ability to compute various relations.

Definition 3. *Given a* relation $R \subset \Sigma^* \times \Sigma^*$*, we say that a Turing machine* M computes R *if the following holds. For all inputs* $x \in \Sigma^*$*,* M *on input* x *halts and outputs a* $y \in \Sigma^*$ *such that* $(x, y) \in R$ *if such a* y *exists and indicates "no" if no such* y *exists.*

Indicating no can be accomplished by some encoding scheme on the output or by having two different halt states.

As an example relation, consider the shortest path problem. For this relation, we wish to compute a shortest path between two vertices in a graph. Here, the input is a description of the graph and source and destination vertices; the output is the description of a path through the graph. Notice that a function is a special case of a relation, where for a given x there is at most one y such that $(x, y) \in R$. Factoring is an example, where the input is an integer n, and the output is the unique prime factorization of n listing the prime factors from smallest to largest.

We are often interested in a particular kind of relation, called a decision problem.

Definition 4 (decision problem)**.** *A* decision problem *is a relation where the output is always either 1 or 0. In this case, an output of 1 indicates that the input has some property, and 0 indicates the input does not have the property. We refer to the set of inputs with corresponding output 1 as a* language.

As an example, consider the problem of determining if a string is a palindrome. We define the language PALINDROMES as the set of strings that read the same front to back as they do from back to front. The corresponding relation assigns 1 to each palindrome and 0 to each non-palindrome.

It is often the case that the complexity of computing a relation is captured by the complexity of computing a related decision problem. For example, we can turn the factoring problem into a decision problem by trying to compute the i^{th} bit of the output. That is, on input $\langle n, i \rangle$, we try to compute the i^{th} bit of the description of the prime factorization of n.

3 Time and Space Complexity

The Turing machine was originally defined and used in the theory of computability (also known as recursion theory). In this setting, the goal is to determine which relations are computable. For example, a famous early result is that the Halting Problem is not computable by any Turing machine.

It may seem surprising at first that there are uncomputable relations. A closer inspection makes this fact obvious: there are only countably many Turing machines, while there are uncountably many different relations.

In contrast to the setting of computability, complexity theory is concerned with the efficiency with which a relation is computable. To this end, we consider the amount of resources a Turing machine uses during its computation. The two standard resources to consider are time and space.

Definition 5. *Let M be a Turing machine and x an input to M. Then*

$t_M(x) =$ *the number of steps until M halts on input x,*
$t_M(n) = \max(t_M(x) | x \in \Sigma^n)$,
$s_M(x) =$ *the sum over all work tapes of the maximum cell touched until M halts,*
$s_M(n) = \max(s_M(x) | x \in \Sigma^n)$.

$t_M(x)$ corresponds to the time used by M on input x, while $s_M(x)$ corresponds to the amount of memory used. A possible alternative to the definition we have given for $s_M(x)$ would be to only count the number of work-tape cells that are touched. Our definition counts all cells that are left of some work-tape cell that is touched, and hence counts even unused cells that are left of some used cell. Our definition is more natural from the perspective that if you want to run the algorithm on a computer, you'll need one with that much memory. If an algorithm uses 10K space with the alternative definition, it may not run on a machine with 10K of RAM, at least not without modification; it does with the our definition. Also notice that the configuration of a machine (the positions of its tapes, contents of its work tapes, and its internal state) can be described using $O(s_M + \log(|x|))$ space, while this is not true with the alternative definition. For most purposes, the choice in definition does not effect the power or efficiency of the model.

We are often interested in the worst-case complexity of a Turing machine, and hence have defined the worst-case measures $t_M(n)$ and $s_M(n)$. These correspond to the worst performance of M on any input of length n.

Notice in the definition of $s_M(x)$ that we do not count the input tape or output tape memory cells that are used. This choice in definition is the reason we distinguished between the different types of tapes in the first place, and allows us to consider Turing machines that compute non-trivial relations using sub-linear space. A definition including input tape usage in $s_M(x)$ would preclude non-trivial sub-linear space algorithms as then the entire input could not even be read. We will consider algorithms that use at least logarithmic space as this is the amount required to index into the input. For time usage, linear is the smallest we consider as we must at least read the entire input.

3.1 The Goal of Complexity Theory

The main goal of complexity theory is to characterize the amount of time and space required to compute a given relation. In other words, find a machine M so that either $t_M(n)$ or $s_M(n)$ is minimal over all machines that solve the relation. However, a number of issues arise when asking this question.

- *Hard wiring.* The solution to any finite subset of the possible inputs can be hard-wired into the transition function of a Turing machine. By using a lookup table, any finite subset of possible inputs can be solved very fast with low space usage.

  Because of this issue, we focus on asymptotic run-time and space usage.

- *Constant factor speedups.* Consider a Turing machine that uses a certain amount of space using the binary alphabet. By changing to the alphabet $\{0,1\}^{10}$, the machine can store the information of 10 of its original tape cells into 1 of the new tape cells. The finite control of the machine can be modified to work properly in the new setting, thus reducing the space usage of the machine roughly by a factor of 10. The space usage can be decreased by any constant factor by increasing the size of the tape alphabet to a suitably large constant. Notice that if we had used the alternative definition of space usage that only counts memory cells that are touched during the computation, then this argument would only work for random access machines for algorithms that have a high amount of memory locality.

  A similar argument can be made for time usage. The argument is a bit more complicated because the finite control must read both the current cell and immediately adjacent cells to properly mimic the original Turing machine. It may seem that random access operations would destroy any possible time speedup. An argument to get around this potential problem makes use of the fact that indexing operations will still be sped up and an algorithm making many random access operations will also spend a large amount of time indexing those operations.

  Because of these constant factor speedups, we ignore constant factors in running time and space usage.

- *Incomparable behavior.* Even modulo the above issues, a fundamental problem remains in attempting to determine the optimal time and space usage to compute a relation. It may be the case that there are two machines M_1 and M_2 so that on some input lengths t_{M_1} is smaller while on other input lengths t_{M_2} is smaller. In this case, a third machine can be created to simulate both machines and thus achieve near optimal run-time over all inputs. However, because of the hard-wiring issue already discussed, given a finite set of inputs, there is always a Turing Machine that decides these instances correctly very fast and with small space usage. Each of these machines is nearly optimal on some different set of inputs, and it is not in general possible to create a machine combining the optimal performance from each of these infinitely many machines. Thus, in general, there is no single single Turing machine with minimal run-time or space-usage on all input lengths.

  Because of this, we instead look at the dual of the problem we originally stated: rather than trying to find minimal $t_M(n)$ for a given relation, we instead try to determine which relations can be computed given a certain time or space bound.

Definition 6. *For a given $t : \mathbb{N} \to \mathbb{N}$ and $s : \mathbb{N} \to \mathbb{N}$, we define:*
DTIME($t(n)$) = { *Decision problems solvable in $O(t(n))$ time by some random access Turing machine* },
DSPACE($s(n)$) = { *Decision problems solvable in $O(s(n))$ space by some random access Turing machine* }.

The main goal of complexity theory can then be restated as follows: given a time or space bound $t(n)$, which relations can be computed on a Turing machine in this amount of time or space? Given this new goal, a number of issues still remain.

- *Model dependence.* We would like our results to be independent of the particular choices we have made in defining the Turing machine. Recall the *Church-Turing thesis* - that any

relation computable on a physically realizable computing device can also be computed on a Turing machine. This belief underscores the use of the Turing machine in computability theory as the computing device that is studied. Note that the Church-Turing thesis is not violated by any of the models of computation we consider in this course.

In the setting of complexity theory, we consider the *Strong Church-Turing thesis* - that any relation computable on a physically realizable computing device can be computed by our model of a Turing machine with a polynomial overhead in time and a constant overhead in space. Namely, if some machine uses $t(n)$ time to compute the relation, there is a Turing machine using $\text{poly}(n, t(n))$ time; and if there is a machine using $s(n)$ space, there is a Turing machine using $O(\log n + s(n))$ space. Notice that the Strong Church-Turing thesis is a much bolder statement than the Church-Turing thesis. In particular, it is widely believed to be violated by quantum machines (although it is debatable if these are physically realizable), and may even be violated by randomized machines (although this is believed not to be the case).

Consider the language PALINDROMES and a single tape sequential Turing machine. It can be shown that the trivial algorithm of scanning back and forth across the input string is the best possible, with a running time of $\Theta(n^2)$. A similar result shows that on a multi-tape sequential access Turing machine, the product of the space and time usage to solve palindromes is $\Omega(n^2)$. However, PALINDROMES can be decided in simultaneous quasi-linear time and logarithmic space on random access machines.

Notice that this result implies that a single-tape Turing machine must incur roughly a quadratic overhead in time if it attempts to simulate our model of a random access multi-tape Turing machine. That is, different physically realizable models of computation often differ by a polynomial factor in the time to solve a relation and in a constant factor in the space needed. Hence the Strong Church-Turing thesis stated above certainly can not be strengthened.

Also notice that the quadratic lower bound for PALINDROMES on sequential access Turing machines is model dependent: it is true on certain models but not on others. We have chosen the model of the random access Turing machine to avoid such model dependent results. Each of the results we present in this course will be model independent.

- *Input representation.* The input representation is clear for the PALINDROMES language. However, for many relations there is some choice that can be made. Consider the shortest path problem. The input to the problem includes a graph, which must be represented somehow on the input tape. Standard methods of representing a graph include an adjacency matrix and an adjacency list. Note there may be a linear factor difference in the size of these. This effects the running time and space usage because these are functions of the size of the input. If an input is made to be artificially large (say, by padding with unnecessary 0's) then the running time and space usage are artificially decreased as functions of the size of the input.

  For this reason, we always assume a "reasonable" encoding of the input. We leave this notion qualitative, although it can be quantified. As long as we choose a reasonable encoding of the input, the running time and space usage do not depend too much on the input encoding.

Because of the above issues, we define complexity classes that are robust with respect to both the model and the input representation - namely the complexity class remains the same regardless of the particular model or reasonable input encoding used.

Definition 7. *The following are definitions of standard complexity classes.*

$$\begin{array}{lcl} \text{P} & = & \cup_{c>0}\,\text{DTIME}(n^c), \\ \text{EXP} & = & \cup_{c>0}\,\text{DTIME}(2^{n^c}), \\ \text{E} & = & \cup_{c>0}\,\text{DTIME}(2^{c\cdot n}), \\ \text{L} & = & \text{DSPACE}(\log n), \\ \text{PSPACE} & = & \cup_{c>0}\,\text{DSPACE}(n^c). \end{array}$$

Notice that each of these is robust with respect to the model under the strong Church-Turing thesis. Each of the classes other than E also is robust with respect to input representation.

It may seem that restricting the space usage to logarithmic in the case of the complexity class L is too great. For graph problems, logarithmic space is only enough to remember a constant number of vertices within the graph. However, it turns out that many interesting problems can be solved within L. For example, a recent result showed that the problem of determining connectivity in an undirected graph can be achieved using logarithmic space on deterministic machines.

4 Universality

A Turing machine U is called a *universal Turing machine* if it is able to simulate all other Turing machines: for all Turing Machines M and inputs x, $U(\langle M, x\rangle) = M(x)$. A key property that leads to a number of results is that there exist universal Turing machines that incur a small overhead in time and space.

Theorem 1. *There are universal Turing machines U_{DTIME} and U_{DSPACE} such that for all Turing machines M and inputs x:*

$$\begin{array}{l} t_{U_{\text{DTIME}}}(\langle M, x\rangle) = O(|M| \cdot t_M(x) \cdot \log(t_M(x))), \\ s_{U_{\text{DSPACE}}}(\langle M, x\rangle) = O(\log(|M|) \cdot s_M(x)), \end{array}$$

where $|M|$ denotes the length of the description of M.

In fact, we will show that a single machine can be used for both U_{DTIME} and U_{DSPACE}.

Proof sketch: The main difficulty in designing U_{DTIME} and U_{DSPACE} is that each must have a fixed number of work tapes while simulating machines with any number of work tapes.

First consider U_{DSPACE}. Suppose M has k work tapes. We would like to keep the contents of these on a single tape for U_{DSPACE}. We call this tape the storage tape. This is done by first storing the first cell from each of M's k tapes in the first k cells of U_{DSPACE}'s storage tape, then storing the second cell from each of M's k tapes in the next k cells, and so on. Recall that in a single step, M reads the contents of each work tape, and the locations of the tape head can be different for each of these. So, U_{DSPACE} must know where each of M's tape heads is. U_{DSPACE} stores this information on an additional work tape - we call this tape the lookup tape. U_{DSPACE} also must have an index tape in order to perform tape head jumps. We leave it as an exercise to verify that U_{DSPACE} can simulate M using these three tapes (storage, lookup, and index), and that the simulation is as efficient as claimed.

Consider the time required to run the above simulation. For each step of M's execution, U_{DSPACE} must read the contents of the sequential access work tapes and must remember the

contents of the current cell on each of the random access work tapes. As the sequential access tapes access locations that are at most $t_M(x)$, the address for each sequential access tape head takes $O(\log t_M(x))$ space. We leave it as an exercise to verify that each transition function step of M takes $O(|M| \log t_M(x))$ steps for U_{DSPACE} to simulate. Together with the fact that U_{DSPACE} performs tape head jump operations in constant time just as M does, we conclude that $U_{\text{DTIME}} = U_{\text{DSPACE}}$ is as efficient with respect to time as claimed. Notice that this analysis takes into account that U_{DTIME} has a random access input tape although M may have a sequential access input tape. For simulating M that have a sequential access input tape, U_{DTIME} incurs a log factor overhead in time to simulate sequential access to its input tape. □

A key component of the analysis of the time efficiency in the above proof is that each work tape is only either sequential access or random access. If a single work tape were allowed to be both sequential and random access, the analysis would fail (the counterexample in this case is a machine M that jumps to location 2^r and then performs r local operations near this address - then M runs in $O(r)$ time while the simulation would take time $\Theta(|M| \cdot r^2)$). Even in this situation, a universal machine U_{DTIME} with similar overhead in time can be constructed by ensuring a simulation of M where U_{DTIME} only ever accesses tape cells with small addresses - namely $O(\log t_M(x))$. This is achieved by keeping track of random access tape operations as (tape cell address, tape cell contents) pairs and storing these in an efficient data structure such as a balanced binary tree.

The same trick of keeping track of random access tape operations in a data structure can be used to convert any machine M using time t into another machine M' that computes the same relation and uses space $O(t)$. Because of how we have defined space complexity, notice that this transformation is not trivial - a machine M can use space that is exponential in t by writing down a large address and accessing that location on its random access tape.

The existence of a universal machine is a key component to a number of important results - in particular hierarchy theorems and completeness results. A hierarchy theorem states that a Turing machine with slightly more resources (either time or space) can compute relations that cannot be computed with slightly less resources. A problem is complete for a complexity class if it is both within the class and at least as difficult as all other problems within the class. The universal machines given above can be used to prove time and space hierarchy theorems and the existence of complete problems.

We formalize these notions and give the appropriate proofs in the next lecture. We will also show the following relationships between complexity classes:

$$\text{L} \subseteq \text{P} \subseteq \text{PSPACE} \subseteq \text{EXP}.$$

It is open whether any of these containments is proper.

CS 810: Complexity Theory 1/24/2007

Lecture 2: Determinism and Nondeterminism

Instructor: Dieter van Melkebeek Scribe: Matt Elder

In this lecture, we discuss two applications of efficient, deterministic, universal Turing machines: deterministic time and spaces hierarchy theorems, and completeness results. We also introduce the nondeterministic Turing machine, our model for nondeterministic computation.

1 Hierarchy Results

A *hierarchy result* is a theorem that imposes a strict-subset relation between two complexity classes; that is, for some two classes of problems $\mathcal{C}$ and $\mathcal{C}'$, a hierarchy result might yield that $\mathcal{C}' \subsetneq \mathcal{C}$. Thus, class $\mathcal{C}$ is computationally more powerful than class $\mathcal{C}'$.

For example, we can show that if the function t' is sufficiently smaller than the function t, then $\text{DTIME}(t') \subsetneq \text{DTIME}(t)$. We prove this using the technique of *diagonalization*, which originated in Cantor's proof that $[0,1]$ is uncountable.

Theorem 1 (Cantor)**.** *The interval* $[0,1]$ *is uncountably infinite.*

Proof. We prove this theorem by contradiction. Given any enumeration of numbers from $[0,1]$, we show that it cannot contain all numbers from $[0,1]$ by constructing a number r' from $[0,1]$ that it cannot contain.

Assume that $[0,1]$ is countable. If the interval $[0,1]$ is countable, then its individual elements can be enumerated. Suppose we somehow enumerate all numbers in $[0,1]$. Let r_i be the infinite binary representation of the i^{th} number in this enumeration, and let $b_{i,j}$ be the j^{th} bit in r_i.

	b_1	b_2	b_3	$\cdots$
r_1	**0**	1	1	$\cdots$
r_2	1	**0**	0	$\cdots$
r_3	1	0	**1**	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$
r'	1	1	0	$\cdots$

Figure 1: r' is constructed to complement the diagonal elements of each r_i, used to prove Theorem 1.

Now, we consider the diagonal elements $b_{i,i}$, the bold elements in Figure 1. To build r' from these diagonal elements, set bit i of r' to the complement of $b_{i,i}$. Since the i^{th} bit of r' differs from the i^{th} bit of r_i, we know that $r' \neq r_i$ for all i. Thus, r' represents a number x that is in $[0,1]$ but not already enumerated as some r_i. Actually, this isn't *quite* true, because every rational number has two infinite binary representations: one terminating with $000\ldots$ and one terminating with $111\ldots$. We have several ways to avoid this issue. For example, we could let all representations be in base 4, and set digit i of r' to be 1 if $b_{i,i}$ equals 2 or 3, and set it to 2 if $b_{i,i}$ equals 0 or 1. We could instead ensure that r' does not end in an infinite sequence of zeroes or ones, a construction that is more complicated.

Provided we have taken care of the issue described above, the construction of r' contradicts the fact that our enumeration contains every element of $[0, 1]$, so our initial assumption is false, and $[0, 1]$ is not countable. □

The proof of the deterministic time and space hierarchies emulate Cantor's diagonalization proof. The theorem requires a condition on the time bounds known as time-constructibility which we describe as it appears in the proof.

Theorem 2. *If t and t' are functions from $\mathbb{N}$ to $\mathbb{N}$, t is time-constructible, and $t(n) = \omega(t'(n) \log t'(n))$, then* $\text{DTIME}(t') \subsetneq \text{DTIME}(t)$.

Proof. We will use a universal Turing machine to construct a contrary machine M. We build M so that for each Turing machine M_i running in time t', M's output differs from the output of M_i on some input x_i. In this way, our contrary machine M will accept a language that no machine can accept in time t'. We must also ensure M runs in time t.

Let μ be a function mapping inputs to descriptions of Turing machines with the following properties: i) μ is computable in linear time, and ii) each machine M' appears infinitely often as an image of μ. We leave it to the reader to verify that such a μ can be constructed from any computable enumeration of deterministic Turing machines. Let x_i denote the i^{th} possible input, and $\mu(x_i) = \langle M_i \rangle$. Then the code for M is as follows.

(1) Read input x_i.
(2) Compute $\mu(x_i) = \langle M_i \rangle$.
(3) Pass $\langle M_i, x_i \rangle$ to a universal Turing machine, and run the simulation as long as the total time used by M is at most $t(|x_i|)$.
(4) If the universal Turing machine halts and rejects, accept x_i. Otherwise reject x_i.

In the above simulation, M must be able to keep track of its time usage: M wishes to use at most $t(|x_i|)$ time in total. This is the place where we use the time-constructibility of t. We define this notion after the proof. The property we require is that M can in time $O(t(|x_i|))$ write down $t(|x_i|)$ many zeroes on a work tape. We use these zeroes as a clock by erasing one for each step of execution and halting if all of them are ever erased.

As t is time-constructible, the discussion above shows that M can keep track of its time usage to ensure the entire execution is $O(t(|x|))$. Because $t(n) = \omega(t'(n) \log t'(n))$ and the efficiency of the universal Turing machine proved in the previous lecture, M has enough time to perform the simulation for machine M' running in t' time for all but finitely many inputs. Because each Turing machine M' appears infinitely often as an image of μ, there is an input for which M has enough time to complete the simulation and complement the behavior M' so long as M' runs in t' time.

Thus, $L(M) \notin \text{DTIME}(t')$ whereas $L(M) \in DTIME(t)$, and $\text{DTIME}(t') \subsetneq \text{DTIME}(t)$. □

The condition required of the time bound in the theorem is stated formally in the following definition. It can be shown that all of the functions we are used to dealing with (polynomials, exponentials, logarithms, etc.) that are at least linear are time-constructible and those that are at least logarithmic are space-constructible.

Definition 1. *A function $t : \mathbb{N} \to \mathbb{N}$ is* time-constructible *if the function that maps the string 0^n to the string $0^{t(n)}$ can be computed in time $O(t(n))$. Similarly, a function $s : \mathbb{N} \to \mathbb{N}$ is* space-constructible *if the function that maps 0^n to $0^{s(n)}$ can be computed in space $O(s(n))$.*

We can use an identical proof to derive a hierarchy theorem for the amount of space used by deterministic machines, giving the following theorem. Due to the fact that the overhead in space for a universal Turing machine is smaller than the overhead for time, the space hierarchy is tighter than the time hierarchy.

Theorem 3. *If $s(n) = \omega(s'(n))$ and s is space-constructable, then* $\text{DSPACE}(s') \subsetneq \text{DSPACE}(s)$.

Notice that the separation in space bounds above is as tight as we could hope for, as we have already discussed that constant-multiple differences between two functions do not change their computational power.

As a corollary of these theorems, we know that $\text{P} \subsetneq E \subsetneq EXP$, and that $L \subsetneq PSPACE$.

2 Reductions and Completeness

Problem reducibility is central to the study of complexity. Reductions allow us to determine the relative complexity of problems without knowing the absolute complexity of those problems. If A and B are two problems, then $A \leq B$ denotes that A reduces to B. So, A can be efficiently solved if B can be efficiently solved. This implies that the complexity of A is no greater than the complexity of B (modulo the complexity of the reduction itself), so the notation $A \leq B$ is sensible in this context.

We consider two types of reductions, mapping and oracle reductions. Mapping reductions are more restrictive.

Definition 2. *A* mapping reduction*, also called a many-one reduction or a Karp reduction, is a function that maps instances of one problem to instances of another, preserving their outcome.*

More specifically, if A and B are decision problems, then $A \leq_m B$ if there exists some function $f : \Sigma_A^* \to \Sigma_B^*$ such that, for all problem instances $x \in \Sigma_A^*$, $x \in A \iff f(x) \in B$. That is, $f(x)$ is in the language of B iff x is in the language of A.

To capture the intended meaning of a reduction, we would like the function f to be efficient. The efficiency requirement varies by context - we describe the two most common settings in a moment.

Definition 3. *An* oracle reduction*, also called a Turing reduction or Cook reduction, is an algorithm to solve one problem given a solver to a second problem as an instantaneous subroutine.*

*A B-*oracle *is a hypothetical machine that can solve any instance of problem B and return its answer in one step. An* Oracle Turing Machine *(OTM) is a Turing Machine with a special oracle tape, and a specific query state. When the machine reaches the query state, it invokes its oracle on the current content of the oracle tape. In this state, the oracle's input is erased, the oracle's output is placed on the oracle tape, and the oracle tape head is placed on the leftmost position.*

For a given OTM M, M^B is that machine with a B oracle. We say that A oracle-reduces *to B, denoted $A \leq_o B$, if there exists some efficient OTM M such that M^B solves A.*

Given a mapping reduction f from A to B, we can give an oracle-reduction of A to B in the following OTM: "Given input x, compute $f(x)$ and write it to the oracle tape. Then, query the oracle, and return its output." Since any mapping reduction may be thus reduced to an oracle reduction, oracle reductions are at least as powerful as mapping reductions.

We denote time or space constraints on the efficiency of a reduction via superscripts on the $\leq$ symbol. Polynomial-time reducibility is denoted $\leq^{\mathrm{P}}$, and log-space reducibility is denoted $\leq^{\log}$. log-space oracle reductions have a slightly complicated specification, which may differ by author and context. For our purposes, we impose the restriction that that we may read and write on the oracle tape only from left to right, so that the oracle tape cannot be used as procedural memory. We do not count the size of the oracle tape against our space efficiency.

Proposition 1. *Let $\tau \in \{m, o\}$ and let $r \in \{\mathrm{P}, \log\}$. The following are true:*

- *Reducibility is transitive. If $A \leq_\tau^r B$ and $B \leq_\tau^r C$, then $A \leq_\tau^r C$.*
- *If $A \leq_\tau^{\mathrm{P}} B$ and $B \in \mathrm{P}$, then $A \in \mathrm{P}$.*
- *If $A \leq_\tau^{\log} B$ and $B \in \mathrm{L}$, then $B \in \mathrm{L}$.*

Definition 4. *Given a reduction relation $\leq$ and complexity class $\mathcal{C}$, B is* hard *for $\mathcal{C}$ under $\leq$ if, for every problem A in $\mathcal{C}$, $A \leq B$. We say that B is* complete *for $\mathcal{C}$ under $\leq$ if B is hard for $\mathcal{C}$ under $\leq$ and $B \in \mathcal{C}$.*

The choice of reduction depends on context. For example, it is known that $\mathrm{L} \subseteq \mathrm{P}$, but is P equal to L? In this case, we use log-space reductions for the following reason: given a problem B that is complete for P under $\leq_\tau^{\log}$, $\mathrm{P} \subseteq \mathrm{L}$ if and only if B is in L.

Given the connection between complete problems and complexity class collapses, it is useful to have a variety of complete problems to make use of. As a first start, we can construct complete languages out of efficient universal Turing machines.

Proposition 2. *Let K_D be the language of tuples $\langle M, x, 0^t \rangle$ such that M halts and accepts input x in no more than t steps. K_D is complete for* P *under $\leq_m^{\log}$.*

Proof. The language K_D is in P because U_{DTIME} runs with polynomial time overhead.

Suppose A is a language in P. We demonstrate a function f demonstrating that $A \leq_m^{\log} K_D$. If we know the Turing machine N that computes A in polynomial time, then we can define the mapping f that takes x to $\langle N, x, 0^{|x|^c} \rangle$ such that $f(x) \in K_D$ iff $x \in A$. We can hard-code N into f, and it is possible to output both x and $0^{|x|^c}$ in log space with c hard-coded into f. Thus $f(x)$ is a log-space mapping reduction from A to K_D. □

3 Nondeterminism

Note well: the nondeterministic model of computation is not intuitive, and for good reason. *Nondeterminism is not a physically realizable model of computation.* The utility of the model lies not in modeling actual computation, but in exactly capturing the complexity of an important class of problems.

3.1 Model

The standard model for nondeterministic computation is the nondeterministic Turing machine (NTM). An NTM has a definition very similar to a normal, deterministic Turing machine, except that the transition function δ in a TM may be a relation in an NTM. Thus, for any combination of internal state and read-head state, δ may provide multiple next-state instructions.

The language of an NTM M, $L(M)$ is the set of strings x such that there exists a valid computation of M on input x that halts and accepts. That is, we say that M accepts x if some possible computation path of M accepts x.

There are several fruitful interpretations of NTMs. We can view an NTM as a machine allowed to make guesses, and these guesses are controlled by some force that wants the machine to accept. For a particular input x, if any permissible series of guesses will lead the machine to its accepting state then the machine will make those guesses and accept x. If there is no such series of guesses, then the machine cannot accept x and is forced to reject x.

We can also think of an NTM as a massively parallel computer. Every time it makes a choice, it spawns a copy of itself for every possible branch of that choice, and all of these copies continue processing. If any child machine accepts an input, then the machine as a whole accepts that input.

3.2 Time and Space

NTM efficiency is stated in terms of worst-case behavior. So, "M runs in time t" means that *every* branch of the execution of M halts within t steps. Similarly, "M runs in space s" means that every branch of the execution of M uses only s cells.

The nondeterministic complexity classes NTIME(t) and NSPACE(s) are precisely analogous to their deterministic counterparts DTIME(t) and DSPACE(s): a problem is in NTIME(t) if there exists an NTM M that solves that problem and runs in time $O(t(n))$ for inputs of size n. A problem is in NSPACE(s) if there exists an NTM M that solves that problem and runs in space $O(s(n))$ for inputs of size n.

The following complexity classes are analogous to their deterministic counterparts:

- NP $= \cup_{c>0}$ NTIME(n^c)
- NE $= \cup_{c>0}$ NTIME(2^{cn})
- NEXP $= \cup_{c>0}$ NTIME(2^{n^c})
- NL $=$ NSPACE($\log(n)$)
- NPSPACE $= \cup_{c>0}$ NSPACE(n^c)

We won't much talk about NPSPACE, because it turns out that PSPACE = NPSPACE, a result we'll prove later.

3.3 Universal Machines

Just as we constructed universal deterministic Turing machines, we can construct universal nondeterministic Turing machines. First notice that we can use the deterministic universal machine construction given in the first lecture here as well. This gives a single universal nondeterministic machine with logarithmic overhead in time and constant overhead in space.

For the task of simulating nondeterministic machines time-efficiently, we can do better than in the deterministic setting. If given as input the running time of the machine we wish to simulate, we demonstrate a machine that simulates the given machine with only a constant factor overhead in time.

Theorem 4. *There exists a NTM U_{NTIME} such that $t_{U_{\text{NTIME}}}(\langle M, x, 0^t \rangle) = O(\text{poly}(|M|) \cdot t)$ and for all $t \geq t_M(x)$, $U_{\text{NTIME}}(\langle M, x, 0^t \rangle) = M(x)$.*

Proof. U_{NTIME} assumes that $M(x)$ runs in time at most t and begins by guessing a computation record for $M(x)$ and writing it down on one of its work tape. For each time step, the computation record includes a guess for each of the following: motion (either L or R) for each of M's tape heads, new cell contents under each of M's tape heads, and new internal state. The length of this computation record is $O(\text{poly}(|M|) \cdot t)$.

After writing down the guessed computation record, U_{NTIME} verifies that the guessed computation record corresponds to a valid computation according to M's transition function, and that the computation ends in an accepting state. This is achieved by checking the validity of the computation for each of M's work tapes in turn. The important point is that if there is an accepting computation path for $M(x)$ of length at most t, then at least one of U_{NTIME}'s guessed computation records causes it to accept as well. We leave it to the reader to verify the time efficiency of the simulation. □

It is critical in the above proof that U_{NTIME} takes as input the amount of time to simulate M for. The construction can be modified if this information cannot be given. Namely, we run the construction above by first passing in 1 as the maximum time. If no accepting computation is found, the construction is repeated with double the amount of maximum time passed in. This is continued until an accepting computation is found. Note that if $M(x) = 1$, then this construction is correct and runs in time $O(\text{poly}(|M|)t_M(x))$. However, if there is no accepting computation path for $M(x)$, this construction never halts.

As for deterministic machines, we use the universal machines for nondeterministic machines to define a complete language - now for NP. The proof of the following is similar to the proof of Proposition 2

Proposition 3. *Let K_N be the language of tuples $\langle M, x, 0^t \rangle$ such that the NTM M accepts on input x in time t. K_N is complete for NP under $\leq_m^{\log}$.*

Hierarchy results in the nondeterministic model are more difficult to achieve than in the deterministic model, as complementation is more difficult. To perform the same diagonalization for NTMs that we did on DTMs, we would need a simple way to accept an input when a simulated machine rejects that input, and vice versa. However, an NTM accepts when any of its possible computation branches accept, and rejects only when all of its possible computation branches reject. This asymmetry seems to make complementation inefficient. Whether complementation on NTMs can be efficiently computed is unknown; this is the NP vs. coNP problem.

However, hierarchy results are still possible, as we will show in a later lecture.

4 Next Lecture

Next time, we will start by discussing the relationship between the class NP and deterministic verification.

CS 810: Complexity Theory 1/27/2007

Lecture 3: NP-Completeness

Instructor: Dieter van Melkebeek Scribe: Baris Aydinlioglu

Last time we introduced the model of a NTM, which we mentioned is not a realistic model of computation but nevertheless exactly captures the complexity of some important classes of problems. In particular, we started to turn our attention to the complexity class NP and briefly mentioned that this complexity class exactly characterizes those languages for which there exists an efficient[1] verification mechanism. In this lecture we continue our discussion of NP and its connection to efficient verifiability. We present the striking relationship between the complexity of almost all the problems known to be in NP. To formalize our results we use the tools we developed in the previous class, namely reducibility and completeness.

1 Efficient Verification

Definition 1. *We say that a language L has an efficient verifier if there exist $c > 0$ and $V \in P$ s.t. $x \in L \iff (\exists y \in \Sigma^{\leq |x|^c}) \langle x, y \rangle \in V$.*

V refers to a verification procedure that runs in polynomial time, and y refers to a short proof (i.e, a certificate, or a witness) for the membership of x in L.

It turns out that many problems of practical interest have this property of efficient verifiability. Before giving examples, we establish the connection between these class of problems and the class NP alluded to earlier.

Theorem 1. *$L \in$ NP iff L has an efficient verifier.*

Proof. $\Leftarrow$) Call L's verifier V, and call V's TM M_V. We describe a NTM N for L that runs in polytime. By definition, there is a constant c such that whenever x is in L there is a string y of length at most $|x|^c$ such that $\langle x, y \rangle$ is accepted by M_V. N simply "guesses" that string y and then (with polynomial overhead) simulates M_V.

$\Rightarrow$) Call L's NTM N. By definition, a string x is in L iff N accepts x within $|x|^c$ steps on some computation path, c being a constant. Then the description of any of those accepting paths is a certificate for x. Formally, $V = \{\langle x, y \rangle |$ *y is the sequence of nondeterministic choices that N makes on an accepting branch of computation* $\}$. We see that V is in P: the TM for V simulates N on x by simulating its nondeterministic transitions in accordance with y, and accepts iff N accepts. □

Now we give some examples of problems that are efficiently verifiable (i.e., that are in NP):

- A boolean formula is in "conjunctive normal form" (CNF) if it is a conjunction of disjunctions of literals, where a literal is a boolean variable or its negation. Our first example of an efficiently verifiable language is SAT = $\{$*boolean formulas ϕ in CNF* $|$ $\exists$ *an assignment that satisfies ϕ*$\}$. Given a satisfiable ϕ, the certificate for its membership in SAT is any satisfying assignment, which is clearly of polynomial (in fact linear) size in the length of ϕ.

[1] Recall that we capture efficiency in a time-bounded setting by the class P.

This fundamental problem is a stripped down version of its more natural form, SEARCH-SAT, a commonly occurring problem in many fields such as artificial intelligence. SEARCH-SAT further asks, given ϕ, a satisfying assignment if one exists. Even though at first sight SAT might appear to be too simple compared to SEARCH-SAT, a closer look reveals that they are equally difficult in the sense that, given ϕ, if we know a solution for either problem then we can efficiently compute the solution for the other. So SAT exactly captures the inherent complexity of the SEARCH-SAT problem.

We can argue this by using the notion of polynomial time oracle reductions: Clearly, SAT $\leq_o^{\mathrm{P}}$ SEARCH-SAT. We show that SEARCH-SAT $\leq_o^{\mathrm{P}}$ SAT by providing a procedure that, given ϕ, uses a SAT-oracle to compute a solution for SEARCH-SAT as follows. Query the oracle with ϕ, and if the answer is negative then ϕ is not satisfiable. Else pick any variable in ϕ, say x_o, and set it to any value, say true. Now query the SAT-oracle to see if ϕ with the value true substituted for x_o is satisfiable. If not, then set x_o to false. Proceed with the remaining variables and gradually build up a satisfying assignment for ϕ. The total number of oracle-queries is linear in the number of variables of ϕ. So SAT $\leq_o^{\mathrm{P}}$ SEARCH-SAT and SEARCH-SAT $\leq_o^{\mathrm{P}}$ SAT, which is abbreviated as SAT $\equiv_o^{\mathrm{P}}$ SEARCH-SAT.

- It is often more convenient to work with a different version of satisfiability with formulas that have more structure. For this purpose, we define the language 3SAT $= \{\phi \mid \phi$ is a satisfiable boolean formula in 3-CNF form $\}$. A formula is in 3-CNF form if it is in CNF form and each clause contains exactly three literals.

- A "vertex cover" in a graph G is a subset of vertices such that every edge in G is incident to at least one vertex from that subset. The language VC $= \{\langle G, k\rangle | \exists$ *vertex cover for* G *of size* $\leq k\}$ is efficiently verifiable: the certificate for membership is the purported vertex cover.

  Similar to the relationship between SAT and SEARCH-SAT, VC is the decision-version of its more naturally occurring form, namely the SEARCH-VC problem, which asks, given G, a vertex cover of minimal size. To show that VC $\equiv_o^{\mathrm{P}}$ SEARCH-VC, it is helpful to first consider an intermediate problem, namely the VC optimization problem, which asks, given G, the size of the minimal vertex cover. We leave it as an exercise to find a procedure that solves this problem by making a logarithmic number of queries to a VC-oracle. After this, the reader can verify that SEARCH-VC is no harder than the optimization problem.

- SUBSET-SUM $= \{\langle y_1, ..., y_n, t\rangle \mid \exists\ I \subseteq \{1, ..., n\}\ s.t.\ \sum_{i\in I} y_i = t\}$

2 The P vs NP question

The three problems above are examples of literally thousands of problems that are encountered in everyday science and engineering, all of which have equivalent (in the sense of polytime Turing reductions) decision-versions that are in NP, and none of which have a known efficient solution. This is referred to as the P vs NP question: clearly P $\subseteq$ NP, as any deterministic TM is also a NTM; but is P = NP? This is one of the most important problems of contemporary mathematics due to its philosophical and practical significance. A philosophical interpretation of this question is whether being able to efficiently verify proofs implies the ability to efficiently come up with those proofs. When stated this way its answer seems obvious, nevertheless this question remains open.

On the practical side, a positive answer to this question would have utopic consequences[2] for science and engineering, but would also bring the demise of public-key cryptography, as we will see later in the semester.

3 The NP vs co-NP question

Given a complexity class $\mathcal{C}$, we define co-$\mathcal{C}$=$\{\bar{L}|L \in \mathcal{C}\}$. The NP vs co-NP question relates to the question of whether nondeterministic computations can be easily complemented. Recall from last lecture the asymmetry between a nondeterministic computation that yields a "yes" answer and one that yields a "no" answer: on a given input, an NTM rejects only if every possible computation path rejects that input. Equivalently, a co-nondeterministic computation accepts an input iff every possible computation path accepts it.

The NP vs co-NP question asks whether having an efficient NTM for a language implies the existence of an efficient NTM for the complement of that language. Similar to the P vs NP question, the answer intuitively seems to be negative: it doesn't look like we can take the question of membership in an NP-language and efficiently translate it into a question of membership in a co-NP language and vice versa. Nevertheless this problem is also open.

The NP vs co-NP question relates to the topic of proof complexity, which can be summed up as the question of whether tautologies have short witnesses. Similar to the way that the satisfiability problem captures the complexity of the class NP as we discuss below, it follows that the problem of whether a formula is a tautology captures the complexity of the class co-NP. We will cover proof complexity at a later lecture.

Notice in passing that P = NP implies NP = co-NP but not necessarily the other way around.

4 NP-Completeness

It turns out nearly all the problems for which there is an efficient verification procedure but for which we don't know of an efficient solution, are different manifestations of the very same problem. That is, the inherent complexity behind them is so tightly connected that if one of them can be efficiently solved then so can all the rest.

The next theorem states the connection of the language SAT to the entire class NP under polynomial time mapping reductions. For didactic reasons, we show this result for a simplified model of computation, namely a single-tape TM with sequential access. The result extends to the random access model since we can simulate a random access machine on a sequential machine with a quadratic blowup in the running time of the former.

Theorem 2. *SAT is complete for* NP *under* $\leq_m^{\mathrm{P}}$. *Further, each bit of the mapping reduction is computable in logarithmic space and polylogarithmic time.*

Proof. (Sketch) We already showed above that SAT $\in$ NP. We now need to show that every problem in NP polynomial time mapping reduces to SAT. For this, fix a language L $\in$ NP. Let M be an NTM that decides L and that runs in time n^c on an input of length n, where c is a constant. In the rest of the proof we discuss how, for any string x, the question of x's membership in L can be

[2]One might question the utility of being able to solve a problem in time, say $O(n^{1000})$, but historical evidence suggests that once a problem is pulled in to the class P, it is quickly trimmed down to friendlier exponents.

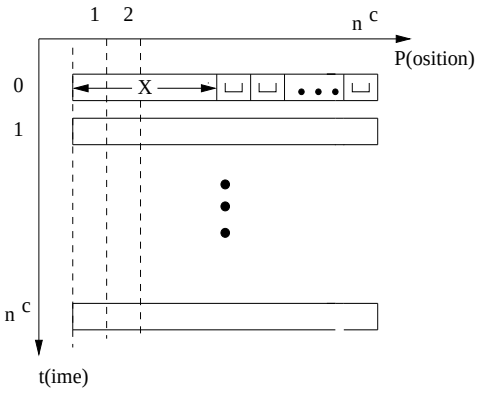


Figure 1: The computation tableau of M on input x

efficiently translated into a question of satisfiability of a CNF formula, ϕ_x. That is, given $x \in \Sigma^n$ we construct a CNF formula ϕ_x in polytime in n such that $x \in \mathrm{L(M)} \iff \phi_x \in SAT$. We do this by considering the computation tableau of M on input x.

A *computation tableau* of M on input x depicts the contents of the tape of M in successive steps during a computation of M on x. Clearly, we need only consider the tape positions up to n^c, and time steps up to n^c.

ϕ_x comprises variables and clauses. We first describe its variables. For each time step, we have variables that describe the configuration of M at that time step. Therefore for all $0 \leq t \leq n^c$, and for all $q \in Q$, we have a boolean variable $y_{t,q}^{(state)}$ which is true iff M is in state q at time step t while computing on x. In addition, for all $0 \leq t \leq n^c$, and for all $1 \leq p \leq n^c$, we have $y_{t,p}^{(tapehead)}$ which is true iff N's tape head is located at the p^{th} position of its tape at time step t while computing on x. Also, for all $a \in \Sigma$ we have $y_{t,p,a}^{(tapehead)}$, which is true iff N's tape contains the symbol a at the pth position of its tape at time step t while computing on x.

Next we describe the clauses of ϕ_x. The clauses basically force the variables of ϕ_x to have their intended meaning. That is, we set-up these clauses so that they are satisfiable iff there is a setting of the variables that describes a valid accepting computation of M on input x. To achieve this, ϕ_x contains: i) clauses that capture the initial configuration of M, ii) clauses that express the valid transitions of M (valid tape head movements and valid evolution of tape contents at each time step, in accordance with the transition relation δ of M), and iii) clauses that express the final accepting configuration of M at time step n^c. There are many details involved in setting up these clauses, we briefly touch on some here and leave the rest to the reader.

For example, since the 0th row in the tableau corresponds to the initial configuration of M on x, we have the unit clauses $(y_{0,q_0}^{(state)} \wedge \bigwedge_{i \neq 0} \overline{y}_{0,q_i}^{(state)})$ to demand that M be at its initial state before the first step of the computation. Also associated with the initial configuration we have the unit clauses $(y_{0,0}^{(tapehead)} \wedge \bigwedge_{0<p} \overline{y}_{0,p}^{(tapehead)})$ to specify the tape head position of M. In addition, we have $(\bigwedge_{p>n} y_{0,p,\sqcup}^{(contents)} \wedge \bigwedge_{p>n, a \neq \sqcup, a \in \Sigma} \overline{y}_{0,p,a}^{(contents)})$ to require that M's tape only contain blanks beyond the input x. Further, denoting the ith symbol of x as x_i, we have $(\bigwedge_{p \leq n} y_{0,p,x_p}^{(contents)} \wedge \bigwedge_{p \leq n, a \neq x_p, a \in \Sigma} \overline{y}_{0,p,a}^{(contents)})$ to specify that the input is x.

Since the contents of a cell cannot change if the tape head was not on that cell the step before, for each $1 \leq p \leq n$ and $0 \leq t \leq n^c$ we have the clause $(y_{t,p}^{(tapehead)} \vee \bigwedge_{a \in \Sigma} (y_{t,p,a}^{(contents)} \leftrightarrow y_{t+1,p,a}^{(contents)}))$

(we don't write it in CNF form here for readability.)

For the cell on which the tape head is positioned at time (i.e, row) t, there may be multiple possibilities for the next position of the tape head and the contents of the cell depending on the transition relation δ of M. For example, if $\{(q_2, b, R), (q_3, c, L)\} \subset \delta(q_1, a)$, then for each $1 \leq p \leq n$ and $0 \leq t \leq n^c$ we have the clauses

$$(y_{t,p}^{(tapehead)} \wedge y_{t,q_1}^{(state)} \wedge y_{t,p,a}^{(contents)}) \rightarrow$$
$$(y_{t+1,p+1}^{(tapehead)} \wedge y_{t+1,q_2}^{(state)} \wedge y_{t+1,p,b}^{(contents)} \wedge \psi_R) \vee (y_{t+1,p-1}^{(tapehead)} \wedge y_{t+1,q_3}^{(state)} \wedge y_{t+1,p,c}^{(contents)} \wedge \psi_L) \ ,$$

where ψ_R is $(\bigwedge_{i \neq p+1, 1 \leq i \leq n^c} \overline{y}_{t+1,i}^{(tapehead)} \wedge \bigwedge_{j \neq q_2, j \in Q} \overline{y}_{t+1,j}^{(state)} \wedge \bigwedge_{s \neq b, s \in \Sigma} \overline{y}_{t+1,p,s}^{(contents)})$ and ψ_L is left to the reader.

In the end, the formula ϕ_x contains $O(n^{2c})$ variables and is constructible in time polynomial in the length of x. The freedom of setting the variables of ϕ_x corresponds to M's freedom of making nondeterministic choices while computing on x, and thus ϕ_x is satisfiable iff there is a valid computation path of M that accepts x.

The resulting formula ϕ_x has a very simple structure. Due to this structure the reader can verify that each individual bit of the formula can be computed very efficiently, namely in time polynomial in the length of the position(address) of the bit to be computed, or equivalently, in time polylogarithmic in the size of the input x.

This establishes the mapping reduction, and the proof is complete.

□

4.1 Completeness for NQLIN

Next we present a theorem that shows a much tighter result on the complexity of reductions that connect certain problems in NP to the SAT problem. Namely, SAT is also complete for those problems in NP that can be solved in quasi-linear time, under quasi-linear time mapping reductions. Moreover, in proving the hardness part of the theorem, we do not make any simplifying assumptions on the model of computation like we did in the previous theorem.

We start by defining the complexity classes related to quasi-linear time in the obvious way.

- $\mathrm{QLIN} = \cup_{c>0} \mathrm{DTIME}(n(\log^c(n)))$
- $\mathrm{NQLIN} = \cup_{c>0} \mathrm{NTIME}(n(\log^c(n)))$

Theorem 3. *SAT is complete for NQLIN under $\leq_m^{QLIN}$. Further, each bit of the mapping reduction is computable in logarithmic space and polylogarithmic time.*

Proof. As in Theorem 2, we first show that SAT $\in$ NQLIN, then discuss how any language in NQLIN can be reduced in quasi-linear time to SAT.

First, observe that SAT can be computed in quasi-linear time by a nondeterministic random access machine that first guesses in linear time a satisfying assignment and then evaluates the formula with the guessed assignment in quasi-linear time.

To show that SAT is NQLIN-hard, we begin by making a key observation. In principle, a quasi-linear-time nondeterministic machine M can access locations on non-index tapes that have addresses of quasi-linear length. We claim that without loss of generality, we can assume that these addresses are at most of logarithmic length. The reason is that we can construct a NTM M' that simulates M with only a constant factor overhead in time and satisfies the above restriction. For

each non-index tape τ of M, M' uses an additional non-index tape τ' on which M' stores a list of all (address,value) pairs of cells of τ which M accesses and that have an address value of more than logarithmic length. During the simulation of M, M' uses τ in the same way as M does to store the contents of the cells of τ with small addresses; it uses τ' for the remaining cells of τ accessed by M. M' can keep track of the (address,value) pairs on tape τ' in an efficient way by using an appropriate data structure, e.g., sorted doubly linked lists of all pairs corresponding to addresses of a given length, for all address lengths used. Note that the list of (address,value) pairs is at most quasi-linear in size so the index values M' uses on τ' are at most logarithmic. M' can retrieve a pair, insert one, and perform the necessary updates with a constant factor overhead in time by exploiting the power of nondeterminism to guess the right tape locations[3]. Thus, M' simulates M with a constant factor overhead in time and only accesses cells on its tapes with addresses of at most logarithmic length.

Next, similar to the proof of Theorem 2, with each step in a computation of M' we associate a block of boolean variables. Each block represents the following information at the beginning of a particular time step: i) the internal state of the machine, ii) the configuration (i.e, the contents and the tape head position) of all index tapes, iii) the tape head positions of all non-index tapes, iv) the contents of each cell that is under a tape head, and v) the transition that the machine is about to take at that step. Notice that this information is all that is needed to advance M''s execution from that particular step to the next.

Each block needs to contain only $O(\log n)$ many variables to be able to capture its intended information: i), iv) and v) can be represented by a constant number of variables, while in light of our earlier observation ii) and iii) can be described by $O(\log n)$ many variables.

We use these blocks of variables to set up clauses in such a way that the clauses are satisfied iff the blocks represent a valid accepting computation of M' on a given input. We achieve this by checking: (i) that the initial block corresponds to a valid transition out of an initial configuration of M', (ii) that all pairs of successive computation steps are consistent in terms of the internal state of M', the contents of the index tapes, and the tape head positions of all tapes that are not indexed, (iii) that the accesses to the indexed non-input tapes are consistent, (iv) that the accesses to the input tape are consistent with the input x, and (v) that the final step leads to acceptance. As in the proof of Theorem 2, conditions (i), (v), and each of the linear number of constituent conditions of (ii) can be expressed by clauses of polylogarithmic size using the above variables and additional auxiliary variables. Each bit of those clauses can be computed in polylogarithmic time and logarithmic space. All that remains is to show that the same can be done for conditions (iii) and (iv).

We check the consistency of the accesses to the indexed non-input tapes for each tape separately. For tape τ, one (inefficient) way to perform the consistency check is to look at all pairs of blocks and verifying that, if they accessed the same cell of τ, and if no other block in between accessed that cell, then the contents of that cell in the second block is as dictated by the transition encoded in the first block. While this approach captures the essence of the formulation, it isn't adequate for us due to the quadratic overhead it introduces.

This construction can be made efficient by first sorting the blocks, for each non-index tape τ, in a stable way[4] on the value of the tape head location in each block. Then we can perform the

[3] A deterministic simulation would incur a logarithmic overhead in time, which would be fine for our purposes, but would require a more involved data structure like a balanced search tree.

[4] A stable sort is one that exchanges the order of two elements only when it has to.

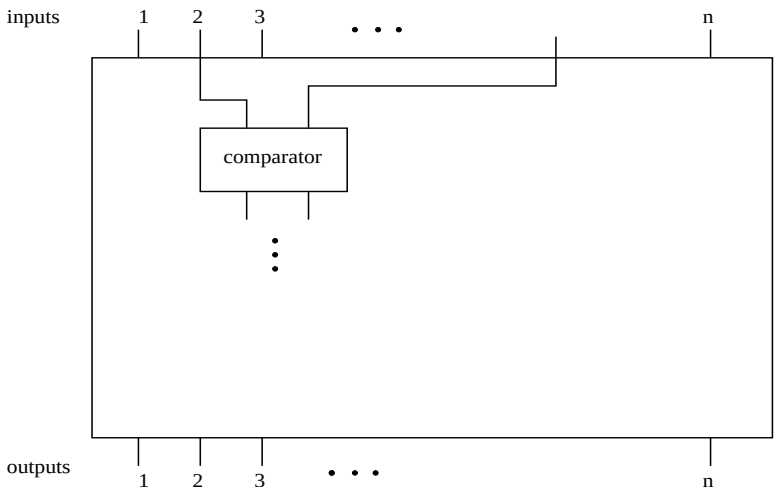


Figure 2: A simple diagram of a sorting network

consistency check for tape τ by looking at all pairs of *consecutive* blocks and verifying that, if they accessed the same cell of τ, the contents of that cell in the second block is as dictated by the transition encoded in the first block, and if they accessed different cells, then the contents of the cell in the second block is blank. These conditions can be expressed in the same way as (ii) above. Note that a stable sort is necessary in order to maintain the order of accesses to the same tape cell.

We now construct boolean clauses that mimic a deterministic sorting procedure. Since we are avoiding quadratic overheads, we focus on any $n \log n$ sorting procedure. At this point, it may look as if we hit a block: in order to efficiently formulate a quasi-linear computation, we need to efficiently formulate another quasi-linear computation. We are not stuck, however; the latter computation is of a very specific type, one which lends itself to quasi-linear formulations, as we see next.

We formulate the sorting procedure by making use of *sorting networks*. Sorting networks are a specific type of circuit with n inputs and n outputs which, given n input values, each of length $\log n$, outputs those inputs in stable-sorted order. The circuit consists of a single type of element, called comparator element, which is basically a stable-sorting box for two elements. See Figure 2.

Without going into further detail we use the fact that there exist easily computable sorting networks of size $O(n \log^2 n)$. There are a number of constructions that yield this result, among which Batcher's networks are worth mentioning due to their simplicity. These are built using the merge-sort divide-and-conquer strategy, where each (so-called odd-even) merger network is constructed using another level of divide-and-conquer. We refer the reader to the algorithms book CLRS for more on sorting networks.

We associate a block of boolean variables with each connection in the network and include clauses that enforce the correct operation of each of the comparator elements of the network. The latter conditions can be expressed in a similar way as condition (ii) above. The size and constructibility properties of the network guarantee that the resulting Boolean formula is of quasi-linear size and such that each bit can be computed in polylogarithmic time and logarithmic space.

The consistency of the input tape accesses with the actual input x can be checked in a similar way as condition (iii). The only difference is that before running the stable sorting for the input tape, we prepend n dummy blocks, the ith of which has the input tape head set to location i. The

approach for handling condition (iii) then enforces that all input accesses are consistent with the values encoded in the dummy blocks. Since we know explicitly the variable that encodes the ith input bit in the dummy blocks, we can include simple clauses that force that variable to agree with the ith bit of x.

So we have established a quasi-linear mapping reduction from any problem in NQLIN to the SAT problem, and the proof is complete.

□

Strikingly, it turns out that all of the known naturally occurring NP-complete problems are actually NQLIN-complete. Hence these problems in NP are really connected in a very tight sense. Ignoring polylogarithmic factors, they can all be solved in the same amount of (nondeterministic) time.

Because we have already shown that SAT is NQLIN-complete and QLIN reductions are transitive, proving NQLIN-completeness for subsequent languages is an easier task: to show L is NQLIN-complete, we must only show that SAT $\leq_m^{QLIN}$ L. In fact, each time a new problem is proved NQLIN-complete, this gives us another possible tool for proving subsequent problems NQLIN-complete. To give a flavor for these reductions, we prove NQLIN-completeness of two additional problems: 3SAT and VC.

Theorem 4. *3SAT is complete for NQLIN under* $\leq_m^{QLIN}$.

Proof. We give a quasi-linear time reduction from SAT to 3SAT. Given a formula ϕ in CNF form, all that needs to be done is to output an equivalent formula ϕ' in 3-CNF form. Clauses containing three literals in ϕ are transferred to ϕ' without modification. A clause containing only one or two literals is converted into a clause with three literals by repeating one or two of the literals already contained in the clause.

Consider a clause containing four literals: $(\ell_1 \vee \ell_2 \vee \ell_3 \vee \ell_4)$. This clause can be converted into an equivalent 3-CNF formula by introducing new variables: $(\ell_1 \vee \ell_2 \vee z) \wedge (\overline{z} \vee \ell_3 \vee \ell_4)$. This new formula is satisfiable if and only if the original clause was satisfiable. In general, a clause of the form $(\ell_1 \vee \ell_2 \vee \ell_3 \vee ... \vee \ell_k)$ is converted into the formula

$$(\ell_1 \vee \ell_2 \vee z_1) \wedge (\overline{z_1} \vee \ell_3 \vee z_2) \wedge (\overline{z_2} \vee \ell_4 \vee z_3) \wedge ... \wedge (\overline{z_{k-3}} \vee \ell_{k-1} \vee \ell_k).$$

We leave it to the reader to verify this reduction takes quasi-linear time. □

The reduction from SAT to 3SAT is quite simple as the two problems are very closely related. The following proof gives a reduction that is more typical of those required to proof NP-completeness.

Theorem 5. *VC is complete for NQLIN under* $\leq_m^{QLIN}$.

Proof. We prove VC is NQLIN-complete by reducing from SAT. Given a CNF formula ϕ, we show how to convert it into a graph G and integer k so that $\phi(x)$ is satisfiable iff the minimum size of a vertex cover in G is k.

Let ϕ have m clauses $c_1, c_2, ..., c_m$ and n variables $x_1, x_2, ..., x_n$. As with many reductions from 3SAT or SAT, the basic idea is to create a set of gadgets C for each clause and a set of gadgets X for each variable and connect them in an appropriate fashion. The gadget C_j for clause c_j is a complete graph over as many vertices as there are literals in c_j, where each vertex represents a

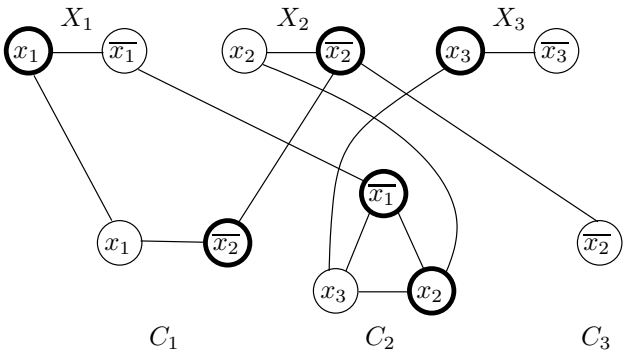


Figure 3: The graph generated by the reduction from SAT to VC for the formula $\phi = (x_1 \vee \overline{x_2}) \wedge (x_3 \vee x_2 \vee \overline{x_1}) \wedge (\overline{x_2})$. The assignment $x_1 = true, x_2 = false, x_3 = true$ satisfies ϕ, and the induced vertex cover is highlighted in the graph.

literal in the clause. The gadget X_i for variable x_i is a two vertex complete graph, where one vertex represents x_i and the other represents $\overline{x_i}$. The gadgets are connected in the following way. Let v be a vertex representing variable x_i in a clause c_j. Then an edge is inserted into the graph connecting v with the gadget X_i. If x_i appears positively in c_j, the connection is made to the portion of the gadget representing x_i, otherwise it is made to the portion representing $\overline{x_i}$. An example of the construction is given in Figure 3.

Now consider the minimum size of a possible vertex cover. For the gadgets representing variables, at least one of the vertices must be chosen. For a gadget representing clause c_j, at least $|c_j| - 1$ of the vertices must be chosen. Then a vertex cover in the graph must have size at least $n + \sum_{j=1}^{m}(|c_j| - 1)$. In fact, there is a vertex cover of this size if and only if ϕ is satisfiable, and our reduction outputs the graph as described and $n + \sum_{j=1}^{m}(|c_j| - 1)$ as the target vertex cover size. We demonstrate the reverse implication, and leave the other direction as an exercise. Given a satisfying assignment to ϕ, we first take one vertex from each of the variable gadgets according to the assignment. Because the assignment satisfies every clause, this choice of variables already covers at least one edge going out of each clause gadget. Choosing the other $|c_j| - 1$ vertices from each clause gadget completes the vertex cover. We leave it as an exercise to verify that the graph can be generated in adjacency list form in quasi-linear time. □

5 Next lecture

The fact that all naturally occurring NP-complete problems are also NQLIN-complete brings two questions: 1) Are there any complete problems in NP that are not in NQLIN? If there is a sufficiently strong hierarchy for nondeterministic time, then clearly there are. Recall that due to the asymmetry involved, obtaining hierarchy results for nondeterministic computation is more difficult. We will address this issue next time and obtain some results.

2) Assuming that P $\neq$ NP, are there problems in NP\P that are not NP-complete, namely are there any NP-intermediate problems? We will see that there do exist NP-intermediate problems, although they might not be natural. In fact, the general belief is that the current natural NP problems for which we do not yet know an efficient solution, and which we cannot show to be NP-complete either, such as graph isomorphism or factoring, are actually either NP-complete or

efficiently solvable but we haven't been clever enough so far.

CS 810: Complexity Theory 1/29/2007

Lecture 4: Time-Bounded Nondeterminism

Instructor: Dieter van Melkebeek Scribe: Baris Aydinlioglu

Last time we presented the strong connection among the problems that are in NP, which followed from the proof that the SAT problem is NP-complete under polynomial time mapping reductions. We strengthened this result by showing an even more efficient reduction that runs in quasi-linear time, establishing the NQLIN-completeness of SAT. We mentioned that almost all of the known natural problems in NP for which a polynomial-time algorithm was unknown were eventually shown to be NP-complete, and even NQLIN-complete. We finished the last lecture by asking two questions: If P differs from NP, then i) are there any complete problems in NP that are not in NQLIN, ii) are there problems in NP that are not NP-complete? In the first part of today's lecture we tackle these two questions in order. We answer the first question by obtaining hierarchy results for non-deterministic polynomial time, and the second by explicitly constructing an NP-intermediate language.

In the remainder of the lecture we discuss the concept of relativization, which will help explain why the P vs NP question has defied the efforts of many researchers for decades.

1 Nondeterministic Time Hierarchy

We start by showing that in the nondeterministic setting, similar to the deterministic case, with more time we can do strictly more.

Theorem 1. *For any two time bounds $t', t : \mathbb{N} \to \mathbb{N}$ such that $t(n) = \omega(t'(n+1))$ and t is time-constructible, we have* $\mathrm{NTIME}(t'(n)) \subsetneq \mathrm{NTIME}(t(n))$.

Proof. Recall why we cannot directly apply the diagonalization technique that we have used in separating deterministic time. In the proof of the deterministic time hierarchy, with polynomially more time in our hands, we were able to use the universal DTM to simulate the computation of a DTM with a lower time bound and then flip the result, hence differ from all deterministic computations with the lower time bound. In the nondeterministic setting, on the other hand, although we can efficiently simulate an NTM, we do not know how to negate the output of an NTM without using exponential time[1]. Since a hierarchy result with exponential jumps in time is not interesting, we need to come up with a different technique.

What we use is a modified version of the diagonalization technique from the proof of the deterministic time hierarchy, named *delayed diagonalization*, which is based on the following idea: just as in the deterministic setting, we try to diagonalize against all machines running in the lower time bound t', one by one, but instead of ensuring that we disagree with each machine on one particular input, we ensure that we disagree somewhere in an interval of inputs (we don't care exactly which particular input we disagree on, in fact we won't know).

To be able to simulate a machine on an interval of inputs, in contrast with the deterministic case where we mapped input i to machine M_i, here we map intervals of inputs I_i to machine M_i.

[1]Specifically, the problem lies in the case where the simulated machine accepts on some branches and rejects on others.

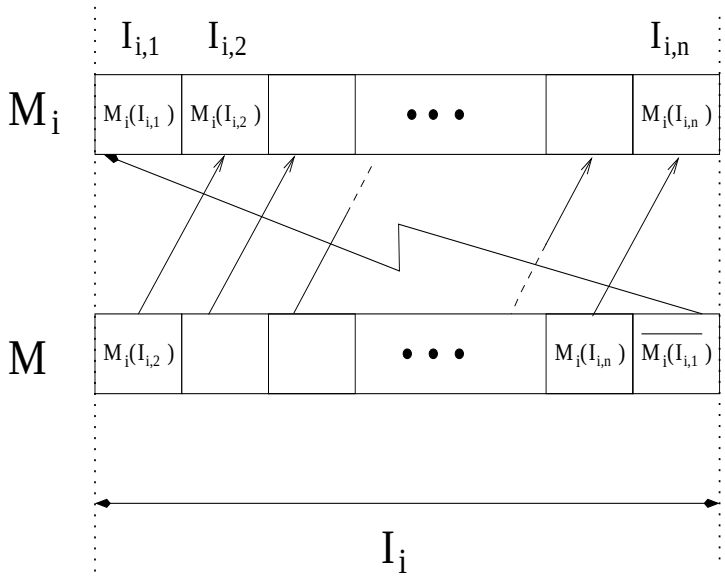


Figure 1: The delayed diagonalization of M on M_i in interval I_i. For each element of the interval except the last one, M nondeterministically simulates M_i on the next element of the interval. For the last element of the interval, M deterministically simulates M_i on the first element of the interval and flips the result.

On interval I_i, M attempts to diagonalize against machine M_i, and while doing so it will not use any more time than it is allowed, which is defined by the function t. As was the case for the deterministic time hierarchy, it must be the case that an infinite number of intervals is allocated for each machine M' - this ensures that the asymptotic behavior of t and t' guarantee that M has enough time to complete the construction for at least one of the intervals associated with M'. This can be achieved, although we skip these details in the present discussion. For a detailed example on how this mapping can be done, see p67 of the book by Arora & Barak.

The diagonalization is done as follows. Let the interval I_i contain n elements (we will find out what n must be shortly). Let $I_{i,k}$ denote the k^{th} element of I_i. The intervals are set up so that each element in the interval is 1 larger in size than the previous element so that for $1 \leq j \leq n-1$ we have $|I_{i,j}| = |I_{i,j+1}| - 1$ (for example, we could use $I_{i,1} = 0^m$ and $I_{i,k} = 0^{m+k-1}$ for some integer m and each $2 \leq k \leq n$). For $1 \leq j \leq n-1$, on $I_{i,j}$, M simulates M_i on $I_{i,j+1}$. Notice that for at least one of the intervals associated with M_i, M has enough time to simulate M for t' time steps on each $I_{i,j+1}$ since $t(|I_j|) = \omega(t'(|I_{j+1}|))$ and we can do universal nondeterministic simulation with a constant factor overhead in time. The behavior of M on the last element $I_{i,n}$ of the interval is quite different, and dictates the value of n. On $I_{i,n}$, M *deterministically* simulates M_i on $I_{i,1}$, and negates M_i's result. Notice that in order to be able to do this using a brute force deterministic simulation, M needs exponentially more time than how long it takes for M_i to halt on $I_{i,1}$. Therefore, the size of the interval, n, must be large enough so that $t(|I_n|) > 2^{t'(|I_{i,1}|)}$. See Figure 1.

We claim that M disagrees with M_i on some element of the interval. To see why, suppose towards a contradiction that for $1 \leq j \leq n$ we have $M(I_{i,j}) = M_i(I_{i,j})$. In particular, we have $M(I_{i,n}) = M_i(I_{i,n})$. By the way M was constructed, we also have $M(I_{i,n-1}) = M_i(I_{i,n})$ so $M(I_{i,n-1}) = M(I_{i,n})$. By repeated application of this argument, we get $M(I_{i,1}) = M(I_{i,n})$ and

therefore $M_i(I_{i,1}) = M(I_{i,1}) = M(I_{i,n})$. However, M is constructed so that $M(I_{i,n}) = \overline{M_i(I_{i,1})}$, and we have reached a contradiction.

So we have shown that $L(M) \notin \text{NTIME}(t'(n))$, and the proof is complete. □

Theorem 1 tells us in particular that we can do strictly more in quadratic time than what we can do in quasi-linear time. This brings us to the answer of the first question we had raised, the proof of which we leave as an exercise.

Corollary 1. *There exist languages that are* NP*-complete but not NQLIN-complete.*

2 Existence of NP-Intermediate Problems

We now obtain the answer to our second question.

Theorem 2. *If* $\text{P} \neq \text{NP}$ *then there are* NP*-intermediate problems.*[2]

Proof. Notice that the statement of the theorem does not give any restrictions on the type of reductions involved. Therefore our proof must show this result in its strongest possible interpretation, namely under polynomial time oracle reductions. Hence we will display the existence of a language L in NP\P, such that L is not hard for NP even under $\leq_O^P$.

The main idea behind the proof is similar to the delayed diagonalization technique we employed in Theorem 1. We construct a language L that differs from all languages that have polytime DTM's, i.e. from all languages in P, by making L disagree with any possible polytime DTM on some input. Moreover, we make L not NP-hard by ensuring that the language for SAT disagrees on some input with any possible DOTM that uses an oracle for L. And we do these in such a way that L is in NP.

The first thing to be done then is to enumerate all polytime DTM's, and all polytime DOTM's with access to an L-oracle. We can do this as follows. Consider the set of all strings, $\Sigma^* = \{N_1, N_2, N_3, ...\}$. Given a member N_i of Σ^*, we interpret it (or parse it, if you will) in two different ways:

(1) as a pair $\langle M_j, n^k \rangle$, where M_j is a DTM that is clocked by time n^k. For brevity, call this interpretation $\Gamma(N_i)$.

(2) as a pair $\langle M_j^L, n^k \rangle$, where M_j^L is a DOTM that has access to an L-oracle and that is clocked by time n^k. Call this interpretation $\Psi(N_i)$.

It should be clear that any reasonable interpretation would give us the enumeration desired.[3]

Now, we want to construct $L \in \text{NP}$ such that for $i \geq 1$ two conditions are met:

(1) $\Psi(N_i)$ fails to serve as a reduction for the language SAT.

(2) $\Gamma(N_i)$ decides a language different from L.

[2]The converse is also true, but is trivial.

[3]For example, given a string, we can parse its first half as the description of a TM and the second half as a time bound.

By meeting condition (1) we ensure that L is not NP-hard. For if L were NP-hard then there would be some polynomial time oracle reduction that reduces SAT to L, and meeting condition (1) over $i \geq 1$ explicitly rules out all such reductions. Similarly, with condition (2) we realize $L \notin \mathrm{P}$. For reasons that will become clear later, name condition (1) as C_{2i-1}, and condition (2) as C_{2i}.

In order to satisfy these two conditions, we construct L so that it somehow "interpolates" between some language in P and the language SAT. By interpolate, we mean on some intervals of inputs L agrees with the language in P, and on some others it agrees with SAT. On an interval where L agrees with the language in P, we make the interval large enough so that it contains a string on which the DOTM corresponding to that interval disagrees with SAT. This satisfies condition (1). On an interval where L agrees with SAT, we make the interval large enough that it contains a string on which the DTM corresponding to that interval disagrees with SAT. This satisfies condition (2). It doesn't matter which language in P we pick, so we just take the empty set, $\emptyset$.

We first present a construction that makes L meet the two conditions. Later we modify this construction so that L ends up in NP.

(1) $L \longleftarrow \emptyset$, $y \longleftarrow \varepsilon$(the empty string)
Construct L in phases. In phase i, realize conditions C_{2i-1} and C_{2i}, respectively:
(3) **foreach** phase $i = 1, 2, 3, \ldots$
(4) Since $L \in \mathrm{P}$, for any polytime DOTM M^L equipped with an L-oracle, there are infinitely many inputs on which M^L disagrees with SAT. In particular, this is the case for $\Psi(N_i)$. Let w be the lexicographically smallest string that comes after y such that $\Psi(N_i)$ disagrees with SAT on y.
(5) $L \longleftarrow (L \cap \Sigma^{\leq |w|}) \cup (SAT \cap \Sigma^{>|w|})$. In words, make L agree with SAT starting with strings of length $|w+1|$ and larger. Notice that now $L \notin \mathrm{P}$.
(6) Since $L \notin \mathrm{P}$, for any polytime DTM M, there are infinitely many inputs on which M differs from L. In particular, this is the case for $\Gamma(N_i)$. Let y be the lexicographically smallest string that comes after w such that $\Gamma(N_i)$ disagrees with SAT on w (and thus also with L.)
(7) $L \longleftarrow (L \cap \Sigma^{\leq |y|}) \cup (\emptyset \cap \Sigma^{>|y|})$. In words, make L agree with $\emptyset$ starting with strings of size $|y+1|$ and larger. Notice that now $L \in \mathrm{P}$.

As mentioned, L is not quite in NP yet. But it is close. In order to decide the membership of a string x in L, the only difficulty is in efficiently finding which of SAT or $\emptyset$ L agrees with on x. Once this is found out, the rest is an NP-computation (in case L agrees with $\emptyset$, the computation is trivial).

Given x, if we were not constrained on time, we could find out which of SAT or $\emptyset$ L agrees with on x by just running the above procedure until we hit x. The problem lies in steps 4 and 6—it may take too long, time exponential in $|x|$, to find a suitable string y on which N_i (in the form $\Psi(N_i)$ in case of step 4, $\Gamma(N_i)$ in step 6) disagrees with SAT.

We fix this problem by employing a kind of delayed diagonalization. Recall that the idea behind delayed diagonalization is to spread out each interval so that towards the end of the interval, the elements are so much larger than those at the beginning that we can brute-force check an otherwise

hard-to-compute condition on the earlier elements. Specifically, we modify our construction in steps 4 and 6 by "waiting long enough" so that it becomes easy to detect a disagreement between SAT and N_i. In other words, a new interval in L is not begun until we can verify that condition for the present interval has been satisfied on some smaller input. The modified construction for L is presented below. The procedure COND takes the length of the input string x and returns the index of the condition that L realizes on that length. If the returned value is odd, then we know that L realizes condition (1) on x, i.e. L agrees with $\emptyset$ on x. If it is even, then we know that L realizes condition(2) on x, i.e. L agrees with SAT on x.

(1) **if** COND($|x|$) is odd **then** reject
(2) **else** return SAT(x)

PROCEDURE COND($n \in \mathbb{N}$)
(1) **if** $n = 0$
(2) **return** 1
(3) Compute COND($n-1$)
(4) Check the first n strings in lex order to see whether they witness $C_{\text{COND}(n-1)}$. Clock each check for n steps only. If can't decide in the given time bound, then conclude witness was not found.
(5) **if** witness found
(6) **return** COND($n-1$)+1
(7) **else**
(8) **return** COND($n-1$)

Given n, the procedure COND recursively calls itself on decreasing values of n. In step (4), we must compute SAT on the first n strings in lex order to verify if N_i agrees with SAT. As these n strings are of length $O(\log n)$, this can be done with a brute force search in polynomial time. Given this fact, the reader can verify that COND runs in polynomial time, and thus $L \in$ NP as desired while still satisfying each condition C_{2i-1} and C_{2i}.

□

3 Relativization

Definition 1. Relativizing a *(complexity theoretic)* statement with respect to *(a language)* A *means giving each machine involved in that statement access to an A-oracle. We say that* a statement *(theorem, proof, etc.)* holds relative to A *if it holds when it is relativized with respect to A. We say that* a statement relativizes *if it holds with respect to any language.*

Example: Consider the statement

$$\text{NTIME}(n) \subsetneq \text{NTIME}(n^2), (*)$$

which follows from Theorem 1. There are two types of machines involved in this statement, those that run in time $O(n)$ and those that run in time $O(n^2)$. We relativize this statement with respect

to an oracle for some language A, by giving all of those machines access to an A-oracle. We express this modification by writing NTIME(n) as NTIME$^A(n)$, and NTIME(n^2) as NTIME$^A(n^2)$.

We say that (*) holds relative to A if

$$\text{NTIME}^A(n) \subsetneq \text{NTIME}^A(n^2).(**)$$

We say that (*) relativizes if (**) holds for any language A.

We claim that (*) relativizes. To see why, notice that the proof of Theorem 1 applies almost verbatim. The only modification needed is, during its simulation,whenever M_i uses the A-oracle (by writing onto the query tape, by transitioning to the query state, or by reading the query output), the simulator M does the same for its A-oracle. To state succinctly, the proof of the theorem relativizes and thus so does its statement. ⊠

Example: Recall from lecture 2 the language K_N, containing all tuples $\langle M, x, 0^t \rangle$ such that the NTM M accepts on input x in $\leq t$ steps. We had shown that K_N is complete for NP under $\leq_m^{\log}$.

Define, given a language A, $K_N^A = \{\langle M, x, 0^t \rangle \,|M^A$ halts and accepts x in $\leq t$ steps$\}$. As in the previous example, the proof of NP-completeness of K_N relativizes. Therefore, given any language A, K_N^A is complete for NP^A under $\leq_m^{\log}$. ⊠

One conclusion we can draw from the last example is that for any given language A, there is a NOTM N such that $L(N^A) = K_N^A$. But we can observe more. Consider two relativizations of the proof of completeness of K_N, one with respect to a language A, and another to a language A'. In the two proofs, the only difference between N^A and $N^{A'}$ is their oracles. Hence we reach a stronger result: there exists a *fixed* NOTM N such that for *any* A we have $L(N^A) = K_N^A$. In fact, N is essentially our efficient universal NTM. This remark will be a key ingredient in the proof of the next theorem.

3.1 Limits of the simulation technique and the P vs NP question

A fact which turns out to be very revealing is that almost all the results in complexity theory relativize. In particular, the kinds of techniques that we have used so far, such as diagonalization and lazy diagonalization, all relativize. This is because at the core of these techniques lies simulation; if we equip both the simulator and the simulated with a certain oracle, then all of the statements in the proofs without the oracle carry us through to yield proofs with the oracle.

There are many other techniques that we will see through the semester, nevertheless almost all of them relativize. In fact, those techniques that do not relativize are so infrequent that the reader should assume (and verify) that a given statement relativizes unless told otherwise.

What makes this important is our next result: There exist oracles relative to which P differs from NP, but there also exist oracles relative to which P equals NP. In other words, neither of the statements "P = NP" and "P $\subsetneq$ NP" relativizes. Before we prove this, we note its significance.

The implication of this corollary is that none of the techniques that we have seen so far (in general those that relativize) are enough to solve the P vs NP question. To settle this problem we need a new idea.

Theorem 3. *There exist oracles A and B s.t.* P^A = NP^A *and* P$^B \neq$ NP^B.

Proof. (First half) We start with the construction of the oracle (i.e., the language) A, relative to which P equals NP. The usual proof of this fact uses a PSPACE-complete language as the oracle A. We give an alternate proof that illustrates the techniques used in oracle constructions. We mention the alternate proof in a later lecture. We will construct A in such a way that we can build a DOTM M^A that runs in polytime and that decides a NP^A-complete language, giving us the first half of the theorem. The NP^A-complete problem that M^A solves will be K_N^A.

The key idea is the following. Call the NOTM for K_N^A N^A. We want to somehow make M^A as powerful as N^A. We will achieve this by making A compensate for M^A's lack of nondeterministic power. Specifically, we will construct A so that it contains the results of N^A's computations.

This strategy contains a pitfall that must be avoided. At any point during the construction of A, whenever we store a computation result of N^A in the language A, we must be careful not to affect any of N^A's computations *prior* to that point in the construction.

In order to address this issue, we make a key observation. Given an input of length n, the longest oracle query that N^A can make is of length $n-1$. To see why, recall how N^A works: it simulates the NOTM encoded by the first part of its input, for at most the number of steps equal to the length of its input, n. But if the simulated machine has n steps to execute, the longest query it can make to the oracle is of length $n-1$, since it needs 1 step to actually perform the query.

This observation clears the way for the construction of A. Similar to the proof of Theorem 2, we will build A in phases. At the end of phase i, we will have assigned the membership to A of all strings of length $\leq i$. Here is the procedure:

```
(1)   A ← ∅
(2)   foreach phase i = 0, 1, 2, ...
(3)      foreach x ∈ Σ^i
(4)         if N^A(x) accepts
(5)            A ← A ∪ {x}
```

We can now elaborate on how this procedure works as intended. Consider the fully constructed A. Let A_i stand for A at any point in phase i during its construction. Then we have, for any phase i, that the set $A \backslash A_i$ does not contain a string of length $< i$, i.e., whatever string we add to A in phase i or subsequent phases is of length i or longer. In light of the observation above regarding the length of oracle queries that N^A makes with respect to its input size, this means that the addition of a string x to A_i does not affect any of the computations of N^A on inputs of length $\leq |x|$. So we do in fact avoid the pitfall mentioned earlier. As a parenthetical remark, observe that when considering $x \in \Sigma^i$ we need not follow any particular order.

With A constructed this way, M_A becomes trivial. On input x, M_A simply queries the oracle with x and returns the result of the query.

We now argue that $L(M^A) = K_N^A$. To see the containment from left to right, suppose that $x \in L(M^A)$. By the way M^A was constructed, this implies that $x \in A$. By the way A was constructed, this implies that x is accepted by $N^{A'}$, where $A' \subset A$. Again by the construction of A, this implies that N^A accepts x.

For the containment from right to left, suppose that N^A accepts x. Then, by construction, A contains x and thus M^A accepts x.

This completes the proof of the first half of the theorem.

(Second half)

We prove the second half by constructing a language through exploiting the power of nondeterminism. Consider the language $L^B = \{0^n | (\exists x \in \Sigma^n) x \in B\}$. Regardless of what the language B is, $L^B \in \mathrm{NP}^B$: given x, a NOTM N^B can guess a string of length $|x|$ in B. In what follows we construct B such that $L^B \notin \mathrm{P}$.

Let $N_1, N_2, N_3, ...$ be an enumeration of all polynomial DOTM's, i.e. all DOTM's clocked with all running times of the form n, n^2, n^3, etc. As the case in previous proofs, we assume each machine appears in the enumeration infinitely often. Without loss of generality, let N_i have a running time of n^i.

We build B also in phases. In phase i, we realize the condition $C_i : L^B \neq L(N_i^B)$. In each phase, we fix the oracle on strings longer than those considered in the previous phase. This ensures that the computation results of a DOTM N_j^B from any previous phase $j < i$ remain unaffected. Here is the procedure for constructing B.

```
(1)   B ← ∅
(2)   f(0) ← −1
(3)   foreach phase i = 1, 2, 3, ...
(4)       pick an integer k such that k > f(i − 1) and k^i < |Σ|^k
(5)       if N_i^B(0^k) rejects
(6)           pick a string y of length k which N_i has not queried while deciding 0^k
(7)           B ← B ∪ {y}
(8)       f(i) ← k^i
```

$f(i)$ keeps track of the maximum length of the string that N_i could have queried given the construction so far. In line 4, we pick k large enough so that strings of length k have not been set yet and so that N_i cannot query all strings of length k. Then we run N_i on 0^k and if it accepts, we do not put in B any string of length k, thereby realizing condition C_i. In case N_i rejects, we put into B a k-length string which N_i has not queried, hence realizing C_i.

No polynomial time DOTM decides L^B, which completes the proof of the second half of the theorem.

□

4 Next lecture

In the next lecture we will move from our current time-bounded setting to the space-bounded one, where we investigate the power of computation with limits on the amount of space that can be used. We will see results that suggest that the space-bounded setting is better understood, for example that nondeterministic space is closed under complementation.

CS 810: Complexity Theory 1/31/2007

Lecture 5: Space-Bounded Nondeterminism

Instructor: Dieter van Melkebeek Scribe: Matthew Anderson

Last lecture we discussed time-bounded nondeterminism. We discussed hierarchy results that showed given a little more time we could solve a strictly larger subset of problems. We also proved the existence of NP-Intermediate problems (under the assumption that P $\neq$ NP). Finally, we explored relativization showing that there exists an oracle, A, for which $\mathrm{P}^A = \mathrm{NP}^A$ and an oracle, B, for which $\mathrm{P}^B \neq \mathrm{NP}^B$, this implies that techniques that relativize are not sufficient for resolving the P versus NP question.

1 Overview

Today we continue our discussion of nondeterminism by looking at space-bounded nondeterminism. We are able to prove stronger results in the space-bounded case than in the time-bounded case. We prove three main results relating space-bounded nondeterminism to other complexity classes.

Theorem 1. $\mathrm{NSPACE}(s(n) \subseteq \cup_{c>0} \mathrm{DTIME}(2^{cs(n)})$

Theorem 2. $\mathrm{NSPACE}(s(n)) \subseteq \mathrm{DSPACE}(s^2(n))$

Theorem 3. $co\,\mathrm{NSPACE}(s(n)) = \mathrm{NSPACE}(s(n))$

The proof of all the results assume that we have a space computable function, $s(n) : \mathbb{N} \to \mathbb{N}$ such that $s(n) \geq \log n$. The restriction that $s(n)$ must be space computable is not necessary for the proofs of these theorems, we discuss how to remove this restriction. We require $s(n) \geq \log n$, because space bounds lower than logarithmic cause strange behavior and make it difficult for the machine to act in an interesting manner on its linear input.

Theorem 1 says that everything that can be done in NL can be done in P and everything that can be done in NPSPACE can be done in EXP. Theorem 2 gives us that in only quadratically more space we can perform nondeterministic computation deterministically. No similar result is known for time-bounded computation. This result is made possible by the fact that space can be reused, while time cannot. The third result, Theorem 3, follows from that fact that it is easier to compute the complement of a language in space-bounded computation because you can iterate over all of the witnesses and check that none are valid. We will actually prove that $co\,\mathrm{NSPACE}(s(n)) \subseteq \mathrm{NSPACE}(s(n))$, but by complementing twice we get the equality. Theorem 3 implies a tight space hierarchy for nondeterministic machines by using diagonalization.

We can use these three theorems along with several results we have seen before to determine the containment of a number of complexity classes:

$$\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subset \mathrm{PSPACE} = \mathrm{NPSPACE} \subseteq \mathrm{EXP} \subseteq \mathrm{NEXP}. \quad (1)$$

Most of the containments follow from the definition of the various classes. The containments NL $\subseteq$ P and NPSPACE $\subseteq$ EXP follow directly from Theorem 1, PSPACE = NPSPACE from Theorem 2 and NP $\subseteq$ PSPACE follows from a simple proof that a PSPACE machine can iterate through all possible witnesses checking whether they verify membership.

Whether these inclusions are strict is an open problem, the only separations that are known come from the hierarchy results which we discussed in previous lectures. For example, we know $\mathrm{L} \subsetneq \mathrm{PSPACE}$ but not whether $\mathrm{L} \subsetneq \mathrm{NP}$.

Corollary 1. *If $s, s' : \mathbb{N} \to \mathbb{N}$ such that $s(n)$ is space constructable and $s(n) = \omega(s'(n))$ then $\mathrm{NSPACE}(s'(n)) \subsetneq \mathrm{NSPACE}(s(n))$.*

The proof of this corollary follows the same diagonalization strategy as we have seen before. Since complementation is easier in the space domain a constant factor is sufficient to accomplish strictly more.

Recall, from Lecture 1, that our definition of space usage of a machine is the index of highest indexed cell it accesses. If a machine runs in space s, the machine's state (tape heads, tape state, computation state) can be described using $O(s)$ bits.

2 Proof of $\mathrm{NSPACE}(s(n)) \subseteq \cup_{c>0} \mathrm{DTIME}(2^{cs(n)})$

Proof. Consider a NTM M running in space $s(n)$. Fix an input $x \in \Sigma^n$. Consider the configuration graph, $G = (V, E)$, of M on input x. Each vertex represents a possible configuration of M that does not use more that $s(n)$ space. It follows that:

$$|V| \leq 2^{cs(n)} \cdot n \leq 2^{c's(n)}, \tag{2}$$

for constants c, c' that depend on M. The first inequality follows from the amount of space used and the number of possible positions of the input tape head. The second inequality follows from the fact that $s(n) \geq \log n$. The edge $(u, v) \in E$ iff there is a valid transition between the configurations u and v under δ. Note that since the input tape cannot change its state information is not contained in the configuration therefore the configuration vertices are independent of the values on the input tape, but the edges are dependent.

If there is a path from the start configuration to some accepting configuration then M would have accepted on input x and so should our simulation. We can make the accept state unique by requiring that M clear up all work tapes and reset tape heads to the starting position before moving into the accepting state. This transforms the problem of M accepting x to the problem of graph reachability between the unique start configuration and the unique accepting configuration. We can solve the reachability problem in time polynomial in the size of the graph using standard algorithms (the graph size in this case is $2^{cs(n)}$), this gives the result $\mathrm{NSPACE}(s(n)) \subseteq \cup_{c>0} \mathrm{DTIME}(2^{cs(n)})$. □

This proof works nicely if $s(n)$ is space constructable. If this is not the case or we do not know the value of $s(n)$, we can run the procedure several times while increasing a fixed space-bound until the computation has enough space to complete. More precisely, iterate over space bounds $s = 1, 2, 3, \ldots$. If the computation reaches an accepting state, then accept. If the computation does not accept but tries to exceed the space bound, repeat this procedure with a larger space bound. If the computation does not on any path try to exceed the space bound, then accept if there is a path to an accepting state, otherwise reject.

Corollary 2. *The Directed Path Problem (DPP) is $\leq_m^{\log}$-complete for* NL.

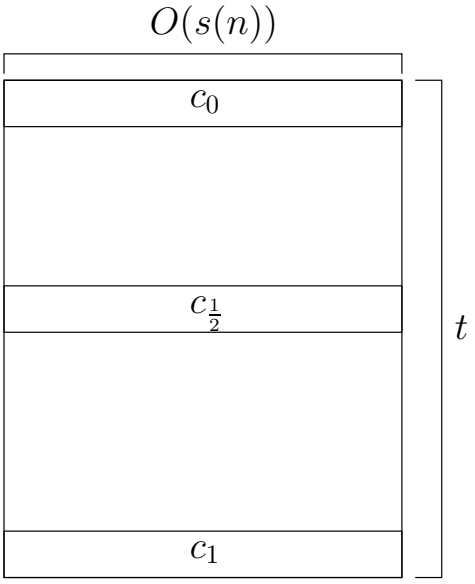


Figure 1: An illustration of the configuration tableau used in the proof of Theorem 2. The tableau has width $O(s(n))$ to describe the configuration at each step and runs for time t which is bounded by $2^{cs(n)}$.

Proof. The DPP is in NL because you can store the location of the vertex you are currently visiting and guess the next vertex to visit, continue this procedure until you reach the destination or have traveled more than $|V|$ steps. The DPP is hard for NL because you can apply it to configuration graphs for any NL machine. □

This corollary implies that if DPP $\in$ L we have L = NL. The DPP is also known as the st-connectivity problem (STCON). It is not known if DPP $\in$ L, however, the undirected version of the problem is.

3 Proof of NSPACE($s(n)$) $\subseteq$ DSPACE($s^2(n)$)

Proof. Consider a NTM M running in NSPACE($s(n)$). Fix input $x \in \Sigma^n$. Consider the computation tableau of M. In our original discussion of computation tableau we applied the techniques to a one-tape Turing machine. Although we are using a random access model, the configuration can still be described in $O(s(n))$ bits.

Label the starting configuration c_0, label the unique accepting configuration c_1 (using the resetting idea from before). In principle the computation time t could be infinite. However, if there exists at least one accepting computation then there exists an accepting computation that does not repeat states so we have $t \leq 2^{cs(n)}$. In fact we can assume that the computation always takes exactly t time by forcing the accepting state to repeat itself. A diagram of the tableau is shown in Figure 1.

Again, we would like to determine if there are a sequence of states that take us from c_0 to c_1 in t time and using no more than $s(n)$ space. A naive approach will take exponential space. To achieve the quadratic space blow up we use a divide and conquer strategy. Instead of going directly

from c_0 to c_1 we guess an intermediate state $c_{\frac{1}{2}}$ and verify independently that we can go from c_0 to $c_{\frac{1}{2}}$ and $c_{\frac{1}{2}}$ to c_1. The benefit of doing it this way is that we can reuse the space used in the first part of the computation when we do the second part.

The highest level of the computation will look something like this:

$$x \in L(M) \Leftrightarrow (\exists c_{1/2})(\forall b \in \{0,1\}) c_{\frac{b}{2}} \vdash_M^{\frac{t}{2}} c_{\frac{b+1}{2}} \tag{3}$$

This reduces the original problem into two subproblems of half the size. We can apply this procedure recursively reusing the space that was taken up in the computation of the first part of the problem. Because we are halving the size of the problem at each level after $\log t$ recursions we reach the base case where we have to verify that one state transitions to another in one step. We can phrase this as a full quantified boolean formula:

$$x \in L(M) \Leftrightarrow (\exists c_{\frac{1}{2}})(\forall b_1 \in \{0,1\})(\exists ...)(\forall ...)...(\exists c_{\frac{x}{t}})(\forall b_y \in \{0,1\}) c_{\frac{x+b_y-1}{t}} \vdash_M^1 c_{\frac{x+b_y}{t}} \tag{4}$$

The index x in the last part of Equation 4 is dependent on the previous choices for the b variables. The predicate checks whether the configurations guessed to come immediately before and after $c_{\frac{x}{t}}$ have valid one step transitions to and from $c_{\frac{x}{t}}$.

Consider how much space is required to evaluate this boolean formula. Each existential quantifier requires $s(n)$ space to describe the configuration. Each universal quantifier requires one bit of information to store its value. There are $\log t$ pairs of quantifiers so the total amount of space required is $O(s \cdot (\log t + 1)) = O(s^2)$.

□

Again we are implicitly assuming that s is space constructable, however we can apply the same technique as before to remove this requirement.

3.1 TQBF

The boolean formula which was introduced in the previous proof is called a quantified boolean formula. These types of formulas are part of an interesting language called true quantified boolean formulas.

Definition 1 (True Quantified Boolean Formula (TQBF))**.** *The language of all quantified boolean formulas which have a number of quantifier alternations, contain no free variables and are true.*

Example: $(\exists x_1)(x_1 \wedge \bar{x_1}) \notin$ TQBF ⊠

Example: $(\forall y_1)(y_1 \vee \bar{y_1}) \in$ TQBF ⊠

Corollary 3. TQBF *is complete for* PSPACE *under* $\leq_m^{\mathrm{P}}$.

Proof. The proof of Theorem 2 gives a polynomial mapping reduction for a general PSPACE problem transforming it into a TQBF in polynomial time making TQBF hard for PSPACE. TQBF is in PSPACE because a PSPACE machine can iterate over all possibilities. □

Note that PSPACE $\subseteq \mathrm{P}^{\mathrm{TQBF}}$ by the corollary and $\mathrm{P}^{\mathrm{TQBF}} \subseteq$ PSPACE because all queries to the oracle can be constructed in polynomial time and we have closure under polynomial composition. Therefore $\mathrm{P}^{\mathrm{TQBF}} = \mathrm{PSPACE}$. Taking this idea further we get the following containment:

$$\mathrm{P}^{\mathrm{TQBF}} = \mathrm{PSPACE} = \mathrm{NP}^{\mathrm{TQBF}} \subseteq (\mathrm{PSPACE}^{\mathrm{TQBF}} = \mathrm{PSPACE}). \tag{5}$$

The last equality is model dependent because it matters whether you count the cells used on the oracle tape or not (if the oracle tape is not counted, a $\mathrm{PSPACE}^{\mathrm{TQBF}}$ machine could abuse the oracle tape to compute more than an unassisted PSPACE machine could). This also gives another example of an oracle (this time a natural one) for which $\mathrm{P}^A = \mathrm{NP}^A$. As before, note that techniques that relativize cannot be the sole means to solve the P versus NP question.

4 Proof of $co\,\mathrm{NSPACE}(s(n)) = \mathrm{NSPACE}(s(n))$

Proof. Consider a NTM M using space $s(n)$ on a fixed input $x \in \Sigma^n$. We saw before that given a graph and two vertices that reachability can be guessed reachability in $\log(|V|)$ space. This theorem shows that the complement can also be computed in log space, by showing that a path *does not* exist.

The key idea behind this proof is that if it is known how many vertices are reachable from the start vertex u, then it can be verified that the destination vertex v is not one of those vertices by looking at all possible vertices and guessing and verifying whether they are reachable. In order to count the number of reachable vertices we use a procedure call inductive counting.

Suppose you have a subroutine that behaves as follows. The subroutine takes as input the graph G, the start vertex u, the destination vertex v, the number of steps allowed t and the number of vertices c_t within t steps of u. The subroutine has three possible return values: YES, NO and ?. A YES answer means that it is possible to reach v from u in $t+1$ steps. A NO answer means that it is not possible to reach v from u in a most $t+1$ steps. A ? return value means that the procedure was unable to determine the answer. Consider the ? return value a value that indicates that the subroutine either made bad internal guesses or got bad input, in either case it's output could not be trusted and that computation path is compromised. This subroutine will run in NL, is always correct if it answers YES or NO, and at least one computation path returns YES or NO. If value passed as c_t is not correct the subroutine will return ?.

We will describe the subroutine more specifically later.

Consider how we can use the subroutine to solve the original problem. Start with $c_0 = 1$ and try to compute c_1. Set $c_1 = 0$. Run the subroutine for all possible v. If the subroutine says YES add one to c_1 and continue, if the subroutine says NO continue, if the subroutine says ? halt and reject. The final value of c_1 after looping through all possible v's is the number of vertices reachable from u in one step. Iterate this procedure to compute c_{n-2}. Only the current and next values of c_i need to be stored at any point in the computation. Finally run the subroutine one last time with parameters (G,u,$n-1$,c_{n-2},v), if the answer is NO, accept if the answer is ? or YES reject.

The entire computation only accepts if there exists a computation path that correctly tracks the counts and v is not reachable from u. If the last step says YES there was a path from u to v. If the last step says ? the computation was inconsistent and it bails out by rejecting.

4.1 Writing the Subroutine

The idea for the subroutine is simple: cycle through all the vertices and guess if they are reachable. If the subroutines guess they are reachable it tries to verify they are reachable. If a vertex is reachable, the subrountine modifies the count of reachable vertices seen so far then checks if v is reachable in one additional step. The count must match the count passed in as a parameter to verify that v was not reached.

SUBROUTINE(G,u,t,c_t,v)
Input: A graph G, a starting vertex u, a number of steps t, the number of vertices within t steps of u c_t, the destination vertex v
Output: YES if v is reachable from u in $\leq t+1$ steps, NO if v is not reachable from u in $let+1$ steps, ? if the computation is not able to determine. If c_t is correct at least one computation path will return the correct answer (not ?).

```
(1)   c ← ∅
(2)   foreach w ∈ V(G)
(3)     Guess whether w is reachable from u in ≤ t steps.
(4)     if Guessed yes
(5)       Verify guess by nondeterministically guessing a path from u to w in
          ≤ t steps.
(6)       if Successfully verified
(7)         if v can be reached from w in one step
(8)           return YES
(9)         else
(10)          c ← c + 1
(11)      else
(12)        return ?
(13)  if c = c_t
(14)    return NO
(15)  else
(16)    return ?
```

□

The main point of this technique is that if the number of positive instances (reachable vertices) is known guessing and verifying positive instances will verify whether the count is correct. Then the count can be used to check whether the instance in question is a member of the positive set.

5 Next Time

Next lecture we will discuss alternation as a generalization of the hypothetical model of nondeterministic computation. In particular alternation captures all languages that can be specified with a fixed number of quantifier alternations as a TQBF.

CS 810: Complexity Theory 2/2/2007

Lecture 6: Alternation

Instructor: Dieter van Melkebeek Scribe: Piramanayagam Arumuga Nainar

In the past few lectures, we studied in depth the non-deterministic model of computation. In this lecture, we will extend it to a more general model: alternation. We derive results for the classes of problems that can be solved using alternation. We also discuss three other characterizations of this new model. One of them is the alternating Turing machine, a generalization of non-deterministic Turing machines.

1 Motivation

Even though non-determinism is a hypothetical model, it is important because it captures the complexity of several interesting problems. For example, NP, the class of problems that can be solved in non-deterministic polynomial time contains languages whose membership can be verified efficiently (i.e. in polynomial time). Let us state that more formally:

$$\text{A language L} \in \text{NP iff there exists a } c > 0 \text{ and } V \in \text{P such that } x \in L \Leftrightarrow (\exists y \in \Sigma^{\leq |x|^c})\langle x, y\rangle \in V \tag{1}$$

However non-determinism is still too restrictive to efficiently solve some interesting problems. One such example is that of minimizing a CNF formula. Consider the equivalent decision problem of testing whether a CNF formula is of minimum size. Let us define MIN-CNF $= \{\phi | \phi$ is in CNF and is of minimum size$\}$. We could state membership in MIN-CNF as follows:

$$\phi \in \text{MIN-CNF} \Leftrightarrow (\forall y \text{ of size } < |\phi|)(\exists x)\phi(x) \neq y(x) \tag{2}$$

Any CNF y that is smaller than ϕ can certainly be expressed as a string of size polynomial in $|\phi|$. The same is the case for an assignment x of values to the variables in ϕ. Moreover, checking whether $\phi(x) = y(x)$ can be done in polynomial time (It takes two SAT verifications followed by a test of equality). So stmt. 2 can be rewritten (to better resemble the formulation of NP above) as follows:

$$\phi \in \text{MIN-CNF} \Leftrightarrow (\forall y \in \Sigma^{\leq |\phi|^c})(\exists x \in \Sigma^{\leq |\phi|^c})\langle \phi, y, x\rangle \in V \text{ for some } V \in \text{P} \tag{3}$$

If we look at the logical formulas used to describe membership in a language in NP(stmt. 1) and MIN-CNF(stmt. 3), we notice two significant differences: the latter has two quantifiers and one of them is an universal quantifier. This is the generalization we are going to make in the alternating model. If we wish to allow more than one quantifier in formulas expressing membership constraints in languages, they must be alternating between universal and existential quantifiers. If not, two adjacent quantifiers of the same type can be combined into one, leaving a quantified variable that is polynomial in the size of the input.

2 The Alternating Model

We define a new, general complexity class Q_k^p as the set of languages whose membership constraints can be expressed using formulas with k alternate existential and universal quantifiers, with each

quantified variable of size polynomial in the length of the input, and the initial quantifier being existential for $Q = \Sigma$ and universal for $Q = \Pi$.

A more notational description of Σ_k^p and Π_k^p is given below:

Definition 1. $\Sigma_k^p = \{L | \textit{membership in } L \textit{ can be expressed by a formula of the form } x \in L \Leftrightarrow (\exists y_1 \in \Sigma^{\leq |x|^c})(\forall y_2 \in \Sigma^{\leq |x|^c}) \ldots (Qy_k \in \Sigma^{\leq |x|^c}) \langle x, y_1, y_2, \ldots y_k \rangle \in V \textit{ for some } V \in \mathrm{P} \textit{ and some constant } c\}$

Definition 2. $\Pi_k^p = \{L | \textit{membership in } L \textit{ can be expressed by a formula of the form } x \in L \Leftrightarrow (\forall y_1 \in \Sigma^{\leq |x|^c})(\exists y_2 \in \Sigma^{\leq |x|^c}) \ldots (Qy_k \in \Sigma^{\leq |x|^c}) \langle x, y_1, y_2, \ldots y_k \rangle \in V \textit{ for some } V \in \mathrm{P} \textit{ and some constant } c\}$

Notes: The quantifier Q for y_k in Σ_k^p is $\exists$ if k is odd and $\forall$ if k is even. The dual is true for Π_k^p. The super-script p in Σ_k^p and Π_k^p denotes poly-time verifiability. There is a small notational inconsistency. Σ is used to denote a class of problems as well as to denote the input alphabet. But the intended usage is usually clear from context.

2.1 Facts

If $k = 0$, there are no quantified variables and the verifier V can decide membership just by looking at the input. In other words, V is an algorithm for L and Σ_0^p is the same as P. In the absence of any quantifiers, there is no real distinction between existential and universal quantification. Thus, $\Sigma_0^p = \Pi_0^p = \mathrm{P}$.

The definition of Σ_1^p is exactly the same as that of NP given earlier and so, $\Sigma_1^p = \mathrm{NP}$. We will just state here, and prove a general result later, that $\Pi_1^p = \mathrm{coNP}$. And finally, the description of MIN-CNF matches that of Π_2^p, hence MIN-CNF $\in \Pi_2^p$

3 Class Hierarchy results

Proposition 1. $\Pi_{k-1}^p \subseteq \Pi_k^p$ *and* $\Sigma_{k-1}^p \subseteq \Sigma_k^p$

A simple result we can state for the classes introduced above is that we can solve more (or at least an equal number of) problems if we allow greater number of quantifier alternations. This is trivial because we can add a new, unused variable y_k and quantify it appropriately but leave V unchanged.

Proposition 2. $\Sigma_k^p \cup \Pi_k^p \subseteq \Sigma_{k+1}^p \cap \Pi_{k+1}^p$

We can transform the membership constraint for Σ_k^p to that of: (1) Σ_{k+1}^p by adding an universally quantified variable that is not used by V in the end; (2) Π_{k+1}^p by adding an universally quantified variable that is not used by V in the beginning. Hence $\Sigma_k^p \subseteq \Sigma_{k+1}^p \cap \Pi_{k+1}^p$. A similar argument applies for Π_k^p and hence the above result holds.

Proposition 3. $\Sigma_k^p = co\Pi_k^p$ *and* $\Pi_k^p = co\Sigma_k^p$

Proof. A language L is in $co\Pi_k^p$ iff we could verify in Π_k^p whether a string is not in L. This negates the membership constraint in Defn. 2. The quantifiers for $y_1, y_2, \ldots y_k$ will get switched, leaving a Σ_k predicate. The verification task will be whether $\langle x, y_1, y_2, \ldots y_k \rangle$ is not in V. Since $V \in \mathrm{P}$, we can do this efficiently. Thus, $co\Pi_k^p = \Sigma_k^p$. A similar argument holds for $\Pi_k^p = co\Sigma_k^p$ □

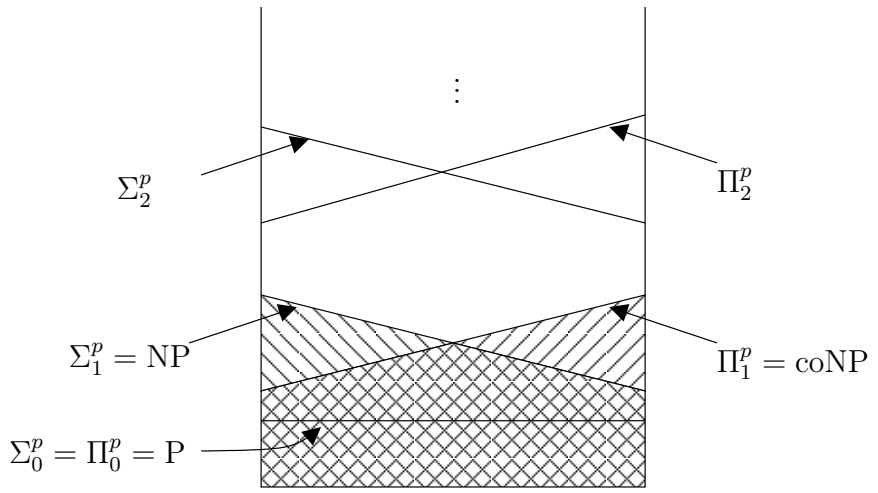


Figure 1: Pictorial illustration of the polynomial hierarchy

Propositions 1 and 2 can be illustrated using Figure 1. The k^{th} line sloping upwards bounds languages in Π_k^p and the k^{th} line sloping downwards bounds Σ_k^p. Both these lines are above the lines at the previous levels. For $k = 0$, the two lines collapse into each other, as P = *co*P.

Corollary 1. $\Pi_1^p = \text{coNP}$*, since we already derived that* $\Sigma_1^p = \text{NP}$

One question we can ask is whether the inclusion in Proposition 1 is strict. i.e. whether $\Sigma_k^p \subset \Sigma_{k+1}^p$. Even the simplest version of this question, for $k = 1$ (i.e. is P $\subset$ NP or in other words P $\neq$ NP), has turned out to be hard, so far. The next result we prove is based on the definition of a polynomial hierarchy.

Definition 3. PH *is the class of all problems in the* polynomial time hierarchy *: any fixed number of alternations are allowed. i.e.* $\text{PH} = \bigcup_k \Sigma_k^p$

Note: We need not include Π_k^p in the previous definition as a consequence of Proposition 2

Theorem 1. *If* $\Sigma_k^p = \Pi_k^p$ *for some* $k \geq 1$ *then* $\text{PH} = \Sigma_k^p$*, in other words, the polynomial time hierarchy collapses to level* k*.*

Proof. Consider a language $L \in \Sigma_{k+1}^p$. A string $x \in L$ iff the following condition is true:

$$(\exists y_1 \in \Sigma^{\leq |x|^c})(\forall y_2 \in \Sigma^{\leq |x|^c}) \ldots (Qy_k \in \Sigma^{\leq |x|^c})(\overline{Q} y_{k+1} \in \Sigma^{\leq |x|^c}) \langle x, y_1, y_2, \ldots y_k, y_{k+1} \rangle \in V \qquad (4)$$

Now, $(\forall y_2 \in \Sigma^{\leq |x|^c}) \ldots (Qy_k \in \Sigma^{\leq |x|^c})(\overline{Q} y_{k+1} \in \Sigma^{\leq |x|^c}) \langle x, y_1, y_2, \ldots y_k, y_{k+1} \rangle$ is a Π_k predicate on input $\langle x, y_1 \rangle$. Because $\Sigma_k^p = \Pi_k^p$, there is an equivalent Σ_k predicate ϕ on input $\langle x, y_1 \rangle$. The existential quantification in equation 4 can be merged with the initial existential quantifier of ϕ, thus leaving a new Σ_k predicate ϕ' on input $\langle x \rangle$. In other words, if $\Sigma_k^p = \Pi_k^p$, then every Σ_{k+1}^p language can be expressed using a Σ_k predicate. Thus, $\Sigma_{k+1}^p \subseteq \Sigma_k^p$. Combining this with proposition 1, we get $\Sigma_k^p = \Sigma_{k+1}^p$. Using multiple, alternate applications of the two operations: converting between Π_k and Σ_k predicates and merging like quantifiers together, we can prove that $\Sigma_j^p \subseteq \Sigma_k \forall j > k$. Thus, $\text{PH} = \Sigma_k^p$. □

Note: The condition $k \geq 1$ in the above theorem is required because we need at least one quantifier to make the premise $\Sigma_k^p = \Pi_k^p$ non-trivial.

Corollary 2. *If* P = NP *then* PH = P.

Proof. If P = NP, we have already proved that NP = coNP. From the above theorem, the whole polynomial hierarchy collapses to NP and thus, to P. □

¿From the above corollary, we can state that if PH does not collapse then P ≠ NP. But the former is a stronger statement. That is, even if P ≠ NP, the polynomial hierarchy may collapse to some other class. But the general conjecture in the community is that PH does not collapse. In fact there have been results that start from some assumption and derive that PH collapses in order to disprove the likeliness of that assumption.

4 Completeness

Now, we see an example of a problem that is complete in Σ_k^p.

Definition 4. *Let $T.\Sigma_k$ be the set of all true, fully quantified Σ_k formulas with a predicate that is a (1) CNF if k is odd and (2) DNF if k is even.*

Similarly, we can define $T.\Pi_k$ over the set of all true, fully quantified Π_k formulas.

Claim 1. *$T.\Sigma_k$ is $\leq_m^p$-complete for Σ_k^p and $T.\Pi_k$ is $\leq_m^p$-complete for Π_k^p.*

Proof. Consider a language $L \in \Sigma_k^p$ with associated verifier V. Suppose k is odd. Then the last quantifier is $\exists$, and the last quantifier together with V is $(\exists y_k \in \Sigma^{\leq |x|^c}) V(\langle x, y_1, y_2, ..., y_k \rangle)$. This is an NP statement, so by the NP-completeness of SAT can be converted in polynomial time into the statement $(\exists z)\phi(\langle x, y_1, y_2, ..., y_{k-1}, z \rangle)$ for some CNF formula ϕ. Then $(\exists y_1 \in \Sigma^{\leq |x|^c})(\forall y_2 \in \Sigma^{\leq |x|^c})...(\exists z \in \Sigma^{\leq |x|^{c'}})\phi(\langle x, y_1, y_2, ..., y_{k-1}, z \rangle)$ is a a $T.\Sigma_k$ instance that is true if and only if $x \in L$.

Suppose k is even. Then the last quantifier is $\forall$, and the last quantifier together with V is $(\forall y_k \in \Sigma^{\leq |x|^c}) V(\langle x, y_1, y_2, ..., y_k \rangle)$. This is a coNP statement, so by the coNP-completeness of TAUTOLOGIES can be converted in polynomial time into the statement $(\forall z)\phi(\langle x, y_1, y_2, ..., y_{k-1}, z \rangle)$ for some DNF formula ϕ. Given this, a $T.\Sigma_k$ instance is created as above.

As this reduction can be accomplished in polynomial time, $T.\Sigma_k$ is hard for Σ_k^p under $\leq_m^p$. As $T.\Sigma_k$ is in Σ_k^p by construction, $T.\Sigma_k$ is complete for Σ_k^p. A similar argument proves that $T.\Pi_k$ is Π_k^p-complete. □

5 Alternate Characterizations

5.1 Using Oracle Machines

Claim 2. $\Sigma_{k+1}^p = \text{NP}^{\Sigma_k^p}$. *In other words, the set of languages in the $(k+1)^{th}$ level of the polynomial hierarchy is the set of languages recognized by non-deterministic machines with access to oracles at the k^{th} level.*

Proof. For $k = 0$, the statement is $\text{NP} = \text{NP}^{\text{P}}$. $\text{NP} \subseteq \text{NP}^{\text{P}}$ is trivial because we can simply choose to not use the oracle. $\text{NP}^{\text{P}} \subseteq \text{NP}$ because the time complexity of the oracle will, in worst case, increase the degree of the polynomial in the time complexity.

The statement is non-trivial for $k > 0$. Consider $k = 1$. Suppose we have a language $L \in \Sigma_2^p$. Then, for some $c > 0$ and $V \in \text{P}$,

$$x \in L \Leftrightarrow (\exists y_1 \in \Sigma^{\leq |x|^c})(\forall y_2 \in \Sigma^{\leq |x|^c})\langle x, y_1, y_2\rangle \in V$$

We can construct a non-deterministic TM M that guesses the value of y_1 and tries to solve $(\forall y_2 \in \Sigma^{\leq |x|^c})\langle x, y_1, y_2\rangle \in V$. The latter is a Π_1 formula and can be solved using an oracle for Σ_1^p because $\Sigma_1^p = co(\Pi_1^p)$ and in general any language can be solved given an oracle to its complement. Thus, $L \in \text{NP}^{\Sigma_1^p}$.

Suppose $L \in \text{NP}^{\Sigma_1^p}$. We can express that L is decidable in Σ_2^p as follows:

1. Express the computation path followed by the base NP machine as a Σ_1 formula by guessing the queries made by the machine, as well as the results of the queries.

2. Express the constraints that positive query responses are valid, again by using a Σ_1 predicate.

3. Express the constraints that negative query responses are valid by using a Π_1 predicate. The universal quantifier is required because in this step, we want to ensure the non-existence of a witness to the query, rather than its existence.

Overall a Σ_2 predicate expresses the decidability of L, and $L \in \Sigma_2^p$. Thus, $\Sigma_2^p = \text{NP}^{\Sigma_1^p}$. (The right hand side can also be written as NP^{NP}.)

Arguments for $k > 2$ are similar. □

5.2 Using Boolean Circuits

Claim 3. *Any language $L \in \Sigma_k^p$ can be expressed as an exponential size boolean circuit, with $k + 1$ alternating levels of AND and OR gates of unbounded fan-in, with the last level having polynomial bounded fan-in and such that each bit of the circuit description can be computed in polynomial time, and vice-versa.*

Proof. Consider a language $L \in \Sigma_k^p$ with verifier V. We can construct an enormous boolean circuit as follows: an exponential number of polynomial circuits evaluating membership in V for all possible combinations of $y_1, y_2, \ldots y_k$ and a specific value for x. We can make these polynomial circuits to have only two levels by expressing the function $V(x, y_1, y_2, \ldots y_k)$ in CNF or DNF. The outputs of these circuits are combined hierarchically at k levels to leave a single output at the top-most level. This output is the decision whether $x \in L$. The i^{th} level contains an array of AND gates if y_i is universally quantified and an array of OR gates if y_i is existentially quantified. The number of gates in the i^{th} level is equal to the number of choices for inputs $y_1, y_2, \ldots y_{i-1}$. By appropriately choosing a CNF or DNF for the verifier V, we can merge the gates from the k^{th} level with the top gates from the normal form that is chosen. It can be verified that each bit is poly-time computable.

Now, consider a boolean circuit as described above with an OR gate at the top-most level. We must construct a Σ_k formula to compute the language decided by the circuit, i.e. we need to determine how many bits to guess for the $\exists$ and $\forall$ quantifiers and what the V will be. Consider

the top-most OR gate. Its output will be 1 if there exists a gate at the lower level whose output is 1. We can express this as

$$(\exists G_2)(\text{output of gate } G_2 \text{ is } 1) \tag{5}$$

where G_2 is a second level gate. G_2 will be an AND gate. It will produce an output 1 if all the OR gates at the third level produce an output 1. So, stmt. 5 becomes:

$$(\exists G_2)(\forall G_3)(\text{output of gate } G_3 \text{ is } 1) \tag{6}$$

where G_3 is any gate whose output is connected as input to the chosen G_2. Since the circuit is of exponential circuit, we can represent gates in the circuit using polynomial size indices. If we repeatedly apply these steps, we will get a Σ_k predicate with the final verification task being that of evaluating the output of the gate at the $(k+1)^{th}$ level. Since each bit of the circuit description is computable in polynomial time, we can use the choices of the indices at each level to identify the gate that needs to be evaluated as well as the bits from the input that go into that gate. Thus, we can evaluate the last predicate of the Σ_k formula in poly-time. Thus, we have a Σ_k^p language corresponding to the circuit. Similarly we can construct a Π_k^p language corresponding to a circuit with an AND gate at the top-most level. □

5.3 Alternating Turing Machines

In this section, we extend the non-deterministic Turing machine to model alternation. An alternating TM is a non-deterministic TM in which each non-halting state has an additional property: whether it is existential or universal. A configuration of the machine is accepting if either:

1. This is a halting configuration and the machine is in an accepting state, or
2. The current state is existential and at least one transition from this state leads to an accepting configuration, or
3. The current state is universal and all configurations from this state leads to an accepting configuration

The machine itself accepts an input if the initial configuration is accepting. The following gives our final characterization of Σ_k^p and Π_k^p.

Claim 4. $\Sigma_k^p = \{L | L$ *is accepted by an alternating TM with an existential start state that runs in polynomial time and has at most* $k-1$ *quantifier alternations*$\}$
$\Pi_k^p = \{L | L$ *is accepted by an alternating TM with an universal start state that runs in polynomial time and has at most* $k-1$ *quantifier alternations*$\}$

Proof. Consider a language $L \in \Sigma_k^p$. We can construct a k stage alternating TM M with the i^{th} stage guessing the value of y_i. To match the quantifier of the y_i's, all the states in the i^{th} stage must be existential if i is odd and universal if i is even. The verifier V of L does not require any non-determinism. So, the states that simulate V can be made existential or universal depending on k. Thus, M recognizes L and has an existential start state and performs at most $k-1$ alternations. Similarly, we can construct an alternating TM that recognizes a language in Π_k^p and satisfies the above constraints.

Consider an alternating TM M that performs at most $k-1$ alternations. We can construct an equivalent Σ_k^p or Π_k^p language L as follows. If M halts in polynomial time, the time it spends

in each stage of the alternation is also polynomial. We can model the choices made by M in the i^{th} step as a polynomial length string y_i. If i existential (universal) stage, then y_i is quantified existentially (universally). The verifier V has to verify that the choices made are valid for the given input and machine M and also that the final state is halting and accepting. This can be done in polynomial time. The resulting formula is a Σ_k formula if M's initial state is existential, and is a Π_k formula otherwise. □

6 Time and Space

Using the same definition of time and space required by a Turing machine we can define complexity classes Σ_k-TIME(t) and Σ_k-SPACE(s). We can also derive hierarchy results using the same technique used earlier, namely delayed diagonalization. These results get simplified if we allow an unlimited number of alternations. We can define ATIME(t) as the set of problems that can be solved in time t on an alternating TM with any number of quantifier alternations. ASPACE(s) is the set of all problems that can be solved using space s on an alternating TM with any number of quantifier alternations.

Some results on Time and Space

Theorem 2. NSPACE(s) $\subseteq$ *ATIME*(s^2)

Proof. This follows from our proof of NSPACE(s) $\in$ DSPACE(s^2) in last lecture. Remember that we constructed a formula very similar to a Σ_k formula in our divide-and-conquer formulation of that proof. The existential quantifier guessed an intermediate configuration ($O(s)$ long) of the Turing machine and the universal quantifier was used to specify two independent reachability conditions (1 bit is enough). There were $O(s)$ such quantifiers. The final predicate verifies whether the transition from one configuration to another is valid, which takes $O(s)$ time since each configuration is $O(s)$ long. Thus the guessing stages of the machine take $O(s^2)$ time and the verification at the end takes $O(s)$ time, for a total running time of $O(s^2)$. □

Theorem 3. *ATIME*(t) $\subseteq$ *DSPACE*(t^2)

Proof. Let M be an alternating machine running in time t. If we separate M's execution into existential and universal stages, there are at most t many and each is at most t long. We begin by simulating M as long as it remains within the first stage. For the second stage of M, we simulate it for all possible choices from the first stage. There were at most t separate choices, so we can cycle through all of these using space $O(t)$. For the third stage of M, we must simulate it for all possible choices in the first and second stages. It takes space $O(t+t)$ to cycle through all of these. As there are a total of at most t stages, we can cycle through all possible guesses of M using $O(t^2)$ space. For the guesses corresponding to an existential stage, we ensure that at least one of the guesses results in an accepting computation; for the guesses corresponding to an universal stage, we ensure that all of the guesses result in an accepting computation.

Given a sequence of guesses, we simulate M with these guesses using the space-efficient universal Turing Machine from the first lecture. If we first convert M into an equivalent machine M' that uses $O(t)$ space, the total space usage is $O(t^2 + s_M) = O(t^2 + t) = O(t^2)$. □

Corollary 3. PSPACE $=AP$

Proof. ¿From Thm. 3, AP $\subseteq$ PSPACE. Also, PSPACE $\subseteq$ NPSPACE. Thus from Thm. 2, PSPACE $\subseteq$AP. $\square$

Theorem 4. *$ASPACE(s(n)) = \bigcup_{c>0} \mathrm{DTIME}(2^{c.s(n)})$. In other words, we need time exponential in the space requirement to deterministically simulate an alternating TM.*

Proving this theorem will be a homework problem.

Corollary 4. $AL = \mathrm{P}$

Corollary 5. $APSPACE = \mathrm{EXP}$

7 Next Lecture

Next time, we will use alternation to prove the non-existence of SAT-solvers that are both time and space efficient. The main topic of next class is non-uniformity where we allow different algorithms for inputs of different lengths.

CS 810: Complexity Theory 2/5/2007

Lecture 7: Nonuniformity

Instructor: Dieter van Melkebeek Scribe: Chi Man Liu

Last lecture we studied alternations and the related polynomial-time hierarchy (PH). We gave four different characterizations of the alternation model, and also introduced some alternating time and space complexity classes. In the first part of this lecture, we make use of alternations and prove some time-space lower bound results for SAT.

In the second part of the lecture, we introduce the notion of nonuniform computation. Nonuniform models we discuss include boolean circuits, branching programs and uniform machines with advice. We conclude the lecture by looking at some connections between uniform and nonuniform models.

1 Time-Space Lower Bounds for SAT

Whether SAT can be solved in polynomial time remains an open question until the P = NP question has been resolved. Although it is very unlikely that SAT can be solved in linear time, none of the known results have ruled out this seemingly ridiculous possibility. More strikingly, it is possible (though not likely) that SAT is in L. However, if we put both time and space into consideration, we can prove some nontrivial lower bounds for SAT.

Definition 1. *Let $t(n)$ and $s(n)$ be functions from $\mathbb{N}$ to $\mathbb{N}$. The class* $\mathrm{DTISP}(t(n), s(n))$ *is defined to consist of all languages L that can be accepted by a DTM using at most $t(n)$ steps and $s(n)$ space.*

Note that $\mathrm{DTISP}(t(n), s(n))$ is not the same as $\mathrm{DTIME}(t(n)) \cap \mathrm{DSPACE}(s(n))$ in general.

Our first theorem says that no algorithm for SAT is both time- and space-efficient. The theorem is proved by considering nondeterministic linear time. We first prove a lemma that says that SAT essentially captures the power of nondeterministic linear time.

Lemma 1. *If* $SAT \in \mathrm{DTISP}(n^c, n^d)$ *then* $\mathrm{NTIME}(n) \subseteq \mathrm{DTISP}(n^c \,\text{poly-log}(n), n^d \,\text{poly-log}(n))$.

Proof. Recall that in Lecture 3 we proved that SAT is complete for NQLIN under $\leq_m^{QLIN}$ and each bit of the reduction can be computed in polylogarithmic time and logarithmic space. Given a language $L \in \mathrm{NTIME}(n) \subseteq NQLIN$ and an input of length n, we compute its reduction to SAT. The reduction has length $O(n \,\text{poly-log}(n))$. Once we have the reduction we can solve it in deterministic $O(n^c \,\text{poly-log}(n))$ time using our deterministic algorithm for SAT in the hypothesis. However, if we computed the reduction in a straightforward manner it would take up too much space — $O(n \,\text{poly-log}(n))$ space just for storing the reduction (note: d may be less than 1). Instead, we can modify our SAT algorithm to run directly on inputs for L: whenever the algorithm requires a bit of the reduction, that bit is computed from the original input. This computation of a single bit can be done in polylogarithmic time and logarithmic space. This incurs only a polylogarithmic blow-up in time. The space requirement is clearly $O(n^d \,\text{poly-log}(n))$ (the $\log n$ factor for computing the reduction on-the-fly is absorbed). □

Theorem 1. *SAT* $\notin \text{DTISP}(n^c, n^d)$ *for constants* c *and* d *such that* $c(c+d) < 2$.

Proof. By Lemma 1, it suffices to show that $\text{NTIME}(n) \not\subseteq \text{DTISP}(n^c \,\text{poly-log}(n), n^d \,\text{poly-log}(n))$. We prove this by contradiction. In this proof we only assume that $\text{NTIME}(n) \subseteq \text{DTISP}(n^c, n^d)$ and show that this leads to a violation of the nondeterministic time hierarchy theorem. The polylogarithmic factors add a $o(1)$ term in the exponent of the time and space usage. We will see that we reach the same contradiction even when we include these factors.

Our proof involves alternations which we discussed in the previous lecture. We split the proof into two parts. First we speed up deterministic computations by introducing alternations (going from DTISP to Σ_2-TIME). Then we eliminate these alternations (going from Σ_2-TIME to NTIME) using our hypothesis.

Part 1. Let L be a language in $\text{DTISP}(t, s)$. Fix an input string x, and consider the computation tableau which has $t = t(|x|)$ rows and $s = s(|x|)$ columns. Each row describes a configuration of the Turing machine at some point in time. We call the initial configuration c and the (unique) accepting configuration c'. Then $x \in L$ if and only if c' can be reached from c in t steps. In the proof of $\text{NSPACE}(s(n)) \subseteq \text{DSPACE}(s^2(n))$ in Lecture 5, we guessed an intermediate state $c_{\frac{1}{2}}$ and verified independently that we could go from c to $c_{\frac{1}{2}}$ and $c_{\frac{1}{2}}$ to c'. We will use the same technique here, but this time instead of splitting up the tableau into two parts, we split it up into b parts (value of b will be determined later). We can guess $b-1$ configurations c_1, c_2, $\ldots$, c_{b-1}, and verify that we can go from c_{i-1} to c_i in t/b steps, for $1 \le i \le b$, $c_0 = c$, and $c_b = c'$. This computation can be stated as:

$$x \in L \iff (\exists c_1, \ldots, c_{b-1})(\forall 1 \le i \le b) c_{i-1} \vdash^{t/b} c_i$$

Note that this is in fact a Σ_2 computation. We now analyze the time taken by this computation. Guessing the existential part takes time $O(bs)$ since each c_i is of length s. Guessing the universal part takes time $O(\log b)$. The verification part takes time $O(s + t/b)$ since we can easily simulate each step of the deterministic computation in constant time. Combining we get a $O(bs + t/b)$ running time. To achieve the best speedup, we minimize this value by setting b appropriately. Setting $bs = t/b$ achieves a value within a constant factor of the minimum. So $b = \sqrt{t/s}$ and the running time becomes $O(\sqrt{ts})$. Thus we have

$$\text{DTISP}(t, s) \subseteq \Sigma_2\text{-TIME}(\sqrt{ts})$$

Part 2. Let L be a language in Σ_2-$\text{TIME}(n^a)$ for some $a \ge 1$. Computation for L can be written as

$$x \in L \iff (\exists y \in \Sigma^{|x|^a})(\forall z \in \Sigma^{|x|^a}) R(\langle x, y, z\rangle) \tag{1}$$

where R is a deterministic linear time computable predicate. Note that $(\forall z \in \Sigma^{|x|^a}) R(\langle x, y, z\rangle)$ is in co-$\text{NTIME}(N)$ for fixed y with $|y| = |x|^a$, where $N = |x| + |x|^a$ (length of $\langle x, z\rangle$). Since $\text{NTIME}(N) \subseteq \text{DTISP}(t, s)$, so is co-$\text{NTIME}(N)$ as DTISP is closed under complement. Hence $(\forall z \in \Sigma^{|x|^a}) R(\langle x, y, z\rangle)$ is in $\text{DTIME}(N^c)$. It follows that (1) is in $\text{NTIME}(N^c) = \text{NTIME}(n^{ac})$, and thus

$$\Sigma_2\text{-TIME}(n^a) \subseteq \text{NTIME}(n^{ac})$$

Combining parts 1 and 2 gives us $\text{DTISP}(n^{2c}, n^{2d}) \subseteq \Sigma_2$-$\text{TIME}(n^{c+d}) \subseteq \text{NTIME}(n^{(c+d)c})$. The final step is showing $\text{NTIME}(n^2) \subseteq \text{DTISP}(n^{2c}, n^{2d})$, from which follows $\text{NTIME}(n^2) \subseteq \text{NTIME}(n^{(c+d)c})$, contradicting the nondeterministic time hierarchy theorem. This inclusion follows

from the hypothesis and a technique called *padding*. Notice that if we had included the polylogarithmic factors in the above, we would end up with an addition $o(1)$ term on the final running time - i.e. we would reach the conclusion that NTIME(n^2) $\subseteq$ NTIME($n^{(c+d)c+o(1)}$). This remains contradictory to the nondeterministic time hierarchy for c and d such that $(c+d)c < 2$.

Padding. Let L be a language in NTIME(n^2). Let $L' = \{0^{|x|^2-|x|-1}1x \mid x \in L\}$. We claim that $L' \in$ NTIME(n). Let M be a NTM accepting L that runs in quadratic time. We construct a NTM M' accepting L' as follows. On input x', L' checks whether $|x'|$ is a possible member of L' by counting the number of leading zeroes. It then extracts x from x' by removing the artificially padded prefix, and simulates the computation of L on x. $|x| = \sqrt{|x'|}$, therefore computation takes time $O(|x|^2) = O(|x'|)$. So $L' \in NTIME(n)$ and therefore also $L' \in$ DTISP(n^c, n^d).

We now use the DTISP(n^c, n^d) algorithm for L' to decide L in DTISP(n^{2c}, n^{2d}). Given x, a possible instance of L, we can pad it to x' and decide in DTISP($(n')^c, (n')^d$) if $x' \in L'$ (equivalent, $x \in L$). Since $n' = n^2$, we have solved L in DTISP(n^{2c}, n^{2d}). □

Corollary 1. *Suppose $SAT \in DTISP(n^c, n^d)$ for constants c and d.*

1. *If $d < 1$ then $c > 1$;*
2. *if $c < \sqrt{2}$ then $d > 0$.*

The first statement of the corollary says that if an algorithm for SAT is space-efficient (logarithmic space in particular), then it runs in super-linear time. The second statement says that if an algorithm for SAT is time-efficient (linear time in particular), then it uses polynomial space – logarithmic space is not sufficient.

The above theorem and its corollary show inefficiency in either time or space of a single algorithm for SAT. It might be possible that there exists a linear-time linear-space algorithm, and another log-space quadratic-time algorithm for SAT. The following lower bound result rules out this possibility.

Theorem 2. *$SAT \notin$ L$\cap$co-NTIME($n^{1+o(n)}$).*

Proof. Exercise. □

Corollary 2. *$SAT \notin$ L $\cap$ DTIME(n).*

2 Nonuniformity: Motivation

Computational models we have seen so far, such as Turing machines and random access machines, work for inputs of all lengths. We call these *uniform* models of computation. In this section, we introduce *nonuniform* models of computation.

In short, a nonuniform model is a computational model in which a different machine (more generally, a computing device) is used for each input length. For example, you can use a machine M_0 for inputs of length 0, another machine M_1 for inputs of length 2, and yet another machine M_2 for length 2, and so on. The machines M_0, M_1, M_2, ... form an infinite family of machines.

Nonuniformity may seem a bit odd at first sight. Indeed, uniformity is a more natural form of computation because it resembles algorithms – finite procedures for all possible inputs. So why are we interested in nonuniform models? There are in fact close relationships between uniform and nonuniform models. By studying nonuniform models, we may be able to derive lower bound or hardness results for uniform models. We will see some of these relationships in Section 4.

3 Nonuniform Models of Computation

We introduce three forms of nonuniform computation. The first two are nonuniform models — boolean circuits and branching programs. Both of them solve instances of a specific problem with a fixed input length. For simplicity we assume that languages are defined over the binary alphabet $\{0, 1\}$. Boolean circuits are useful in analyzing the uniform time complexity of problems; branching programs are more useful for space complexity.

The third model is in fact our old uniform model, but with a nonuniform ingredient known as *advice*. Advices are similar to certificates – both are additional information which speed up computation. The difference between certificates and advices is that while certificates can vary from input to input, all inputs of the same length share the same advice. In other words, the advice for an input only depends on its length.

We briefly discuss the three models in the following. Next section we will relate nonuniform models to uniform models.

3.1 Boolean Circuits

We define boolean circuits similarly to real-world electronic circuits. A boolean circuit is a directed acyclic graph where each node is either a logic gate or an input. An input node has no incoming edges. One of the gate nodes is designated as the output node which has no outgoing edges. Label the input nodes x_1, x_2, ..., x_n. Given input string $x \in \{0, 1\}^n$, the boolean circuit computes its output (a single bit) as follows. The bits of x are first copied to the corresponding input nodes. Then, in topological order, each logic gate receives bits from its incoming edges, performs the boolean operation on the bits, and sends the output bit along all its outgoing edges. The output of the circuit is the bit output by the output node.

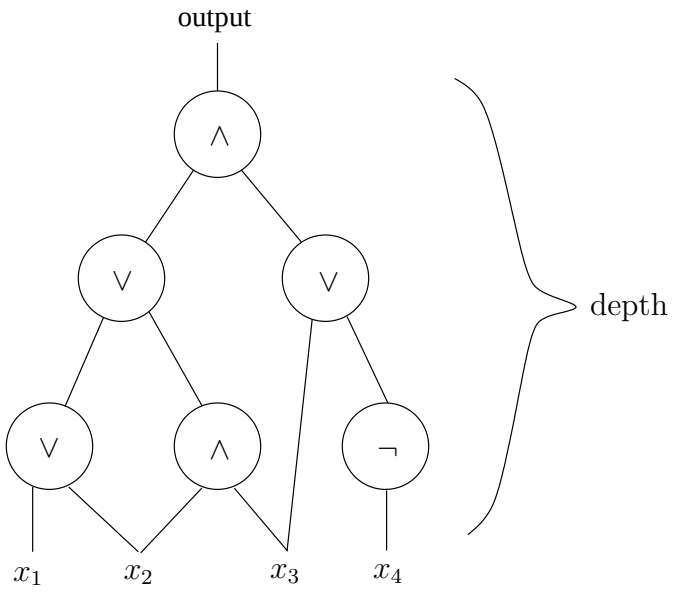


Figure 1: A boolean circuit.

For any function $f : \{0, 1\}^n \to \{0, 1\}$, we say that a boolean circuit B realizes f if the output

of B matches $f(x)$ for every input $x \in \{0,1\}^n$. In this course we consider circuits with AND, OR, and NOT gates with a bounded (say 2) fan-in. We define the *circuit size* $C(f)$ of a function f to be the size of the smallest (in terms of number of nodes) circuit realizing f.

We can also use boolean circuits to accept a language L. Instead of using one circuit for all possible inputs as in uniform computation, we must use a different circuit for inputs of different lengths. Formally, we say that a family of circuits $\{B_i\}$ accepts a language L if for every $n \in \mathbb{N}$, B_n realizes L_n, where L_n is the characteristic function of L restricted to inputs of length n. We define the *circuit complexity* $C_L(n)$ of L by $C_L(i) = C(L_i)$ for all $i \in \mathbb{N}$.

The circuit complexity of a language can be a good measure of its uniform time complexity. The reason is that, given a boolean circuit, we can simulate its computation by a uniform machine in linear time. Likewise, given a Turing machine that runs in time t, we can "encode" its transition function into a boolean circuit of size quadratic in t. Another measure is the *depth* of a circuit, which equals the length of the longest path from an input to the output node. Circuit depth is not as comparable to uniform time complexity as circuit size, since for any specific language and an input length n, we can convert membership into a CNF or DNF, and build a constant-depth circuit with fan-in 2^n. This circuit can be converted into an equivalent linear-depth circuit with a bounded fan-in of 2.

Note: For a language L, its circuit complexity $C_L(n)$ can be considerably smaller than its uniform time complexity $t_L(n)$, due to the "one algorithm for all inputs" restriction imposed on uniform computation.

3.2 Branching Programs

A branching program P is a directed acyclic graph where each node is labeled x_1, x_2, ..., x_n, ACCEPT, or REJECT. Each node (except those labeled ACCEPT or REJECT) has exactly two outgoing edges where one of them is marked 0 and the other 1. One of the nodes is designated as the start node. The computation of P on input $x \in \{0,1\}^n$ is as follows: Starting from the start node, look at its label x_i and follow the appropriate edge to the next node by looking at the input string. Repeat until it reaches an ACCEPT node or a REJECT node. Note that similar to a boolean circuit, a branching program only works for inputs with a specific length. For a function $f : \mathbb{N} \to \mathbb{N}$, we define its *branching program complexity* $BP(f)$ to be the size of the smallest branching program that accepts f.

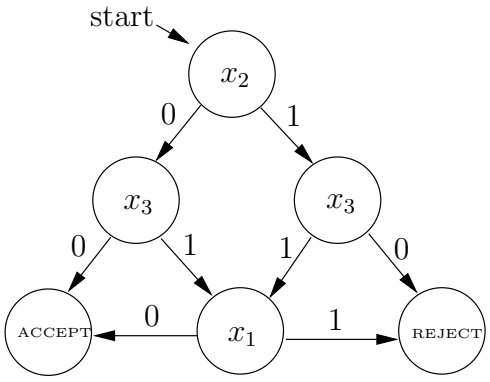


Figure 2: A branching program.

We can use a family of branching programs to accept a language. The branching program complexity $BP_L(n)$ of a language L is defined by $BP_L(i) = BP(L_i)$ for all $i \in \mathbb{N}$. (See subsection on boolean circuits).

The branching program complexity of a language can be a good measure of its uniform space complexity. Suppose we are given a branching program with v nodes. We can simulate its computation on a Turing machine using $O(\log v)$ space — the space used to store the index of the current node. Now suppose we have a Turing machine that uses space s and runs in time t. We can construct a layered branching program (a branching program whose nodes can be partitioned into layers such that every edge goes from one layer to the next layer) with $t+1$ layers and $O(2^s)$ nodes in each layer. Each node represents a machine configuration. The first layer contains a single node representing the initial configuration. This node is the start node. Each node in subsequent layers is labeled by the variable corresponding to the current input tape head position in its configuration. Edges indicate valid transitions between successive configurations. A node becomes an ACCEPT (resp. REJECT) node if it represents an accepting (resp. rejecting) configuration. The size of this branching program is $O(2^s t)$. We see from the above simulations that there is a roughly logarithmic relationship between the size of a branching program and space usage of its corresponding Turing machine. Another possible candidate for measurement is the *width* of a (layered) branching program, which equals the maximum number of nodes in a layer.

Note: For a language L, its branching program complexity $BP_L(n)$ can be considerably smaller than its uniform space complexity $s_L(n)$.

3.3 Uniform Models with Advice

The two models discussed above use a different machine for each input length, resulting in an infinite family of machines. Our third model is similar to uniform models in that it uses a single machine for all input lengths. We give extra power (nonuniformity) to the uniform machine by allowing access to *advice*, a piece of additional information which may help improve the efficiency of computation.

Definition 2. *Let $a(n)$ be a function from $\mathbb{N}$ to $\mathbb{N}$. Let $\mathcal{C}$ be a class of languages. We define the class $\mathcal{C}/a(n) = \{L|$ there exists $L' \in \mathcal{C}$ and $y_0, y_1, y_2, \ldots \in \Sigma^*$ such that $x \in L \iff \langle x, y_{|x|} \rangle \in L'$, where $|y_n| \leq a(n)$ for $n = 0, 1, 2, \ldots\}$.*

The sequence $\{y_0, y_1, y_2, \ldots\}$ in the above definition is called the *advice sequence* and $a(n)$ is an upper bound for the length of advice. Note that the advice taken by the machine only depends on the length of the input. This differs from *certificates* where each input may have a different certificate and there is no restriction on certificates for inputs of equal length.

Advice can be very powerful, even when its length is bounded by one bit. In fact, there exist uncomputable languages which can be computed using one-bit advice. As an example we consider the halting problem HALT $= \{x \,|\, x$ encodes a DTM M and M halts on input $x\} \subseteq \{0,1\}^*$. HALT is known to be uncomputable (by any Turing machine). Consider the language $L' = \{0^z \,|\, z$ is the lexicographic rank of x and $x \in$ HALT $\} \subseteq \{0\}^*$. It is clear that HALT$\leq_m L'$ and hence L' is also uncomputable. If we take the advice sequence $\{y_0, y_1, y_2, \ldots\}$ to be the characteristic function of HALT, then L' can be computed by a machine that simply outputs the advice bit y_z on input 0^z.

4 Connections Between Uniform and Nonuniform Models

In this section, we present a few results relating nonuniform models to uniform complexity classes.

P/poly is the class of all languages which can be computed in polynomial time using polynomial-length advice. Similarly, L/poly is the class of all languages computable in logarithmic space using polynomial-length advice. The next two theorems show relationships between nonuniform models and these complexity classes.

Theorem 3. P/poly $= \{L | C_L(n)$ *is polynomially bounded* $\}$.

Proof Sketch. $\subseteq$: We can construct polynomial-size circuits accepting a language in P/poly by hardwiring the polynomial-length advice strings as "additional inputs". $\supseteq$: If L has polynomial-size circuits, we can use the descriptions of the circuits as advices. □

The proof of the following theorem is similar and is left as an exercise.

Theorem 4. L/poly $= \{L | BP_L(n)$ *is polynomially bounded* $\}$.

We may also consider uniform boolean circuits. A family of circuits $\{B_i\}$ is *uniform* if there exists a uniform machine which, given n, outputs the description of B_n in time polynomial in n. Uniform branching programs are defined similarly, except that the notion of uniformity here is not standardized and may differ by context. The following two theorems say that uniform circuits and branching programs are indeed uniform.

Theorem 5. P $= \{L \mid L$ *has uniform polynomial-size circuits* $\}$.

Proof Sketch. $\subseteq$: Let $L \in$ P and M be a DTM accepting L. We can construct a circuit for inputs of length n by hardwiring valid transitions and constraints in the computation tableau of M. Since M runs in polynomial time, its computation tableau has polynomial size. The resulting circuit also has polynomial size and can be computed in polynomial time. $\supseteq$: On input x, the uniform machine computes $B_{|x|}$ in polynomial time, then simulates the computation of $B_{|x|}$ on x also in polynomial time. □

Theorem 6. L $= \{L \mid L$ *has uniform polynomial-size branching programs* $\}$.

5 Next Time

Consider the class P/poly of problems which are solvable in polynomial time given some advice with polynomial length. If for some NP-complete problems, we happened to have computed the advice strings as well as polynomial-time algorithms making use of them, then we could solve these NP-complete problems efficiently. Next lecture we will show that this is unlikely to be the case; in particular, we will prove that if NP $\subseteq$ P/poly, then the polynomial-time hierarchy collapses to the second level. Then we will move on to talk about constant-depth boolean circuits.

CS 810: Complexity Theory 2/7/2007

Lecture 8: Constant-Depth Circuits

Instructor: Dieter van Melkebeek Scribe: Seeun William Umboh

DRAFT

In the last lecture we applied alternations to prove some time-space lower bound results for SAT. We also introduced the notion of nonuniform computation, and nonuniform models such as Boolean circuits, branching programs and uniform machines with advice.

Today, we begin with a theorem that suggests that SAT does not have small circuits. Then, we investigate constant-depth circuits.

1 If NP has small circuits, then PH collapses

Theorem 1. *If* $\mathrm{NP} \subseteq \mathrm{P/poly}$*, then* $\mathrm{PH} = \Sigma_2^p$.

Note that we only need to show that we can simulate a Π_2^p computation with a Σ_2^p computation: since $\Pi_2^p = \mathrm{co}\Sigma_2^p$, if $\Pi_2^p \subseteq \Sigma_2^p$ then $\Pi_2^p = \Sigma_2^p$.

Proof. For a language L in Π_2^p, we have

$$x \in L \iff (\forall y \in \Sigma^{|x|^c})(\exists z \in \Sigma^{|x|^c}) R(x, y, z)$$

where R is a predicate that can be decided deterministically in time, say, linear in its combined input.

Since $(\exists z \in \Sigma^{|x|^c}) R(x, y, z)$ is a Σ_1^p predicate on input $< x, y >$, we can reduce it to a SAT instance and by the hypothesis, there exists a circuit C_{SAT} that is of size polynomial in the running time to decide R, and so of size polynomial in the size of x. Letting f denote the reduction, we can replace the Σ_1^p predicate with $C_{SAT}(f(x, y))$. We would like to rephrase the formula above roughly as follows: does there exist a circuit solving SAT such that for all strings y, the circuit accepts $f(x, y)$[1]?

We can transform the right-hand side of the above to:

$$(\exists C_{SAT} \textit{ with input size } n^c)(\forall y \in \Sigma^{|x|^c})[C_{SAT}(f(x, y)) \wedge (\forall \varphi \textit{ of size} \leq n^c)[V(C_{SAT}, \varphi)]]$$

where V is the following recursive predicate:

(1) **if** φ has at least 1 variable **then** $C_{SAT}(\varphi) \iff C_{SAT}(\varphi|_{x_1 \leftarrow 0}) \vee C_{SAT}(\varphi|_{x_1 \leftarrow 1})$

(2) **else** $C_{SAT}(\varphi) \iff \varphi$ *is true*

where x_1 is the first unset variable of φ.

[1] One slight detail here: the circuit C_{SAT} takes inputs of fixed input size only, but $f(x, y)$ is of varying size. However, we can simply pad $f(x, y)$ to an equivalent instance of the required size.

Essentially, we guess a circuit C_{SAT} and the combined predicate checks if C_{SAT} accepts $f(x, y)$ and whether or not C_{SAT} is a valid circuit solving SAT. Since we are evaluating polynomial size circuits, and we evaluate n times, V takes polynomial time to check. The second universal quantifier above can be merged with the first, and then we have a single polynomial time verifiable predicate. So, we now have a Σ_2^p formula. Note that our hypothesis is crucial in that if NP does not have polynomial size circuits, then V will always fail. □

Since we do not believe that the polynomial-time hierarchy collapses, this is taken to be evidence suggesting that NP does not have polynomial-size circuits.

The proofs of the following are similar, so we leave them as exercises.

Exercise 1. *If* $PSPACE \subseteq \mathrm{P/poly}$ *then* $PSPACE = \Sigma_2^p$.

Exercise 2. *If* $EXP \subseteq \mathrm{P/poly}$ *then* $EXP = \Sigma_2^p$.

2 Nonuniform Lower Bounds for NP

In the previous lecture, we stated that one application of nonuniform models of computation is in attempts to prove lower bounds for computing certain functions (in particular NP-complete problems). In fact, there has been little progress in proving lower bounds for NP-complete problems. We survey results in this section. Nontrivial lower bounds have been proven for restricted models of computation - these are discussed in the next section.

2.1 Boolean Circuits

We only know the following facts in this area:

Theorem 2. $C(f) = O(\frac{2^n}{n})$ *for any Boolean function* $f : \{0,1\}^n \to \{0,1\}$

Using the naive encoding of the truth table into a DNF, we can get a $O(n2^n)$-size circuit. A better analysis gives the better bound.

Theorem 3. $C(f) = \Omega(\frac{2^n}{n})$ *for most Boolean functions* f*. That is, if we pick* f *uniformly at random from the set of Boolean functions on* n *variables, the probability that* $C(f) = \Omega(\frac{2^n}{n})$ *converges to 1 as* n *grows.*

Proof. Let s denote the number of binary gates. For each of the s gates, we can pick a variable or some other gate as input, and each gate has at most 2 inputs. So, the number of circuits of size at most s is at most $(c(s+n)^2)^s$, where c is some constant that also takes care of the possibility that the input is negated. Since we can map circuits to Boolean functions, this is also the maximum number of Boolean functions computable with at most s gates. We know that 2^{2^n} is the number of Boolean functions on n variables, and $(c(s+n)^2)^s = 2^{O(s \log s)}$. By setting $s = d\frac{2^n}{n}$ for a sufficiently small constant d, $(c(s+n)^2)^s = 2^{O(s \log s)} = 2^{O(d\frac{2^n}{n} \cdot (n \log d - \log n))} << 2^{2^n}$. The claim then follows. □

This shows that most Boolean functions require circuits of the maximum circuit size, up to a constant factor. So we would expect that at least for complicated functions like those that capture NP-complete problems, we can prove non-trivial lower bounds. However, that is not the case. The best known lower bound for any $L \in \mathrm{NP}$ is:

Proposition 1. $\exists L \in \mathrm{NP}$ *with* $C_L(n) = \Omega(n)$

2.2 Branching Programs

For branching programs, we can show that $BP(f) = \Theta(\frac{2^n}{n})$ for most functions just as in the case of Boolean circuits, and the best known lower bound is roughly quadratic. Even though the lower bound is better than in the case for Boolean circuits, it is still a rather trivial result. Recall from the previous lecture that the corresponding Turing machine uses $O(\log v)$ space where v is the size of the branching program.

We can hope to prove a lower bound on branching program size by restricting our attention to layered branching programs of constant width. However, these are more powerful than they might seem at first sight, and we cannot prove substantially better lower bounds than for arbitrary branching programs.

3 Constant-Depth Circuits

Because we have been unable to prove lower bounds for NP-complete problems in the general setting, we focus our attention on a restricted model - namely constant-depth circuits. We first give some basic facts about constant-depth circuits, and then prove that they require exponential size to even compute the parity function.

Definition 1 (constant-depth circuit)**.** *A constant-depth circuit is a Boolean circuit with unbounded fan-in but whose depth is bounded by a constant.*

This model might seem too restricted, but we have already mentioned that any function can be computed by a depth 2 CNF or DNF. However, such a circuit is in general of exponential size, and we would like to know if we can do better. We will see that even for the PARITY function, the size required is exponential.

We have encountered constant-depth circuits before in the lecture on alternation, and we showed that we can simulate alternation with constant-depth circuits of exponential fan-in. Today, we will look at such circuits that are of polynomial size. In particular, we look at the following family of classes:

Definition 2 (AC^k)**.** *$AC^k = \{L | C_{O(\log^k n)}(L_n)$ is polynomial$\}$, where $C_{O(\log^k n)}(L_n)$ denotes the complexity of circuits with unbounded fan-in, depth $O(\log^k n)$, and deciding the restriction of L to length n.*

For now, we are interested in AC^0, the class of languages decidable by constant-depth circuits of polynomial size. Let us now look at examples of languages in and not in AC^0.

Proposition 2. *The decision variant of binary addition is in AC^0.*

Proof. In this proof, all strings are indexed from the right. To determine the ith bit of the sum, we only need to look at the ith bits of the summands and determine if there is a carry from the bits in position $(i-1)$. We first introduce some notation. We will label each column from 1 up to $i-1$ depending on if it either: *g*enerates a carry bit, *t*ransmits a carry bit, or *s*tops a carry bit. If both input bits in the column are 1, the column is labeled g; if only one bit is 1, it is labeled t; and if both bits are 0, the column is labeled s.

For there to be a carry from the $(i-1)^{st}$ column, there must be a g at some point followed by zero or more t columns. In other words, to determine if there is a carry into the ith position, our

job is reduced to detecting if the above string is of the form $t^*g\{s,g,t\}^*$. First of all, we use an OR to guess the length of t^*g, and the number of possible lengths are at most linear in the input size. For each length j, we need an AND to determine if the $(i-j+1)$th symbol on the string is a g, and XORs to ensure that the symbols after it are ts, and then we do a big AND over the XORs and the AND. So, at the first level, we have an OR, at the second we have ANDs, and at the third we have ANDs and XORs. Since XORs can be implemented in constant depth using ANDs, ORs and NOTs, the overall circuit has constant depth. □

We will discuss problems that can be computed in various AC^k in a future lecture. We now sketch a proof that PARITY requires exponential size to be computed by constant-depth circuits.

Definition 3. *PARITY* $= \{x : x$ *has an odd number of 1s*$\}$

We also denote PARITY on n variables as $\bigoplus_n$.

Theorem 4. *PARITY is not in* AC^0.

By proving this result, we will have also shown that PARITY cannot be computed by polynomial size circuits with bounded fan-in and log-depth. This follows from Theorem 4 by using a divide-and-conquer strategy.

In fact, the proof we outline today shows that $C_d(\bigoplus_n) = 2^{\Omega(n^{\frac{1}{d-1}})}$. As PARITY can be computed by circuits of size $2^{O(n^{\frac{1}{d-1}})}$, this gives an exact characterization of the size of constant-depth circuits required to compute parity.

The proof we present today uses the following tool.

Definition 4 (Random Restrictions)**.** *A p-random restriction on n variables is a random function $\rho : \{x_1, \ldots, x_n\} \to \{*, 0, 1\}$, such that for each i, independently, $Pr[\rho(x_i) = *] = p$ and $Pr[\rho(x_i) = 1] = \frac{1-p}{2} = Pr[\rho(x_i) = 0]$. If $\rho(x_i) = *$ then we leave x_i as a variable. Otherwise we set it to the result of $\rho(x_i)$.*

Note that if we apply a random restriction to a parity function, we get a parity function or its complement on those bits set to $*$.

Proof sketch of Theorem 4. Let us start with some AC^0 circuit C. WLOG, we assume that for each level of C, there are only ANDs or ORs, and that the circuit alternates between these. We also assume that the inputs to the circuit are the variables and their negations, allowing us to ignore NOT gates for the most part.

The main ingredients of the proof are:

Proposition 3. $C_2(\bigoplus_n) = \Theta(n2^{n-1})$.

Proof. For depth 2, we can easily prove an exponential lower bound. By assumption, the circuit is either a DNF or a CNF. Let us assume that it is a DNF. Each of the AND terms check for a setting of variables such that $\bigoplus_n = 1$. Thus they must contain all n variables. Otherwise, we can flip a variable and at least one AND will not be able to detect the difference. There are 2^n possible n-variable AND terms in a DNF formula, and we only need half of them as only half of them check for $\bigoplus_n = 1$. Since each AND must be of size n, and there must be 2^{n-1} of them, we get the bound as stated. □

Lemma 1 (Switching Lemma)**.** *Given a CNF with small bottom fan-in. Then we can apply a random restriction that does not set to many variables so that with high probability the resulting function can be written as a DNF with small bottom fan-in.*

Note that the statement is trivial if the restriction can set all variables – the "not setting too many variables" is important. Also, all of the qualifications like "not too many" and "small" need to be quantified appropriately for the statement to hold but we keep the exposition of this approach at a qualitative level.

Proof Idea. Consider an AND of ORs, where each OR has size at most k. Notice that a random restriction is not very likely to set an OR to 0 since all literals involved need to be set to 0 for that to happen. But if k is small, there is a nontrivial probability that this happens. There are two cases:

1. There are a large number of pairwise disjoint ORs. In that case, there are many independent events that can set the AND gate to 0, namely each of those pairwise disjoint ORs being set to 0. Since each of those events happens with a nontrivial probability, the odds are that the random restriction will set the AND gate to 0, in which case it can trivially be written as a DNF will small bottom fan-in.

2. There is not a large number of pairwise disjoint ORs. Let V be a minimal set of variables such that each OR queries at least one variable from V. Since there is a lot of overlap among the ORs, V is small. If we query all the variables in V, then we have essentially reduced our problem to a simpler one of the same type, namely the transformation of a CNF with bottom fan-in at most $k-1$. This is because each of the ORs contains at least one literal in V. We then repeat the case distinction to that simpler problem, depending on the setting of the variables in V.

Along every branch of this process, we will eventually end up in case 1. Since there are at most k steps and each step involves querying a small number of variables, we end up with a decision tree of small depth that represents the given CNF under a random restriction with high probability. A decision tree of small depth can be turned into a DNF with small bottom fan-in by writing down an OR over all paths in the decision tree that lead to acceptance of the AND of all the conditions that define the path. □

Note that we can also switch from a DNF to a CNF by considering the negation of the circuit.

We use the Switching Lemma to reduce the depth of the circuit by 1 at a time until we are left with a circuit of depth 2. Suppose that the bottom gates are ANDs. To apply the switching lemma, we need to ensure the gates at the bottom of the circuit have small fan-in. To ensure this, we insert dummy OR gates below the AND gates. Namely, for each input x to the AND gate, we replace that with x OR x. Now, we apply the switching lemma to the AND of ORs we have created. With high probability, each application is successful in creating an OR of ANDs with small bottom fan-in and without setting too many variables. Now the second bottom-most and third bottom-most levels are both ORs and can be merged. This reduces the depth of the circuit by 1 (back down to d since we added a level initially).

Now the circuit still has small bottom fan-in, so we can apply the switching lemma again. We repeat this process until we get a circuit C' of depth 2. At this point, if C computed $\bigoplus_n$, then C'

computes $\bigoplus_m$ on some m-subset of the variables (those that were unset by the random restrictions). At this point we use Proposition 3 to derive a lower bound on the size of the remaining circuit (and thus also of the original circuit). If m is still relatively large, this gives an exponential lower bound on the size of the original circuit. Further, there must be a positive probability that each application of the switching lemma was indeed successful. □

Next lecture, we give an alternate proof that PARITY requires exponential size constant-depth circuits. This proof will use low-degree polynomial approximations rather than random restrictions. The bound that we will get, $C_d(\bigoplus_n) = 2^{\Omega(n^{\frac{1}{2d}})}$ is not as tight as the previous result, but the advantage is that it applies to circuits with gates other than AND, OR, NOT.

CS 810: Complexity Theory 2/9/2007

Lecture 9: Polynomial Approximations

Instructor: Dieter van Melkebeek Scribe: Piramanayagam Arumuga Nainar

Last time, we proved that no constant depth circuit can evaluate the parity function. We used random restrictions to obtain a bound on the complexity of a circuit evaluating the parity of n inputs. In this lecture, we complete give an alternative proof that gives a slightly weaker bound.

Using the random restriction method, we showed that $C_d(\oplus_n) \geq 2^{\Omega(n^{\frac{1}{d-1}})}$. This is a tight bound and uses the property that the parity function is sensitive to every bit of its input. We can also derive a similar bound for the mod_m function defined as follows:

$$\text{mod}_m(x) = \begin{cases} 0 & \text{if } |x| = 0 \bmod m \\ 1 & \text{otherwise} \end{cases} \tag{1}$$

where $|x|$ is the number of non-zero bits in the input. Parity is a special case of mod_m at $m = 2$. Another function that we can prove is not in AC^0 is the majority function: that returns whether bit 0 or bit 1 occurs the maximum number of times in its input. This can be proved by using the parity function as a black box and is left as an exercise.

This lecture, we use low degree polynomial approximations to show that $C_d(\oplus_n) \geq 2^{\Omega(n^{\frac{1}{2d}})}$. Even though this is a weaker bound, the technique itself is interesting. Moreover this result applies even if we allow mod_3 gates in the circuit. Finally, we also prove that constant depth circuits are not enough to even approximately evaluate parity.

1 Polynomial approximation method

Theorem 1. *$\oplus_n \notin AC^0$. Specifically, $C_d(\oplus_n) \geq 2^{\Omega(n^{\frac{1}{2d}})}$*

Proof outline: We prove this in two steps. First, we show that any constant depth circuit can be approximated using a low degree multi-variate polynomial over the field $\mathbb{Z}_3$. (Using $\mathbb{Z}_3$ gives, for free, the ability to mimic mod_3 gates. In general, if we use $\mathbb{Z}_p$ for a prime number p, we can handle mod_p gates). Second, we show that the parity function cannot be approximated using a multi-variate polynomial of a sufficiently low degree over the field $\mathbb{Z}_3$.

Proof. Step 1: Consider a circuit C made of AND, OR, NOT and mod_3 gates. It can always be represented as a multi-variate polynomial of degree n where n is the size of the input. Our goal is to represent it using a polynomial of lower degree, allowing errors if required. A literal x that is directly passed as input to a gate can be represented using the polynomial x. This is the base case of our construction. Now, we can assume that a polynomial P_i can be associated with the i^{th} input of other gate types (AND, OR, NOT, mod_3). The goal is to construct a polynomial P' that represents the output of the gate. Note that the number of inputs to any gate is at most $|C|$.

If P' is the polynomial representing the input of a NOT gate, then $1 - P'$ represents the output of the NOT gate. Notice that the representation of a NOT gate does not increase the degree of the polynomial.

Consider a mod_3 gate. The output of the gate is zero when $\sum_{i=1}^{m} P_i = 0$. If the summation is 1 or 2, the output is 1. Note that in the field $\mathbb{Z}_3$, $2 \cdot 2 = 1$. So, we can model the gate using the polynomial $P' = (\sum_{i=1}^{m} P_i)^2$. P' accurately models the gate and its degree is at most twice the degree of any of its inputs.

Consider an OR gate. The output of the OR gate is 0 if $\forall i, P_i = 0$ or, in other words, $\forall i, (1 - P_i) = 1$. Otherwise its output is zero. This can be represented as follows:

$$\alpha : P' = 1 - \prod_{i=1}^{m} (1 - P_i) \tag{2}$$

This formulation is accurate but the degree of P' may be up to m times the degree of the P_i with the largest degree. This can be much higher than the trivial bound n if there are many gates and many levels in the circuit. To tackle this, we model P' as a linear combination of P_i for $1 \leq i \leq n$. Let r_i be the coefficient associated with P_i. As with mod_m, we square the linear combination to keep the value of P' boolean. This leaves us with:

$$\beta : P' = (\sum_{i=1}^{m} r_i \cdot P_i)^2 \tag{3}$$

This makes the degree of P' at most twice that of the degree of its inputs. But it is definitely not an accurate description of an OR gate. Suppose we pick the coefficients (r_i for all i) at random and evaluate the probability of P_i being different from the boolean expression $\vee_{i=1}^{n} P_i$. If $P_i = 0$ for all i, then irrespective of the values picked for the coefficients, the output is correct. If $P_i = 1$ for at least one i, $\sum_{i=1}^{m} r_i \cdot P_i$ is $\sum_{i|P_i=1} r_i$. This is the wrong value, 0, in one out of three cases for a random assignment of the coefficients. Thus, P' can introduce errors in the representation with a probability at most $\frac{1}{3}$. As with any randomized algorithm, we can repeat the above calculation for, say, t independent trials and see if the output of at least one of the trials is one. (Note: An output of one will always be correct but an output of zero may be wrong). This leads us to the third, and final, formulation of P'.

$$P' = P'_{\alpha}(P'_{\beta_1}(\hat{P}), \ldots P'_{\beta_t}(\hat{P})) \tag{4}$$

Here, P'_{α} is the application of P' as described in eqn. 2 on t inputs. P'_{β_k} is the k^{th} trial using the formulation of P' in eqn. 3. $\hat{P}$ is a shorthand for $P_1, P_2, \ldots P_m$. The above formulation produces a wrong output if all the trials produce the wrong output, i.e. with probability at most $\frac{1}{3^t}$. The degree of P' increases by a factor of $2t$: a factor t for the α-formulation and a factor of 2 for the β-formulation.

We can handle an AND gate in a similar way, resulting in an approximation P' with at most a factor of $2t$ blow-up in the degree, and giving an imprecise value with probability at most $\frac{1}{3^t}$.

If the depth of the circuit is d, the degree of the polynomial P representing the entire circuit will be at most $(2t)^d$. P gives the wrong value only if the output of at least one of the gates in C was wrong. This happens with probability at most $\frac{|C|}{3^t}$. (Note: This is not a very tight upper bound but it is enough for this proof.) By averaging, the expected number of inputs for which P will

give the wrong value is at most $\frac{|C|}{3^t}2^n$ since there are 2^n possible inputs of length n. There exists a choice for the random coefficients for which P is wrong in no more than the expected number, derived above. More formally,

Lemma 1. *There exists a choice of r_i's such that there exists a set $G \subseteq \{0,1\}^n$ of relative size $\mu(G) \geq 1 - \frac{|C|}{3^t}$ such that $P(x) = C(x) \forall x \in G$ where P is a polynomial of degree at most $(2t)^d$ constructed as described above.*

Here, $\mu(G)$ is the relative size of G with respect to the set of all possible inputs to C and is equal to $\frac{|G|}{2^n}$. This construction can be generalized to work over any field $\mathbb{Z}_p$ for prime p, thus allowing $\mod p$ gates. The property of $\mathbb{Z}_3$ we used is that $a^2 \equiv 1 \pmod 3$ for all $a \neq 0 \pmod 3$. Thus, squaring a polynomial ensures boolean values. To work over $\mathbb{Z}_p$, we would instead raise polynomials to the power $p-1$ as $a^{p-1} \equiv 1 \pmod p$ for all $a \neq 0 \pmod p$. The degree of the resulting polynomial is at most $(p \cdot t)^d$ rather than $(2t)^d$.

Step 2: In this step, given a polynomial P of some degree that approximates $\oplus_n$ on a subset G of inputs, we establish an upper bound below which every function of n inputs has a corresponding polynomial approximating it over G. By equating the number of such functions to the number of polynomials with degrees not greater than the established upper bound, we derive the lower bound on the depth of circuit C.

As a first step, we transform the inputs to a slightly more convenient domain: $\{-1,1\}$ instead of $\{0,1\}$.

Proposition 1. *Suppose there exists a polynomial P of degree at most Δ that computes $\oplus_n$ on a set $G \subseteq \{0,1\}^n$. Then there exists a polynomial P' of degree at most Δ and a set $G' \subseteq \{-1,1\}^n$ such that $\mu(G') = \mu(G)$ and $(\forall x \in G') \prod_{i=1}^{n} x_i = P'(x)$.*

The reason is that parity on boolean inputs is equivalent to multiplication over $\{-1,1\}$.

Lemma 2. *Suppose there exists a polynomial P' of degree at most Δ that represents multiplication in a set $G' \subseteq \{-1,1\}^n$. Then each function $f : G' \to \mathbb{Z}_3$ has a multi-variate polynomial Q over $\mathbb{Z}_3$ of degree at most $\frac{n+\Delta}{2}$ such that it represents f, i.e. $(\forall x \in G') f(x) = Q(x)$.*

Proof. Every function f has a multi-variate polynomial of degree at most n. This is trivial because we can hardwire every possible input using monomials of degree n. Let us start from one such polynomial Q' (such that $f = Q'$ on G'). Consider a monomial in Q' of the form $\prod_{i \in I} x_i$ where I is a subset of the input bits. Because we are only concerned with +/-1 inputs, we can rewrite it as:

$$\begin{aligned} \prod_{i \in I} x_i &= (\prod_{i \in \bar{I}} x_i^2)(\prod_{i \in I} x_i) \\ &= (\prod_{i \in \bar{I}} x_i)(\prod_{i=1}^{n} x_i) \qquad (5) \\ \implies \prod_{i \in I} x_i &= (\prod_{i \in \bar{I}} x_i) P'(x) \qquad (6) \end{aligned}$$

Eqn. 5 holds for any input x of n bits but eqn. 6 holds only for the inputs in the set G'. The LHS of 6 has degree $|I|$. The RHS has a degree at most $\Delta + |\bar{I}| = n + \Delta - |I|$. At least one of the degrees is smaller than or equal to $\frac{n+\Delta}{2}$. Thus, we can make the degree of every monomial in Q' to not exceed $\frac{n+\Delta}{2}$. □

Given the lemmas, we are now ready to combine them in the appropriate way to prove the theorem. Suppose there exists a circuit C of depth d computing $\oplus_n$. From Lemma 1, there exists a polynomial P' of degree at most $\Delta = (2t)^d$ that computes parity on a set G of relative size at least $1 - \frac{|C|}{3^t}$. Consequently, from Lemma 2, all functions $f : G' \to \mathbb{Z}_3$ for some G' such that $|G| = |G'|$ can be represented using a multivariate polynomial of degree at most $\frac{n+\Delta}{2}$. The total number of such polynomials must be at least the number of functions f from G' to $\mathbb{Z}_3$.

The number of multivariate polynomials with degree at most $\frac{n+\Delta}{2}$ is exactly 3^M where M is the number of monomials of degree at most $\frac{n+\Delta}{2}$. There are $\binom{n}{i}$ monomials of degree i. So $M = \sum_{i}^{\frac{n+\Delta}{2}} \binom{n}{i}$. The number of monomials of degree $\leq \frac{n}{2}$ will be 2^{n-1} - half of the 2^n possible monomials. The remaining $\frac{\Delta}{2} = \Theta(\Delta)$ terms in the summation will be lower than $\binom{n}{\frac{n}{2}}$ - the maximum possible number for any degree. Using Stirling's approximation, we can show that:

$$\binom{n}{\frac{n}{2}} = \Theta\left(\frac{2^n}{\sqrt{n}}\right)$$

Thus, $M = 2^{n-1} + \Theta\left(\frac{\Delta}{\sqrt{n}}\right) 2^n = \left(\frac{1}{2} + \Theta\left(\frac{\Delta}{\sqrt{n}}\right)\right) 2^n$.

The number of functions of the form $G' \to \mathbb{Z}_3$ is $3^{|G'|}$ as one of 3 possible values can be assigned to each element of G'. Because the number of functions of this form must be at most the number of polynomials of degree at most $(n+\Delta)/2$, $3^{|G'|} \leq 3^M$ or, in other words, $|G'| \leq M$. This gives us the following bound on the size of G'.

$$\mu(G') = \frac{|G'|}{2^n} \leq \frac{M}{2^n} \leq \frac{1}{2} + \Theta\left(\frac{\Delta}{\sqrt{n}}\right)$$

From Lemma 1, $\mu(G') \geq 1 - \frac{|C|}{3^t}$ when $\Delta = (2t)^d$. Thus,

$$\begin{aligned} 1 - \frac{|C|}{3^t} &\leq \mu(G') \leq \frac{1}{2} + \Theta\left(\frac{(2t)^d}{\sqrt{n}}\right) \\ \implies |C| &\geq 3^t \left[\frac{1}{2} - \Theta\left(\frac{(2t)^d}{\sqrt{n}}\right)\right] \end{aligned}$$

Setting $(2t)^d = O(\sqrt{n})$ gives a tight value for the RHS in the last equation. Thus, $t = \Theta(n^{\frac{1}{2d}})$. This gives $|C| \geq 2^{\Omega(n^{\frac{1}{2d}})}$. □

The only part of the above analysis that changes when working over $\mathbb{Z}_p$ rather than $\mathbb{Z}_3$ is that $\Delta = (p \cdot t)^d$ rather than $(2t)^d$. Thus the result holds with the same lower bound on $|C|$ for boolean circuits with $\mod_p$ gates for any prime p. In fact, the argument in the above proof can be generalized to give a lower bound for circuits with $\mod_p$ gates to compute $\mod_q$ (recall that parity is the special case of $q = 2$). This is achieved by viewing Step 2 as harmonic analysis over

$\mathbb{Z}_2$ and then generalizing that to harmonic analysis over $\mathbb{Z}_q$. As this generalization takes a bit of work to prove, we leave it at that.

Because the lower bound for parity was proved by viewing parity as multiplication, we get a lower bound for multiplication as well.

Corollary 1. *The decision variant of binary multiplication is not in AC^0.*

We further use the proof above to give a lower bound on circuits that even approximate parity.

Corollary 2. *A depth d unbounded fan-in circuit that agrees with parity on a fraction at least $\frac{1}{2} + \frac{1}{n^{(1-\epsilon)/2}}$ of $\{0,1\}^n$ must have size $2^{\Omega(n^{\epsilon/2d})}$.*

Proof. Suppose we have a circuit that is correct on at least $\frac{1}{2} + \rho$ of the inputs. Similar to Theorem 1, we can prove that there exists a polynomial of degree $\Delta = (2t)^d$ that is correct on a set G' that is at least $\frac{1}{2} + \rho - \frac{|C|}{3^t}$ of $\{0,1\}^n$. From *Step 2* of the proof above,

$$\frac{1}{2} + \rho - \frac{|C|}{3^t} \leq \frac{1}{2} + \Theta\left(\frac{(2t)^d}{\sqrt{n}}\right) \implies \rho - \frac{|C|}{3^t} \leq \Theta\left(\frac{\Delta}{\sqrt{n}}\right) \tag{7}$$

$$\implies |C| \geq 3^t \left[\rho - \Theta\left(\frac{(2t)^d}{\sqrt{n}}\right)\right] \tag{8}$$

Note that the $(2t)^d/\sqrt{n}$ term is $\Omega(1/\sqrt{n})$, so ρ must also be $\Omega(1/\sqrt{n})$ to ensure the lower bound we get is even positive. If we let $\rho = 1/n^{(1-\epsilon)/2}$, we set $(2t)^d = \Theta(n^{\epsilon/2})$ to optimize the RHS of 8. So $t = \Theta(n^{\epsilon/(2d)})$, and we get that $|C| \geq 2^{\Omega(n^{\epsilon/(2d)})}$. □

The above corollary proves the inapproximability of the parity function using constant depth circuits. There is another such result that can be proved using random restrictions. It is as follows:

Theorem 2. *A depth d unbounded fan-in circuit that agrees with parity on a fraction at least $\frac{1}{2} + \frac{1}{2^{\Omega(n^{1/d})}}$ of $\{0,1\}^n$ must have size $2^{\Omega(n^{1/d})}$.*

This is interesting because even trivial functions can guess parity correctly on half of the inputs. This is slightly weaker than the $2^{\Omega(n^{\frac{1}{d-1}})}$ bound we derived last lecture but it disproves approximability rather than computability of the parity function. We will see more such results of inapproximability when we discuss pseudo-randomness.

2 Next lecture

Next lecture, we will discuss parallelism where we distribute the computational task among multiple processors to reduce the time complexity.

CS 810: Complexity Theory 2/12/2007

Lecture 10: Parallelism

Instructor: Dieter van Melkebeek Scribe: Matt Elder

The goal of parallelism is to speed up computation by dividing it among many processors. In this lecture, we discuss models for parallelism and complexity classes that capture efficiently parallel-computable problems.

1 Conceptual Model

We want our model of parallelism to be of roughly the same power as large collections of Turing machines, all acting together through some means of communication.

Our model must capture this means of communication. What form the interconnections between processors? In our model, we will blithely assume that connections are free. This is not realistic; in real parallel computers the interconnection network is one of the most confusing pieces. However, there are several network configurations, like butterfly nets and hypercubes, that grow at reasonable rates and yield communication between any two processors in $\log p$ time, where p is the number of processors.

Our model must also impose some limits on the number of processors, and here our model must diverge somewhat from physically realizable computers. If we allow only a constant number of processors, then we can give only constant speedup over a standard Turing machine for any problem. Thus, we must allow the number of processors to grow with the input size. This will entail issues of uniformity.

Our criteria for efficiency change when we move from standard Turing machines to vastly parallel computers; we will now aim for polylog time instead of polynomial time. To achieve this, we will permit a number of processors polynomial in the size of the input.

2 Concrete Model

We model parallel computation with *Uniform NC-Circuits*. The class NC is very similar to AC and is defined as follows:

Definition 1. NC^k *is the set of all languages recognizable by circuits with bounded fanin, polynomial size, and* $O(\log^k n)$ *depth. NC is the union of all classes* NC^k*, that is,* $\mathrm{NC} = \cup_{k \geq 0} \mathrm{NC}^k$.

An NC Circuit is uniform *if the circuit can be computed from the size of its input in logarithmic space. The uniformity condition is considered standard: unless otherwise specified, "NC" typically means "Uniform NC."*

Thesis 1 L has an efficient parallel algorithm iff L is in Uniform NC.

We leave it to the reader to verify that the complexity class Uniform NC^k is closed under composition and log–space mapping reductions.

3 Complexity of NC

Next we place the class L among the classes in the NC^k hierarchy.

Theorem 1. *Uniform $NC^1 \subseteq \mathrm{L} \subseteq$ Uniform NC^2.*

Proof. First, we show that Uniform $\mathrm{NC}^1 \subseteq L$. Suppose that a uniform family F of NC^1-circuits decides language A. Then, given an input x, we can simulate F in logarithmic space as follows:

1. Compute the circuit C appropriate for $|x|$. More precisely, compute each bit of the description of C as it is needed. We do not have enough space to store the entire description of the circuit, but we can compute each part as we need it in logarithmic space because F is uniform.
2. From the output node of C, compute the values of each gate in C recursively, without memoization. This is painfully slow, but we wish to optimize for space. Since the depth of C is in $O(\log |x|)$, we can do this computation in logarithmic space.
3. If the output of C is 1, accept. Else, reject.

The fact that $L \subseteq$ Uniform NC^2 is left as an exercise. □

We can also relate every class in the NC hierarchy to classes in the closely related AC hierarchy, as follows:

Theorem 2. $\mathrm{NC}^k \subseteq \mathrm{AC}^k \subseteq \mathrm{NC}^{k+1}$.

Proof. Since NC^k is a restriction of AC^k, the first inclusion is clear. The second inclusion is implied by the fact that polynomially-bounded fanin can be simulated by a simple logarithmic-depth circuit of bounded fanin. □

4 Languages in Various NC^k

To give a feel for the NC hierarchy, we see how efficiently some basic tasks can be accomplished in parallel.

4.1 NC^0

The class NC^0 contains, by definition, only those languages decidable by constant-depth, constant-fanin circuits. This implies, among other things, that these problems must be decidable by checking only a constant number of bits of the input.

4.2 NC^1

By Theorem 2, this class contains AC^0, and thus contains, for example, binary addition. This class also contains iterated addition.

Theorem 3. *Iterated addition is in* NC^1.

```
  10010101
  01111011
+ 00111101
----------
  11010011  mod 2
+ 001111010 carries
```

Figure 1: Finding two numbers with the same sum as three different numbers.

Proof. Given three binary numbers, we can output two numbers with the same sum using a constant depth circuit. Each triple of input bits contains 0 to 3 ones; thus we produce one binary number containing the value of these additions modulo 2, and another binary number containing the carries. For example, see Figure 1.

This operation is possible with constant-depth circuits. So, we group all of our inputs into groups of 3, apply this operation, and repeat until there remain only two inputs. This operation reduces the number of remaining numbers to add by 1/3, so some logarithmic number of layers of these circuits reduces the problem to binary addition, which is in NC^1, as we have already seen. □

Because iterated addition is in NC^1, binary multiplication is also in NC^1. We can also perform matrix multiplication in NC^1: we do every useful element-wise multiply in one binary multiplication layer, and follow it by a layer of iterated addition. Both subproblems are in NC^1, so their concatenation is in NC^1.

A symmetric function is a function whose value does not change when its input bits are permuted; thus, its value is dependent only on the size and the number of ones in the input. Both of these can be determined by iterated addition, so all symmetric functions are NC^1–computable, though not necessarily in Uniform NC^1.

It is known that iterated multiplication is also Uniform NC^1–computable, though this proof is more complex.

4.3 NC^2

By a divide-and-conquer algorithm using circuits in NC^1, we can show that iterated matrix multiplication is in NC^2. This implies that matrix inversion, linear systems, and most of the rest of linear algebra is NC^2–computable.

4.4 Upper Bounds

For the classes NC^k with $k > 0$, known upper bounds on computation power are quite weak. For example, the truth of the following statements are all open questions:

- $P \subseteq$ Uniform NC^1?
- $P \subseteq$ Uniform NC?
- $NP \subseteq$ Uniform NC^1?

CVP is the Circuit Value Problem: Given a circuit C and an input x, return what C would output given x. Because CVP is P-complete under log-space mapping reductions, we know that P is in Uniform NC iff CVP is NC-computable.

5 Connection Between NC^1 and BP

The following theorem states that NC^1 circuits and bounded-width branching programs of polynomial size are equally powerful. The proof uses formulas, a restricted version of circuits.

Definition 2. *A* formula *is a circuit in which all gates have a maximum fanout of 1.*

Since every gate in a formula has a maximum fanout of 1, the number of gates in a formula matches our notion of the size of a boolean expression. A standard circuit may have the shape of any directed acyclic graph, but a formula must look like a rooted tree, except at the inputs. Thus, a circuit might be much smaller than an equivalent formula by reducing duplication and sharing outputs.

Theorem 4. *The following are equivalent in power:*

1. NC^1 *circuits*
2. *Polynomial–size formulas*
3. *Log–depth formulas*
4. *Bounded-width branching programs of polynomial size*

Proof. We will show that each of the theorem's elements can simulate its predecessor in the above theorem; so, poly-size formulas capture NC^1 circuits, log–depth formulas capture poly-size formulas, and so on.

Proposition 1. NC^1 *circuits can be simulated by poly-size formulas.*

Proof. A circuit forms a rooted directed acyclic graph, with its root at the topmost operator. Given an NC^1–circuit, we can transform it into a formula by recursively replacing subgraphs. For each node with fanout k, with $k > 1$, we replace that node (and its child subgraph) with k copies of the node (and its child subgraph) so that each node has fanout 1, and each node is the child of one of the old parents. For example, the black node in Figure 2 gets transformed in this way.

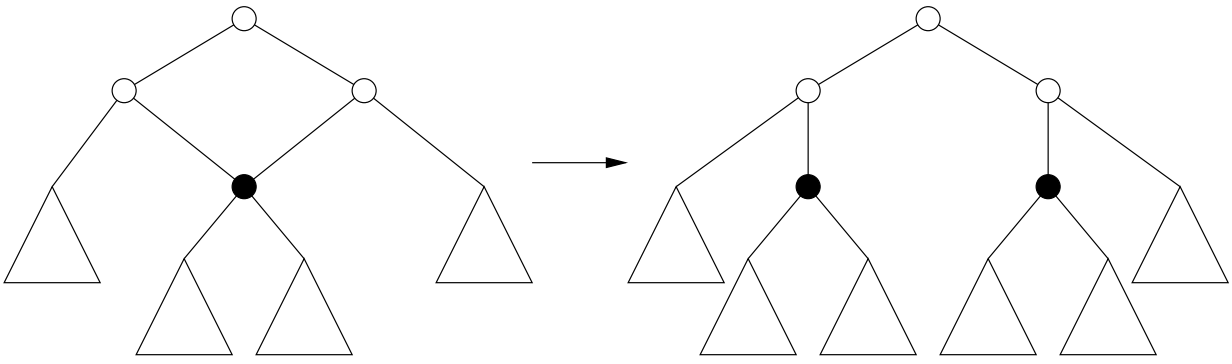


Figure 2: One step of the recursive transformation from NC^1–circuit to poly–size formula.

The circuit has only one root for output. So, suppose we repeat this procedure at every node from the root down on a circuit of maximum fanin f and depth d. The number of nodes at depth

$t+1$ is no more than f times the number of nodes at depth t, so there are at most f^t nodes at layer t. The size of the bottom layer dominates the size of the formula, so this process yields a formula of size $O(f^d)$. Because all NC^1 circuits have depth $O(\log n)$, the size of the formula is $f^{O(\log n)}$, which is polynomial in n. At each step, the function computed by the generated circuit remains the same, so this process creates a poly-size formula equivalent to an NC^1 circuit. □

Proposition 2. *Polynomial–size formulas can be simulated by log–depth formulas.*

Proof. Given a formula with binary fanin, we can find an edge of the formula so that the sub-formulas on either side of that edge are at least 1/3 the size of the whole formula. Let f be the sub-formula at the low side of the cut edge, and let g_x be the sub-formula on the high side of the cut edge with the constant literal x placed where f was. Figure 3 illustrates these two trees.

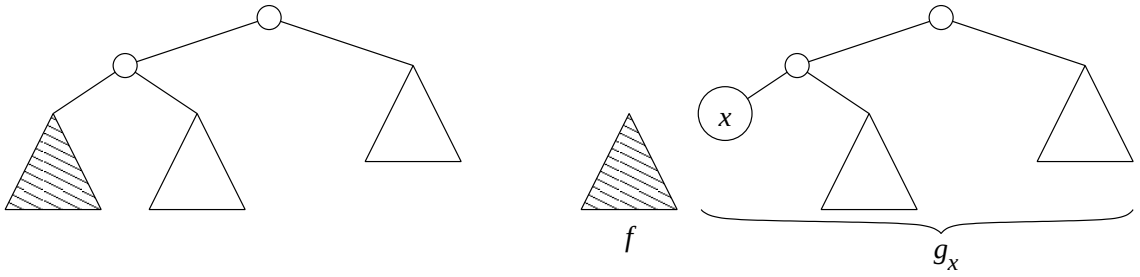


Figure 3: First step in the transformation from polynomial size formulas to log-depth formulas. A cut at a well-chosen edge of a formula yields two sub-formulas, f and g.

From f and g, we create a formula with the same function as the original, but with decreased depth. This formula is $(f \wedge g_1) \vee (\neg f \wedge g_0)$, and is shown in Figure 4. To see that this formula computes the same function as the original, consider the value of f. If f is 1 on its inputs, then the original function would have had the value of g_1. Likewise, if f is 0 on its inputs, then the original function would have had the value of g_0. So, the new function combines both cases.

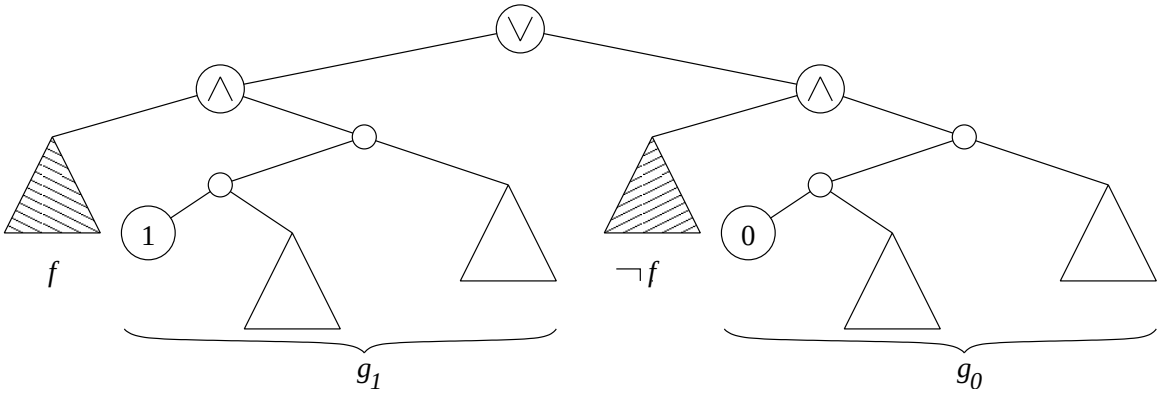


Figure 4: Second step in the transformation from polynomial size formulas to log-depth formulas. How to combine the two sub-formulas.

We then recurse the procedure on the sub-trees of f, g_0, and g_1. We continue recursing until we are considering constant-size formulas. Let s be the size of the original formula. Notice that f, g_0, and g_1 are each of size at most $2s/3$. After applying the next step, the sub-formulas being considered are further reduced by a factor of $2/3$. Also notice that each level of recursion places a depth two circuit at the top of the sub-formula being worked on. Then, the depth d of the final formula generated satisfies the inequality $2 \cdot (2/3)^d s \geq 1$, in other words $d = O(\log s)$. □

Proposition 3. *Bounded-width branching programs of polynomial size can be simulated by* NC^1 *circuits.*

Proof. Suppose B is a branching program of width w, containing a polynomial number of layers p. We construct an NC^1 circuit to simulate B with the following divide–and–conquer strategy:

1. Place an OR gate, with fanin w. We will ensure that this OR gate is true if the input induces a path from the start state in the first layer of B to the accepting state in the last layer (layer p) of B.

2. At the i^{th} input to the OR gate, place an AND gate with fanin two. This AND outputs true if the input induces a path from the start state of the first layer, to the the i^{th} state of layer $p/2$, and to the accepting state of layer p. One input to this AND is true iff the sub-path from layer 1 to layer $p/2$ is induced, and the other input is true iff the sub-path from layer $p/2$ to layer p is induced.

3. Recurse on the inputs of the ANDs until reaching the base case of checking adjacent layers.

Because p is polynomial, this divide–and–conquer strategy recurses only $O(\log(n))$ times, giving the constructed circuit a logarithmic depth. To analyze the size of the circuit, we rely on the fact that we are generating a circuit and not a formula: once a sub-problem is computed once in the circuit, we do not need to compute it again if it is needed again. There are roughly $2p$ intervals considered in subproblems, and w^2 subproblems of the form "Can state a in layer A be reached from state b in layer B?" are asked for each interval. Thus, the number of individual "questions" that our circuit computes is only $2pw^2$, which is polynomial in the size of the input. So, the circuit we have constructed uses a polynomial number of gates; since it also has logarithmic depth, the circuit is in NC^1. □

Proposition 4. *Log-depth formulas can be simulated by bounded–width branching programs of polynomial size.*

Proof. The BP M we construct has the following properties:

- The width of M is 5.
- The label of each node depends only on its layer, that is, M is an *oblivious BP*.
- Between any two levels, all of the 0–branches have distinct end states and all of the 1–branches have distinct end states. Since the width of M is constant, this makes M a *permutation BP*.
- Overall, the effect of M is the identity permutation e if its input is rejected, or a single cycle π if its input is accepted. We say that such a BP is *π–accepting*.

The following claims and corollaries give use the components we'll need to construct M recursively, building structures equivalent to pieces of the log–depth formula we are given.

Claim 1. *If there exists a π–accepting BP of size s, for some cyclic π, then there exists a σ–accepting BP of size s for* any *cycle $\sigma \neq e$.*

Proof. Any cyclic permutation σ is conjugate to π; that is, there exists a permutation γ such that $\sigma = \gamma^{-1}\pi\gamma$. So, to construct a σ–acceptor from a given π–acceptor, we need only to permute the machine's top-layer nodes by γ^{-1} and the bottom-layer nodes by γ. □

Corollary 1. *If we can decide the language A using a π–acceptor of size s, then we can decide the language $\overline{A}$ using a π–acceptor of size s.*

Proof. Build a π^{-1}–acceptor that decides A. Apply π to the last nodes, and you have a π–acceptor that decides $\overline{A}$. □

Claim 2. *If there exists a π–acceptor of size s_A that decides language A and there exists a σ–acceptor of size s_B that decides language B, and $\tau = \pi^{-1}\sigma^{-1}\pi\sigma \neq e$, then there exists a τ–acceptor of size $2(s_A + s_B)$ that decides $A \cap B$.*

Proof. Suppose that M_A is the π–acceptor deciding language A and M_B is the σ–acceptor deciding language B. Let $M_{A\cap B}$ be the machine formed by concatenating M_A^{-1}, M_B^{-1}, M_A, and M_B, in that order, start-to-finish. (M_A^{-1} is a π^{-1}–acceptor deciding A, and M_B^{-1} is analogous. The previous claim shows how to build these.)

Consider the fate of input x fed to $M_{A\cap B}$. If $x \in A \cap B$, then x undergoes the permutation π^{-1} from M_A^{-1}, followed by σ^{-1} from M_B^{-1}, followed by π from M_A, followed by σ from M_B. Because $\tau = \pi^{-1}\sigma^{-1}\pi\sigma$, the net permutation that x undergoes is τ.

If, instead, x is in $A \setminus B$, then M_B^{-1} and M_B perform the identity permutation on x. Therefore, the net permutation experienced by x is $\pi^{-1}\pi$, or just e, and x is rejected. The case where x is in $B \setminus A$ is similar, and the case where x is in neither A nor B is even clearer. So, the machine $M_{A\cap B}$ accepts precisely the language $A \cap B$, and the size of the machine $2(s_A + s_B)$. □

There exist permutations π and σ so that $\tau \neq e$, for example, let $\pi = (12345), \sigma = (13542)$, and $\tau = (12534)$. Thus, we will be able to combine branching programs to simulate AND gates; and since we've already seen how to complement these branching programs, we can simulate OR gates by De Morgan's law.

We need to encode individual input variables as well, but these are trivial within this model: the 0–branches form the permutation e and the 1–branches form the permutation π, and the label of every node in the layer is the relevant variable.

So, consider the standard post-order formula traversal that these constructions suggest. At each level, our branching program may increase by no more than a factor of two. If the given formula has depth d, then the branching program we construct has size $O(2^d)$. Since d is $O(\log n)$, the size of the branching program is polynomial in n. □

So, we have proved that each of the four computational models in question can be transformed into another of the four. Since we can chain these transformations as we please, any machine in any these models can be transformed into an equivalent machine in any of the other models. All four models are therefore equivalent in power. □

CS 810: Complexity Theory 2/14/2007

Lecture 11: Randomness

Instructor: Dieter van Melkebeek Scribe: Jake Rosin

Last time we discussed parallelism as an extension of our existing model of computation. Today we introduce randomness and extend the Turing machine model to make use of it. We show the qualitative usefulness of randomness through efficient probabilistic algorithms for a few difficult problems, and then quantify its power in comparison to existing computational classes.

1 Motivation

Randomness appears to have great value as a computational tool. It simplifies problems in a large number of settings, and is essential in others; for example the Dining Philosophers Problem, which cannot be solved by any deterministic solution. The use of randomness also seems essential to cryptography, where the use of a deterministic method to hide a secret means that it could then be deterministically found by an adversary. Randomness in a cryptographic setting will be covered in upcoming lectures.

Here we deal with randomness in the standard setting: that of realizing a mapping from inputs to outputs, for example decision problems. Intuitively randomness seems like a hindrance in such a setting, but there are certain problems which may be solved more efficiently using randomness than by any known deterministic algorithm. Unfortunately it is not known whether randomness truly provides additional computing power, as we will see.

2 Concept

To make use of randomness we allow a TM to flip coins and base decisions on the outcome of those flips. The configuration of the machine at a given time therefore becomes a random variable based on coin flips. As the outcome is dictated by the final configuration of the machine, it also becomes a random variable. In the context of decision problems this introduces the possibility of an incorrect result.

Obviously the probability of an error should be made negligible, but useful results can be easily generated by a machine which errs with probability nontrivially less than $\frac{1}{2}$, by running the machine repeatedly and taking the majority vote of its outcomes. Given a machine which produces an erroneous result with probability $\epsilon = \frac{1}{2} - \delta$, and the results from k independent runs:

$$\Pr[\text{Majority vote is wrong}] = \sum_{i=\frac{k}{2}}^{k} \binom{k}{i} \epsilon^i (1-\epsilon)^{k-i} \leq 2^k (\epsilon(1-\epsilon))^{\frac{k}{2}} \leq (1-4\delta)^{\frac{k}{2}} \leq e^{-2k\delta^2} \qquad (1)$$

This shows it is possible to produce an exponentially small probability of error through majority vote of a polynomial number of runs if the original probability of error is not too close to 1/2.

There are different ways to allow error in a randomized computation, and which type of error is allowed may affect the class of languages that can be computed.

- 2-sided error: error possible on both the membership and non-membership sides.
- 1-sided error: error possible on only one side; typically the membership side. Invalid input is always rejected, while valid input is rejected with small probability.
- 0-sided error: never provides an incorrect answer, but may answer "unknown" with small probability.

Randomized quick sort is an example of a 0-sided algorithm. Quick sort is guaranteed to produce the correct result: it will never answer "unknown." The behavior of quick sort is affected by randomness, however, and time, space, and other aspects of program behavior become random variables.

3 Randomized Algorithms

The goal when randomness is introduced is to obtain solutions with more efficient algorithms than are known deterministically, with efficiency measured in time or space.

3.1 Sequential Time Efficiency

Polynomial identity testing is a problem where we know of an efficient randomized algorithm but do not know of an efficient deterministic algorithm. Given an arithmetic formula φ composed of addition, subtraction, multiplication, brackets and variables, the problem is to determine if $\varphi \equiv 0$. Solving this problem deterministically appears difficult; one method involves expanding all terms and comparing monomials, but the number of monomials may be exponential in the length of the formula. In fact all known deterministic algorithms run in exponential time.

A randomized solution is formed by choosing values for all variables at random and evaluating the formula. If the result $\neq 0$ the formula is rejected; if it $= 0$ then the formula may $\equiv 0$ with some probability. A bound on the probability of an error can be determined as below. Given that values for each variable x_i are chosen uniformly at random from some set I, with d being the degree of φ:

$$\Pr[\varphi(x_1, \ldots x_n) = 0 | \varphi \not\equiv 0] \leq \frac{d}{|I|} \tag{2}$$

The degree of a formula is bounded by its length, so $d \leq N$. The interval I should be chosen such that $\frac{d}{|I|}$ is sufficiently small. We use $|I|$ some polynomial in N thus ensuring $\frac{d}{|I|}$ is small and still each value is specified with $O(\log N)$ bits. Evaluating the formula at worst raises the variables to the power N, meaning the resulting numbers have $O(N \log N)$ bits. All the arithmetic operations involved may be performed in polynomial time in the bit length of these numbers. This forms a 1-sided randomized algorithm for polynomial identity testing, which errs on nonmembers with small probability (producing false positives).

Arithmetic circuit testing can also benefit from this approach. The relevant decision problem is the following: given an arithmetic circuit and inputs, does it evaluate to zero? Because the degree of the corresponding polynomial can be exponential in the depth of the circuit, exact calculation uses $2^{\text{poly}\, N}$ bits, which prevents evaluation in polynomial time. The solution is to perform all calculations modulo some random number m of at most $\text{poly}\, N$ bits. This allows evaluation in polynomial time but introduces another source of error to be dealt with.

Polynomially many prime numbers produce a zero incorrectly (these numbers are the prime factors of the true result). To reduce the probability of an error we must pick our prime numbers from a set so that picking such a prime number from the set happens with small probability. To achieve this, we pick a prime number at random from $[0, \text{poly}(N)]$ for a large enough polynomial. For this to work, we must have some guarantee that for a randomly chosen $\text{poly}(N)$ there are a nontrivial number of primes in the interval: the prime number theorem provides this assurance. If $\pi(x)$ is the prime counting function that gives the number of primes less than or equal to x, then:

$$\pi(x) \sim \frac{x}{\ln x}. \tag{3}$$

The result is a 1-sided algorithm with Pr[correct]= $\frac{1}{\text{poly}}$ which errs on nonmembers. Running the algorithm multiple times can increase the confidence in the result. As was the case for polynomial identity testing, the best known deterministic algorithm for arithmetic circuit testing runs in exponential time.

3.2 Parallel Time Efficiency

The *existence of a perfect matching in bipartite graphs* problem has no known deterministic parallel algorithm. An inherently sequential polynomial time algorithm exists, but finding a parallel algorithm would greatly improve the efficiency with which this problem can be solved. Such an algorithm can be generated by using randomness.

The graph G can be represented as an $N \times N$ adjacency matrix. We replace every '1' in the graph with a random variable unique to that location:

$$M = \begin{bmatrix} x_{11} & 0 & 0 & x_{14} & \cdots \\ 0 & x_{22} & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{4}$$

Claim 1. *G has a perfect matching iff Det(M) is not $\equiv 0$*

Proof. One term in the determinant exists for every possible permutation; permutations are in a 1-to-1 correspondence with perfect matchings. The determinant contains exactly one term for every possible perfect matching; the coefficient of that term is zero if that matching is not possible in the graph. Different perfect matchings lead to different monomials, which if they have a non-zero coefficient are valid for the graph. □

The determinant is a multi-variate polynomial of degree at most N. Checking for a perfect matching can be performed by checking if that polynomial is $\equiv 0$, which can be accomplished using the randomized algorithm above. However, this would not be a parallel algorithm. Computing the determinant is a linear algebra problem, which as we saw last lecture is in NC^2. We use this fact to device an NC^2 algorithm:

- Replace all '1's in the adjacency matrix with a random value in a suitable interval $I = [0, \text{poly}(N)]$
- Compute the determinant in parallel

If the result is non-zero then a perfect matching exists. If zero, the algorithm can be repeated until we become sufficiently confident in the result.

3.3 Space Efficiency

The *undirected path problem* was until recently most efficiently solved by a randomized algorithm. We now have an efficient deterministic Ł solution, but the problem is included as a well-known example. Note that the related directed path problem is NL-complete.

Given an undirected graph G and two vertices s and t the problem is to determine if a path exists from s to t. The randomized algorithm is a random walk beginning from s. This walk ends after a certain number of steps, or immediately if t is reached. Performing a polynomial number of steps reduces to a small probability the chance of not reaching t if s and t are connected. This will be demonstrated in an upcoming lecture.

This algorithm can be performed in logarithmic space, storing the following:

- Current location
- Destination t
- # of steps taken - at most polynomially large, and so representable in logarithmic bits.

4 Model

We model randomness by equipping a standard Turing machine with a random bit tape, to which the finite control has one-way read access. We assume this tape is filled with bits from a perfectly uniform distribution before computation begins.

The machine is given one-way read-only access because the random bit tape is not counted as used space. To reuse a random value it must be stored on some work tape, while using a random value takes a single computational step to read an entry from the random bit tape.

4.1 Time and Space

The complexity classes BPTIME(t) and BPSPACE(s) are analogous to their deterministic counterparts. They represent the classes of problems solvable by a randomized machine under some time or space bound, with the requirement that:

$$\Pr[\text{error}] \leq \frac{1}{3}$$

Specifically we refer to the classes BPP (bounded-error probabilistic polynomial time) and BPL (log space).

Complexity classes also exist for 1-sided and 0-sided machines. RTIME(t) and RSPACE(s), and the specific classes R (or RP) and RL, refer to problems solvable by randomized machines which meet the following criteria:

$$\Pr[\text{error}|x \in L] \leq \frac{1}{2}$$
$$\Pr[\text{error}|x \in \bar{L}] = 0$$

ZPTIME(t) and ZPSPACE(s), and specifically the classes ZPP and ZPL, refer to problems solvable by 0-sided randomized machines. These machines must meet these criteria:

$$\Pr[\text{error}] = 0$$
$$\Pr[\text{“Unknown”}] \leq \frac{1}{2}$$

Claim 2. *The class* ZPP *is equivalent to the class of problems solvable by randomized machines with no chance of error that run in expected polynomial time (call this class* ZPP2*)*

Proof. A ZPP machine can be run repeatedly until it outputs a definite answer. By the definition of ZPP each run halts in polynomial time, and outputs an answer with probability $\geq \frac{1}{2}$. If N^c is the running time of our ZPP algorithm, then the expected running time of our modified algorithm is:

$$\text{E(run time)} \leq \sum_{i=1}^{\infty} (N^c \cdot i) \cdot (\frac{1}{2})^i \tag{5}$$
$$= N^c \sum_{i=1}^{\infty} i \cdot (\frac{1}{2})^i = 2N^c \tag{6}$$

This new machine has expected running time polynomial and thus satisfies the definition of ZPP2.

Similarly a ZPP2 machine can be run on a clock for $t\cdot$ E(run time), then terminated with output “Unknown”. This modified algorithm outputs “Unknown” with probability at most:

$$\Pr[x \geq t \cdot E(x)] \leq \frac{1}{t} \text{ (Markov's inequality).} \tag{7}$$

To ensure the probability of outputting “Unknown” is at most 1/2, we set $t = 2$. As this new algorithm satisfies the error criterion of ZPP and runs in polynomial time, it is a ZPP algorithm for the language. □

Claim 3. $\text{ZPP} = \text{RP} \cap \text{coRP}$

The proof of this claim is left as an exercise.

4.2 Space Complications

In a deterministic setting any log-space algorithm terminates in polynomial time (if it terminates at all), due to the polynomial bound on the number of possible machine configurations. Obviously a repeated configuration signifies a loop, which occurs iff the machine does not halt.

Things become more complicated when randomness is involved. *When discussing space bounds in this setting we assume machines that always halt.* As in the deterministic setting this requires that the machine never repeats a configuration, because doing so would introduce a non-halting loop. This restriction gives us the nice property that a randomized log-space machine runs in polynomial time (and in general, a space s machine runs in $2^{O(s)}$ time).

Note that machines of this type are distinct from those which halt with probability 1. We define a separate class, $\lfloor \text{ZPL} \rceil$, for that type of machine. An example of the addition power this model has over the one stated above is given by the following claim, which we leave as an exercise.

Claim 4. $\lfloor \text{ZPL} \rceil = \text{NL}$.

For more on how these definitional issues affect the power of randomness in the log-space setting, see the survey [1].

5 Relation to Deterministic Classes

Relating these randomized classes to the known deterministic classes provides a measure of their computational power.

$$\begin{aligned} &\mathrm{RP} \subseteq \mathrm{NP} \\ &\mathrm{P} \subseteq \mathrm{ZPP} \subseteq \mathrm{RP} \subseteq \mathrm{BPP} \subseteq \mathrm{EXP} \end{aligned}$$

All but the last inclusion are by definition. BPP $\subseteq$ EXP follows from the fact that exponential time is sufficient to exactly compute the probability of acceptance by a randomized machine, by exhaustively generating all possible coin flip results.

$$Ł \subseteq \mathrm{ZPL} \subseteq \mathrm{RL} \subseteq \mathrm{BPL} \subseteq \mathrm{UniformNC}^2 \subseteq \mathrm{DSPACE}(\log^2 N) \tag{8}$$

Uniform $\mathrm{NC}^2 \subseteq \mathrm{DSPACE}(\log^2 N)$ follows by a proof similar to that of $\mathrm{NC}^1 \subseteq Ł$ given in the last lecture. BPL $\subseteq$ Uniform NC^2 takes a bit more work.

Claim 5. *$BPL \subseteq$ Uniform* NC^2

Proof. A BPL computation can be viewed as a Markov chain of machine configurations. For a log-space machine the size of the set of configurations is polynomial. The Markov chain can be represented as a matrix, with each entry M_{ij} representing the probability of transitioning from state j to state i. Multiplying this matrix against itself N times gives the probability matrix for state transitions taking exactly N steps. This can be used to determine the probability of reaching an accepting configuration:

$$\begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1m} \\ p_{21} & p_{22} & \cdots & p_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m1} & p_{m2} & \cdots & p_{mm} \end{bmatrix}^N \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{9}$$

This is iterated matrix multiplication, which as discussed last time is in Uniform NC^2. □

In fact there is a tighter bound known for the class BPL:

Claim 6. BPL $\subseteq \mathrm{DSPACE}\left(\log^{3/2} N\right)$

The proof for this claim will be presented in a future lecture, along with evidence for the conjectures $BPP = P$ and $BPL = L$. As with all complexity classes we ignore constant-factor increases.

The following facts are also known (proving the second is the final problem on Homework 1).

Proposition 1.

$$\begin{aligned} &\mathrm{BPP}^{\mathrm{BPP}} = \mathrm{BPP} \\ &\mathrm{NP} \subseteq \mathrm{BPP} \implies \mathrm{PH} \subseteq \mathrm{BPP} \\ &\mathrm{NP} \subseteq \mathrm{BPP} \implies \mathrm{NP} = \mathrm{RP} \end{aligned}$$

The results given above relate randomized complexity classes among each other and with deterministic and nondeterministic complexity classes. The following result relates the randomized class BPP to the non-uniform class *P/poly*, showing that in general randomness can be replaced by non-uniformity.

Theorem 1. BPP $\subseteq$ *P/poly and* BPL $\subseteq$ *L/poly*

Proof. Consider a BPP algorithm. We first make the probability of error smaller than the number of inputs of a given length N: Pr[error] $< \frac{1}{2^N}$. Recall from section 2 that running a BPP algorithm with error $1/2 - \delta$ for k times and taking the majority vote results in a BPP algorithm computing the same language but now with error at most $e^{-2k\delta^2}$. We assume original error at most $1/3$, so $\delta \geq 1/6$. We need to pick k large enough so that $e^{-2k(1/6)^2} < 1/2^N$. Then a large enough $k = \Theta(N)$ suffices, meaning the resulting BPP algorithm still runs in polynomial time.

Because we have reduced the error to less than $1/2^N$, given any distinct setting for the random bit tape, the probability that there exists some input x of length N on which the machine makes an error is less than one. Therefore there exists at least one coin flip sequence for which the machine gives the correct result on all inputs of length N. This sequence of coin flips is the advice given. As the amount of randomness used is some polynomial, this is a polynomial amount of advice. □

We stated above that it is conjectured that BPP=P, but we have been unable to prove this. In fact, we cannot yet even prove BPP $\subseteq$ NP, but it has been shown that BPP lies within the polynomial hierarchy.

Theorem 2. BPP $\subseteq \Sigma_2^P \cap \Pi_2^P$

The proof for this theorem will be presented in the next lecture.

6 Next Lecture

In the next lecture we will look at *expanders*, a useful tool which allows error reduction using fewer random bits. Reducing the number of coin flips used will improve efficiency of the trivial derandomization of evaluating every possible series of flips. Expanders are analogous to pseudorandom number generators, in that from a short random input they deterministically produce a series of pseudorandom bits which can be used for many randomized functions as if they were truly random.

We will also address the issue that while our model relies on a series of bits whose randomness is unbiased and uncorrelated, this is functionally impossible to achieve in the real world. Expanders also allow transformation from a series of bits with some randomness to another with essentially perfect randomness.

References

[1] Michael Saks. Randomization and Derandomization in Space-Bounded Computation. *Annual Conference on Structure in Complexity Theory*, 1996.

CS 810: Complexity Theory 2/16/2007

Lecture 12: Expanders

Instructor: Dieter van Melkebeek Scribe: Nathan Collins

In the last lecture we introduced randomized computation in terms of machines that have access to a source of random bits and that return correct answers more than $\frac{1}{2}$ of the time. We showed that BPP has polynomial-size circuits and the conjecture in the community is that BPP = P.

Today we'll introduce expanders, a type of graph that is useful in improving (amplifying) randomized algorithms with little or no additional random bit overhead. But first we'll prove a theorem relating BPP to the polynomial time hierarchy.

1 Hierarchy Results for BPP

Generally, whenever we introduce a new computational model, we look for hierarchy results: Does the class of problems we can solve using the model in question depend on the resources we are allowed? Most hierarchy results use a computable enumeration of all machines of the type in question, but the following can be shown.

Exercise 1. *There does not exist a computable enumeration of the randomized machines with error probability bounded away from* $\frac{1}{2}$.

Therefore, typical hierarchy arguments fail for randomized machines. The typical hierarchy arguments can be tailored to prove a hierarchy theorem for a modified model of randomized computation called promise-BPP. See HW 1.2. However, no hierarchy result is known for BPP.

2 BPP and the Poly Time Hierarchy

Although we don't know if BPP = P, or even if BPP $\subseteq$ NP, we do know that BPP $\subseteq$ PH:

Theorem 1. BPP $\subseteq \Sigma_2^p \cap \Pi_2^p$

Proof. Fix M a randomized polytime machine that accepts $L \in$ BPP, and let r be the number of random bits M uses when running on an input x of size $n = |x|$. We will now show that BPP $\subseteq \Sigma_2^p$. Since BPP = *co*BPP it follows that BPP $\subseteq co\Sigma_2^p = \Pi_2^p$, completing the proof.

We need to remove randomness from the computation. For a given input x, the space $\{0,1\}^r$ of r-bit strings gets partitioned into two pieces: $Acc(x)$, the set of random strings on which M accepts x, and $Rej(x)$, the set of strings on which M rejects x. If the error rate ε of M is small, then $Acc(x)$ will be much larger than $Rej(x)$ when $x \in L$, and $Acc(x)$ will be much smaller than $Rej(x)$ when $x \notin L$. See Figure 1. If our random computation has a small error rate then it will be correct on most random bit strings. We'll turn "most" into "all." The idea is that when $x \in L$ a few "shifts" of $Acc(x)$ will cover the whole space $\{0,1\}^r$ of r-bit random sequences, while the same few shifts of $Acc(x)$ will fail to cover the whole space when $x \notin L$.

If $S \subseteq \{0,1\}^r$ and $\sigma \in \{0,1\}^r$, then $S \oplus \sigma = \{s \oplus \sigma | s \in S\}$, the *shift* of S by σ.[1] Since shifting is invertible we see that $|S \oplus \sigma| = |S|$.

[1] The symbol "$\oplus$" denotes XOR, or, equivalently, addition in $\mathbb{Z}_2^r$

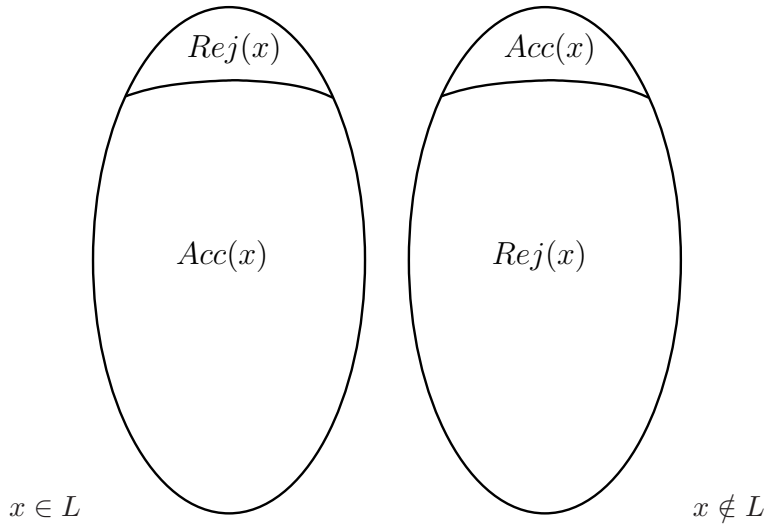


Figure 1: If the error rate ε of M is small, then $Acc(x)$ will be much larger than $Rej(x)$ when $x \in L$, and $Acc(x)$ will be much smaller than $Rej(x)$ when $x \notin L$.

We will use shifts to give a Σ_2-predicate using the intuition discussed above. Namely, consider

$$x \in L \iff \exists \sigma_1, \ldots, \sigma_t \forall \rho \in \{0,1\}^r [\rho \in \bigcup_{i=1}^{t} Acc(x) \oplus \sigma_i]. \tag{1}$$

Now, r is poly in n, and so if we can pick t poly in n as well, then the above will be a Σ_2^p-predicate, provided that we can verify the membership of ρ in $\bigcup_{i=1}^{t} Acc(x) \oplus \sigma_i$ in time poly in n. The membership check is no problem since we can equivalently check that $\rho \oplus \sigma_i \in Acc(x)$ for some i, which is polytime since it corresponds to running M on x with the random bit string $\rho \oplus \sigma_i$, for poly-many σ_i.

We'll show we can pick t a suitable poly to make (1) true by showing that we can choose the σ_is randomly with a high rate of success. There are two cases to consider:

1. $x \in L$: For ρ fixed,

$$\Pr_{\sigma_1,\ldots,\sigma_t}[\rho \notin \bigcup_{i=1}^{t} Acc(x) \oplus \sigma_i] \leq (\mu(Rej(x))^t \leq \varepsilon^t,$$

where $\mu(Rej(x))$ is the probability that M rejects when it should accept, and is hence not larger than ε. So, by union bound,

$$\Pr[\{0,1\}^r \not\subseteq \bigcup_{i=1}^{t} Acc(x) \oplus \sigma_i] \leq |\{0,1\}^r|\varepsilon^t = 2^r\varepsilon^t,$$

and hence if $2^r\varepsilon^t < 1$ then some choice of the σ_is must work.

2. $x \notin L$: We need to be sure that for any choice of the σ_is that some bit string ρ makes M reject x. But

$$\mu(\bigcup_{i=1}^{t} Acc(x) \oplus \sigma_i) \leq \sum_{i=1}^{t} \mu(Acc(x) \oplus \sigma_i) = t\varepsilon,$$

since $\mu(Acc(x)) \leq \varepsilon$ when $x \notin L$, and hence we need $t\varepsilon < 1$.

So, we need to choose t so that both $t\varepsilon$ and $2^r\varepsilon^t$ are less than 1. So $t = r$ works, provided that $\varepsilon < \frac{1}{r}$. From the definition of BPP we only know that $\varepsilon < \frac{1}{3}$, but, using the "majority vote" trick introduced during the last lecture, we can in k runs reduce the error to $e^{-2k\delta^2}$, where $\delta = \frac{1}{2} - \varepsilon > 0$. Since k runs will need kr random bits, we see that it suffices to choose k poly and large enough that $e^{-2k\delta^2} < \frac{1}{rk}$. In fact, $k = O(\log r)$ works, and so an examination of the above shows that the time complexity is $O(T^2 \log T)$ if the original run-time was T. □

3 Expanders

The proof of Theorem 1 used the "majority vote" trick to get an exponential in k increase in accuracy using a linear in k increase in random bit usage. Expander graphs, which are introduced here, lead to an "amplification" technique that gives an exponential in k accuracy improvement using only a constant increase in random bit usage (rk versus $r + k$). Expanders have many other uses, some of which we mention in this lecture, and others we may see in later lectures.

Definition 1 ((k, c)-expanding)**.** *A graph $G = (V, E)$ is* (k, c)-expanding *if $S \subset V$ with $|S| \leq k$ implies that $|\Gamma(S)| \geq c|S|$, where $\Gamma(S) = \{v \in V | \exists s \in S, (s, v) \in E\}$ is the* neighborhood *of S in G.*

Notice that any graph is trivially (k, c)-expanding for all k for all $c \leq 1$.

Definition 2 (Expander family)**.** *An* expander family *is an infinite sequence of graphs $G_1, G_2, \ldots$ and fixed numbers $c > 1$ and d such that each G_n has degree $\leq d$ and is $(\frac{N}{2}, c)$-expanding, where $N = |V_n|$ is the number of vertices in G_n.*

Intuitively, expander graphs are "very connected" in that the number of vertices reachable from a given subset of vertices is proportional to the size of that subset, at least when the subsets aren't so large as to make this impossible.

3.1 Graph Theoretic Properties

From now on we'll assume all our graphs are d-regular. We describe a d-regular graph using a normalized adjacency matrix. This view of a expander graphs proves useful in the analysis of their properties and randomized algorithms that make use of them.

Definition 3 (Normalized adjacency matrix)**.** *Given a d-regular $G = (V, E)$ its* normalized adjacency matrix *A is defined by*

$$A_{ij} = \begin{cases} \frac{1}{d}, & (i, j) \in E \\ 0, & \textit{otherwise} \end{cases}$$

The normalized adjacency matrix describes the Markov chain of a random walk on G:

$$A_{ij} = \Pr[\text{go to state } i \text{ from state } j],$$

and if p is a column vector with p_i the probability of being at vertex i, then $(Ap)_i$ gives the probability of being at vertex i after one random step in G.

The normalized adjacency matrix has a number of properties that can be proved using basic linear algebra. We will use the following properties, but we do not prove them here.

Proposition 1. *A is real-symmetric and so: all eigenvalues of A are real, and A has a basis of orthogonal eigenvectors.*

It turns out that the eigenvalues of the normalized adjacency matrix of an expander are closely connected to the expanders graph properties. The following are some basic properties, whose proofs we omit.

Proposition 2. *Let A be a normalized adjacency matrix of a graph on N vertices.*

1. *Each eigenvalue λ of A satisfies $|\lambda| \leq 1$.*
2. *1 is an eigenvalue of A, with corresponding eigenvector given by the uniform distribution on $(1/N, 1/N, ..., 1/N)$.*
3. *The multiplicity of the eigenvalue 1 is greater than 1 iff G is disconnected.*
4. *A has -1 for an eigenvalue iff G is bipartite.*

As the uniform distribution is always an eigenvector, we look only at the remaining eigenvector/values. It turns out that the second largest eigenvalue in absolute value is often useful to work with.

Definition 4 ($\lambda(G)$)**.** *Let $\Lambda = \{\lambda | \exists e_\lambda [Ae_\lambda = \lambda e_\lambda \wedge \langle e_\lambda, u \rangle = 0]\}$. Then*

$$\lambda(G) = \max_{\lambda \in \Lambda} |\lambda|$$

is the largest eigenvalue corresponding to an eigenvector orthogonal to u.

So, if G is connected and not bipartite then $\lambda(G) < 1$. The next two theorems will not be proven here. See the notes for CS 880 Spring 2006 for proofs.

Theorem 2. *If G is $(\frac{N}{2}, c)$-expanding then $\lambda(G) < f(c, d)$ where $f(c, d) < 1$ if $c > 1$ and is a function whose value depends only on c and d.*

Corollary 1. *Expanders have positive spectral gaps and expander families have spectral gaps bounded away from 0, where the spectral gap of a graph G is $1 - \lambda(G)$.*

Theorem 3. *If $\lambda(G) \leq \lambda < 1$ then G is $(\frac{N}{2}, g(\lambda))$-expanding where $g(\lambda) > 1$ when $\lambda < 1$ and is a function whose value depends only on λ.*

In light of Theorems 2 and 3 we see that we could just as well have defined expander families in Definition 2 as d-regular families with positive spectral gaps.

So, for expanders, the uniform distribution u is the only fixed point. The following Proposition tells us that the larger the spectral gap the faster an arbitrary probability distribution converges to the uniform distribution via a random walk on G.

Lemma 1. *For any probability distribution vector p*

$$\|A^t p - u\|_1 \leq \sqrt{N} \lambda^t,$$

where $\|v\|_q = \left[\sum_i |v_i|^q\right]^{1/q}$ is the q-norm of v and $\lambda = \lambda(G)$.

Proof. Since $Au = u$ we have $A^t p - u = A^t(p-u)$. Now, $(p-u) \perp u$:

$$\langle p-u, u\rangle = \langle p, u\rangle - \langle u, u\rangle = \sum_{i=1}^{N} p_i/N - \sum_{i=1}^{N} 1/N^2 = 1/N - 1/N,$$

since p is a probability distribution.

Write $p - u = \sum_i a_i e_i$, where the $\{e_i\}$ form an orthogonal eigen basis for A and $Ae_i = \lambda_i e_i$. Then, since the u-component of $p-u$ is 0, we have

$$\|A^t(p-u)\|_2 = \|A^t \sum_i a_i e_i\|_2 = \|\sum_i \lambda_i^t a_i e_i\|_2 \leq \lambda^t \|\sum_i a_i e_i\|_2 = \lambda^t \|p-u\|_2$$

Now

$$\|p-u\|_2^2 + \|u\|_2^2 = \|p\|_2^2 \leq \|p\|_1^2 = 1,$$

since $(p-u)\perp u$, and $\|v\|_2^2 \leq \|v\|_1^2$ for all vectors v. By the Cauchy-Schwartz inequality, we have

$$\begin{aligned}\|A^t(p-u)\|_1 &= |\langle(-1^{\sigma_1}, \ldots, -1^{\sigma_N}), A^t(p-u)\rangle| \\ &\leq \|(-1^{\sigma_1}, \ldots, -1^{\sigma_N})\|_2 \|A^t(p-u)\|_2 \\ &= \sqrt{N}\|A^t(p-u)\|_2,\end{aligned}$$

where $\sigma_k = \begin{cases} 0, & (A^t(p-u))_k \geq 0 \\ 1, & \text{otherwise} \end{cases}$. Combining all of the above we get

$$\begin{aligned}\|A^t(p-u)\|_1 &\leq \sqrt{N}\|A^t(p-u)\|_2 \\ &\leq \sqrt{N}\lambda^t\|(p-u)\|_2 \\ &\leq \sqrt{N}\lambda^t,\end{aligned}$$

completing the proof. □

This lemma can be used to prove that the random walk algorithm for the undirected path problem needs only polynomially many steps, i.e. that PATH ∈ BPL. The proof uses Lemma 1 and the following exercise.

Exercise 2. *If G is connected and not bipartite then $\lambda(G) \leq 1 - \frac{1}{dN^2}$.*

3.2 Constructions

To prove that expanders exist one can argue that a randomly chosen d-regular graph G has a high probability of being an expander. However, we want to use expanders to reduce the amount of randomness needed in random algorithms, so using randomness to construct expanders won't help us. We want explicit constructions for which given a vertex v and index i we can compute v's ith neighbor in time poly$(|v|, |i|)$.

Example: For any integer $m \geq 2$, we can get an expander G on vertices $V = \mathbb{Z}_m \times \mathbb{Z}_m$ with edges given by the relations

$$\Gamma(\{(x,y)\}) = \{(x, y \pm x), (x, y \pm (x+1)), (x \pm y, y), (x \pm (y+1), y)\}.$$

If $m \geq 4$ then the graph has degree 8. The proof that this construction works is non-trivial. See the notes for CS 880 for a partial proof using harmonic analysis. ⊠

There are other efficient constructions of expanders, but the aforementioned expander suffices for our needs. See the notes for CS880 Spring 2006 for other constructions.

3.3 Applications

We'll use expanders for two purposes:

1. *Deterministic amplification*: Given a randomized algorithm R that uses r random bits we reduce the error to be less than some arbitrary ε, without using any additional random bits. This requires running R $\text{poly}(1/\varepsilon)$ times.

2. *Randomness efficient amplification*: With R and r as above we reduce the error to be less than some arbitrary ε using $r + O(\log(1/\varepsilon))$ additional random bits and running R $O(\log(1/\varepsilon))$ times.

To accomplish the above amplifications we use an expander graph G whose vertices are in one-to-one correspondence with the bit strings in $\{0,1\}^r$. For application 1 we run R, look at the vertex v in G corresponding to the random bits used by R's run, and then run R once for each neighbor v' of v, with random bits corresponding to v'. For application 2 we use the same G and v. Then, starting from v, we perform a random walk in G starting from v of length $t = \log(1/\varepsilon)$. This gives t vertices $v_1, \ldots, v_t$. We then run R once for each vertex v_i with random bits corresponding to v_i. In both cases, we finish by taking the "majority vote" of R's runs. We discuss these applications and prove their correctness in the next lecture.

4 Next Time

In the next lecture we prove the correctness of the algorithms given in the applications above. To accomplish this, we also prove

Lemma 2 (Expander mixing lemma)**.** *Given a d-regular expander graph $G = (V, E)$ and $S, T \subseteq V$*

$$\left| \frac{|E(S,T)|}{dN} - \mu(S)\mu(T) \right| \leq \lambda(G)\sqrt{\mu(S)(1-\mu(S))}\sqrt{\mu(T)(1-\mu(T))}$$

where $E(S,T)$ is the set of edges connecting S to T.

Notice that dN is the number of ways to choose an edge in G: N choices of starting vertex and then degree d choices for ending vertex. Also, $\mu(S)\mu(T)$ is the probability that, given two randomly chosen vertices, one lies in S and the other in T.

CS 810: Complexity Theory 2/13/2007

Lecture 13: Amplification

Instructor: Dieter van Melkebeek Scribe: Matthew Anderson

Last lecture we introduced expander graphs. From a combinatorial point of view these were relatively sparse graphs where each vertex had a constant degree and were non-trivially expanding. From an algebraic point of view expanders are graphs of constant degree which have $\lambda(G) < 1$. Recall $\lambda(G)$ is the largest absolute value eigenvalue of the normalized adjacency of G corresponding to an eigenvector which is perpendicular to the all ones vector.

Today we discuss two applications of expanders for performing correctness amplification on randomized algorithms. The first application decreases the error rate of an algorithm using no additional randomness. The second application decreases the error rate of an algorithm even further using only slightly more randomness.

1 Expander Properties

$\lambda(G)$ determines how quickly random walks converge to the uniform distribution. Last lecture we showed that for a probability distribution p, a normalized matrix A and the uniform distribution u:

$$\|A^t p - u\|_1 < \sqrt{N}(\lambda(A))^t. \tag{1}$$

If $\lambda(A) < 1$, the distance between the uniform distribution and a random walk from an arbitrary distribution p decreases exponentially with the number of steps t. The LHS of the above equation can be written using the definition of the 1-norm:

$$\|A^t p - u\|_1 = 2 \cdot \max_{B \subseteq V} |\Pr[A^t p \in B] - \Pr[u \in B]| \tag{2}$$

Think of $\|A^t p - u\|_1$ as twice the distance between a uniform distribution and a random walk for any set of vertices B. Normally it would take $\log V$ random bits to select a vertex at random from V, using this property a vertex can be selected almost uniformly using $t \log d$ bits by fixing some start vertex and performing a random walk for t steps.

The next important algebraic property of expanders is called the *expander mixing lemma*:

Lemma 1 (Expander Mixing Lemma)**.**

$$\forall S, T \subseteq V, \left| \frac{|E(S,T)|}{dN} - \mu(S)\mu(T) \right| \leq \lambda \sqrt{\mu(S)(1-\mu(S))\mu(T)(1-\mu(T))} \tag{3}$$

This lemma bounds the difference in the distributions of picking two vertices uniformly at random (second term on left) with picking one vertex and one neighbor of the vertex at random (first term on left). In the first case $2 \log V$ random bits are used, in the second case only $\log V + \log d$ random bits are used. This idea allows the amount of randomness to be reduced at the cost of producing a distribution that is not quite uniform.

Proof. Expander Mixing Lemma

Recall that $A(G)$ is symmetric and real implying that it has a full orthonormal eigenbasis. The number of edges between two sets S and T can be written in terms of their relative characteristic vectors (i.e. $\chi_{S,i} = 1$ iff $v_i \in S$ and $\chi_{S,i} = 0$ otherwise):

$$|E(S,T)| = \chi_S^T (dA) \chi_T. \tag{4}$$

dA is now the standard adjacency matrix for G. Since A is a real and symmetric matrix, A has a full orthonormal eigenbasis. Both χ_S and χ_T can be rewritten in terms of their components parallel and perpendicular to the uniform vector u. Recall $u = (\frac{1}{N}, \frac{1}{N}, ..., \frac{1}{N})$ is the eigenvector corresponding to the eigenvalue 1. The inner product of χ_S with u is $(\chi_S, u) = \frac{|S|}{N} = \mu(S)$ and $\chi_S^{\parallel} = (\chi_S, \hat{u})\hat{u} = (\chi_S, \sqrt{N}u)\sqrt{N}u = |S|u$. Equation 4 can be rewritten:

$$\begin{aligned} |E(S,T)| &= (\chi_S^{\parallel} + \chi_S^{\perp})(dA)(\chi_T^{\parallel} + \chi_T^{\perp}) \\ &= \chi_S^{\parallel} dA \chi_T^{\parallel} + \chi_S^{\perp} dA \chi_T^{\perp} \\ &= d\chi_S^{\parallel} \chi_T^{\parallel} + \chi_S^{\perp} dA \chi_T^{\perp} \\ &= d\frac{|S||T|}{N} + \chi_S^{\perp} dA \chi_T^{\perp}. \end{aligned} \tag{5}$$

The second and third lines follow from the first because $\chi^{\parallel}$ is an eigenvector of A causing the first term to simplify and the cross terms to vanish. Dividing by dN, moving terms around and taking the absolute value gives:

$$\begin{aligned} \left| \frac{|E(S,T)|}{dN} - \mu(S)\mu(T) \right| &= \left| \frac{\chi_S^{\perp}(dA)\chi_T^{\perp}}{dN} \right| \\ &\leq \frac{\|\chi_S^{\perp}\|_2 \cdot \lambda d \cdot \|\chi_T^{\perp}\|_2}{N} \end{aligned} \tag{6}$$

The second line is reached by applying Cauchy-Schwarz to the RHS and using the fact that A decreases the magnitude of $\chi_T^{\perp}$ by at least λ as there is no component of $\chi_T^{\perp}$ along u. Applying the Pythagorean theorem and some simple algebra to $\|\chi_S\|_2$ we can derive the value of $\|\chi_S^{\perp}\|_2$:

$$\begin{aligned} \|\chi_S\|_2^2 &= \|\chi_S^{\parallel}\|_2^2 + \|\chi_S^{\perp}\|_2^2, \text{ therefore} \\ |S| &= |S|^2 \frac{1}{N} + \|\chi_S^{\perp}\|_2^2, \text{ and} \\ \|\chi_S^{\perp}\|_2^2 &= |S|(1 - \mu(S)), \\ \|\chi_S^{\perp}\|_2 &= \sqrt{|S|(1-\mu(S))}. \end{aligned} \tag{7}$$

Substituting this back in for $\|\chi_S^{\perp}\|_2$ and $\|\chi_T^{\perp}\|_2$ and pulling the factor of N into the square root completes the proof.

□

Both of these properties can be used to reduce the error probability of randomized algorithms while using little or no extra random bits.

2 Deterministic Amplification

Using expanders it is possible to transform a random algorithm, R, that takes r random bits and errs with probability $\epsilon_0 \leq \frac{1}{3}$, into an equivalent algorithm, R', also using r random bits which errs with probability $\leq \epsilon$ by calling R poly$(\frac{1}{\epsilon})$ times.

The idea behind this transformation is to consider an explicit expander graph G with 2^r vertices with each vertex corresponding to a random bit string in $\{0,1\}^r$. Pick one vertex, ρ', at random from G using r random bits. Look at all neighbors, $\rho \in V$ at distance t from ρ'. Run R on each of these neighbors and return the majority vote of these neighbors as the output of R'.

Intuitively, because G is expanding the set of neighbors at distance t will be "spread out" on the graph and close to uniformly distributed.

2.1 Analysis

Let S be the set of the bad random strings for R (the strings that give the wrong answer for an input x). Let T be the set of bad random strings ρ' for R' (the strings whose majority of neighbors at distance t give the wrong answer).

$$\begin{aligned} S &= \{\rho | R \text{ gives incorrect answer}\} \\ T &= \{\rho' | \text{ majority of } \mathrm{N^t}(\rho') \text{ give the incorrect answer}\} \end{aligned} \tag{8}$$

By construction of S, $\mu(S) \leq \epsilon_0 \leq \frac{1}{3}$. In order for R' to have the required error we want $\mu(T) \leq \epsilon$. Consider fixing $t = 1$ then:

$$\frac{|E(S,T)|}{dN} \geq \frac{|T|\frac{d}{2}}{dN} = \frac{\mu(T)}{2}. \tag{9}$$

This is because at least half of the immediate neighbors ($t = 1$) of each ρ' in T are in S (they were incorrect for R). Applying the expander mixing lemma to this gives:

$$\mu(T)|\frac{1}{2} - \mu(S)| = |\frac{\mu(T)}{2} - \mu(S)\mu(T)| \leq \left|\frac{|E(S,T)|}{dN} - \mu(S)\mu(T)\right| \leq \lambda\sqrt{\mu(S)\mu(T)}. \tag{10}$$

The two extra terms in the last expression were dropped because they must be less than or equal to 1. Rearranging terms and bounding $\mu(S) \leq \epsilon_0$:

$$\mu(T) \leq \frac{\lambda^2\mu(S)}{(\frac{1}{2} - \mu(S))^2} \leq \frac{\lambda^2\epsilon_0}{(\frac{1}{2} - \epsilon_0)^2} \leq \epsilon. \tag{11}$$

This only gives a constant decrease in error. Consider the effect of taking neighbors at distance t instead of only immediate neighbors. Replace G with G' where G' has the same vertices as G but there is an edge between two vertices u and v in G' if and only if there is a path of length t between u and v in G (allow multiple edges between two vertices in G'). This has the effect of increasing the degree of G' to d^t and decreasing second eigenvalue $\lambda(G') = \lambda^t(G)$, and the prior analysis still holds with $\lambda(G')$ substituted for $\lambda(G)$ This changes the result of the Equation 11 to give:

$$\mu(T) \leq \frac{\lambda^{2t}\epsilon_0}{(\frac{1}{2} - \epsilon_0)^2} \leq \epsilon. \tag{12}$$

Selecting $t = O(\log \frac{1}{\epsilon})$ gives error less than ϵ. The number of neighbors of ρ' at distance t is $d^{O(\log \frac{1}{\epsilon})}$ which is polynomial in $\frac{1}{\epsilon}$. The complexity of determining the neighbors is polynomial because G is an explicit expander construction. Therefore we can reduce the error to $\frac{1}{\text{poly}}$ from a constant using polynomial time and the same randomness as the original algorithm.

3 Randomness Efficient Amplification

The idea of this second application is to reduce the error further by using a little additional randomness.

Given a random algorithm R which uses r random bits and errs with probability $\epsilon_0 \leq \frac{1}{3}$, it can be transformed into an equivalent algorithm R' which uses $r + O(\log \frac{1}{\epsilon})$ random bits with error $\leq \epsilon$ by calling R only $O(\log \frac{1}{\epsilon})$ times in total. Notice this construction allows us to acheive exponentially small error ($\frac{1}{2^n}$) in polynomial time with only slightly more randomness.

Such a transformation can be done trivially using $O(r \log \frac{1}{\epsilon})$ bits by sampling $\log \frac{1}{\epsilon}$ random strings, but does not achieve the additive result suggested. In order to improve on the trivial result, we perform a variation on approach in the previous application.

Again, pick $\rho' \in \{0,1\}^r$ uniform at random. Consider all ρ on a random walk of length t starting at ρ'. Run R on all such ρ and take the majority vote.

The idea here is that we can do a better job than in the previous application because there is more randomness to work with and neighbors at a further distance can be visited (because only one neighbor at each distance is visited).

3.1 Analysis

There is one key lemma that will allow us to bound the error of this approach:

Lemma 2. *Let P be a projection on those ρ for which R errs, then for any vector x:*

$$\|PAx\|_2 \leq \sqrt{\epsilon_0 + \lambda^2}\|x\|_2. \tag{13}$$

Proof. Consider the representation of $x = x^{\|} + x^{\perp}$ in the eigenbasis with respect to u as stated earlier. Then by the triangle inequality:

$$\|PAx\|_2 \leq \|PAx^{\|}\|_2 + \|PAx^{\perp}\|_2. \tag{14}$$

Using the facts that the bad set is small, P projects onto the bad set, and that $x^{\perp}$ contracts by at least λ:

$$\begin{aligned} \|PAx^{\|}\|_2 = \|Px^{\|}\|_2 \leq \sqrt{\epsilon}\|x^{\|}\|_2, \\ \|PAx^{\perp}\|_2 \leq \|Ax^{\perp}\|_2 \leq \lambda\|x^{\perp}\|_2. \end{aligned} \tag{15}$$

Substituting back into the original equation we have:

$$\|PAx\|_2 \leq \sqrt{\epsilon_0}\|x^{\|}\|_2 + \lambda\|x^{\perp}\|_2 = (\sqrt{\epsilon_0}, \lambda) \cdot (\|x^{\|}\|_2, \|x^{\perp}\|_2)^T \leq \sqrt{\epsilon_0 + \lambda^2}\|x\|_2 \tag{16}$$

The last inequality follows from Cauchy-Schwarz. ∎

This lemma can be used to bound the error probability of R'. Consider the probability that R' errs, this is the same as the probability that at least half of the steps in the random walk fell in the set where R errs.

$$\begin{aligned}
\Pr[R' \text{ errs}] &= \Pr[\text{At least } \frac{\text{t}}{2} \text{ of } \rho \text{ fall in the set on which R errs}] \\
&\leq \sum_{B\subseteq[t],|B|\geq\frac{t}{2}} \Pr[(\forall i \in B) i^{\text{th}} \text{ step lies in the bad set for R}] \\
&= \sum_{B\subseteq[t],|B|\geq\frac{t}{2}} \|M_t A M_{t-1} A ... M_2 A M_1 A u\|_1 \\
&\leq \sum_{B\subseteq[t],|B|\geq\frac{t}{2}} \sqrt{N}\|M_t A ... M_1 A u\|_2 \\
&\leq \sum_{B\subseteq[t],|B|\geq\frac{t}{2}} \sqrt{N}\left(\sqrt{\epsilon_0+\lambda^2}\right)^{|B|}\|u\|_2 \\
&= \sum_{B\subseteq[t],|B|\geq\frac{t}{2}} \left(\sqrt{\epsilon_0+\lambda^2}\right)^{|B|} \\
&\leq 2^t \cdot \left(\sqrt{\epsilon_0+\lambda^2}\right)^{\frac{t}{2}} \\
&= (4\sqrt{\epsilon_0+\lambda^2})^{\frac{t}{2}} \leq \epsilon.
\end{aligned} \tag{17}$$

The second line upper bounds the actual probability because it is over counting the bad strings. This probability is rewritten as a matrix in the third line, where $M_i = P$ if and only if $i \in B$ and $M_i = I$ otherwise. The fifth line follows from repeated applications of Lemma 2. A constant number of iterations can decrease $\sqrt{\epsilon_0+\lambda^2}$ to less than $\frac{1}{4}$, making $|B| = \frac{t}{2}$ maximize the RHS. We can use the deterministic amplification procedure to acheive this. If $4\sqrt{\epsilon_0+\lambda^2} < 1$ walking for $t = O(\log\frac{1}{\epsilon})$ steps will give error less than ϵ. This procedure uses r random bits to pick the starting vertex, and $\log d$ bits for each of the $\log\frac{1}{\epsilon}$ steps in the random walk for a total of $r + O(\log\frac{1}{\epsilon})$ random bits for R'.

4 Other Results

A stronger result which considers the variance of random walks is known a the Expander Chernoff Bound. It states that if you take a random walk that the fraction of times the walk lands in the bad set does not vary much from the expected number. Let X_i be an indicator variable that indicates the event that the i^{th} step lies in some set $A_i \subseteq V$. Then the probability that the walk varies from the the expected number of steps in the bad sets can be written as:

$$\Pr[\sum_{i}^{t}(X_i - \mu(A_i)) \geq at] \leq e^{-b(1-\lambda)a^2 t}. \tag{18}$$

where $a \geq 0$ and b is some universal constant. The probability that the walk varies from expected for a constant a decreases exponentially as t increases. This inequality reduces to the

standard Chernoff Bound if G is a complete graph ($\lambda(G) = 0$ because $\text{rank}(G) = 1$ and all the X_i are independent).

5 Next Time

Next lecture we will discuss pseudorandom generators for space-bounded computation, as another application of expanders. This application leverages the power of being able to pick one vertex and a neighbor at random instead of picking two vertices uniformly at random. We will apply this procedure recursively to construct pretty good (though perhaps not the best) pseudorandom generators. Pseudorandom number generation is the dual of amplification. In pseudorandom number generation we will try to reduce the amount of randomness used while not making the error grow by too much more.

CS 810: Complexity Theory 2/20/2007

Lecture 14: Space-Bounded Derandomization

Instructor: Dieter van Melkebeek Scribe: Jake Rosin

Last time we used expanders to reduce the error of probabilistic algorithms by increasing randomness by only a small amount. Today we attempt the opposite: reducing the amount of randomness required with a bounded increase in error. This lecture focuses on derandomization in a space-bounded setting; in the next lecture we will look at derandomization under time-bounds.

1 Pseudorandom Generators

Definition 1. *An ε-PRG for a class $\mathcal{A}$ of algorithms is a collection $(G_r)_r$ of deterministic procedures where $G_r : \{0,1\}^{\ell(r)} \to \{0,1\}^r$ such that for all $A \in \mathcal{A}$:*

$$(\forall^\infty x) \quad |A(x, U_r) - A(x, G_r(U_{\ell(r)}))|_1 < 2\varepsilon \tag{1}$$

Where r is the number of random bits A uses on x, U_n denotes n bits taken from the uniform distribution, and $\forall^\infty$ means "for all x except finitely many."

Note that if A is a decision algorithm, (1) is equivalent to:

$$(\forall^\infty x) \quad \left[|\Pr_{\rho \leftarrow U_r}[A(x,\rho) = 1] - \Pr_{\sigma \leftarrow U_{\ell(r)}}[A(x, G_r(\sigma)) = 1]| < \varepsilon\right] \tag{2}$$

There are three important parameters to the above definition.

- Error ε: the deviation from the original randomized algorithm. For example, if the original algorithm has a probability of error $\frac{1}{3}$ and $\varepsilon = \frac{1}{6}$, the probability of error for the new algorithm will be $< \frac{1}{2}$. We want ε to be small, but it suffices that it be "small enough" given the amplification techniques discussed in previous lectures.
- Seed length $\ell(r)$: the number of random bits required as input to the pseudorandom generator. We want this small.
- Complexity: measured in terms of the output length r. We want PRGs with low complexity so that using them to generate random bits does not increase the total cost of running a randomized algorithm by too much.

2 Uses of PRGs

Pseudorandom generators can be used to reduce the amount of randomness required to run a randomized algorithm. As a side effect they can reduce the complexity of a deterministic simulation of a randomized algorithm, by explicitly computing the probability of acceptance over the set of all possible PRG seeds $\ell(r) < r$. Namely, if G is a $\frac{1}{6}$-PRG for BPTIME(t) computable in DTIME(t'), then

$$\text{BPTIME}(t) \subseteq \text{DTIME}(2^{\ell(t)} \cdot (t'(t) + t)) \tag{3}$$

This is by cycling over all random seeds, running the algorithm on the output of G for each, and outputting the majority answer. For each seed value, the random string must be generated, taking $t'(t)$ time, and the algorithm must be run, for an additional t steps. Since this enumerates all possible seeds and the cumulative error is $< \frac{1}{2}$, a majority vote provides the correct answer.

Similarly, if G is a $\frac{1}{6}$-PRG for BPSPACE(s) computable in DSPACE(s'), then

$$\text{BPSPACE}(s) \subseteq \text{DSPACE}(\ell(2^s) + s'(2^s) + s) \tag{4}$$

Given a PRG computable in polynomial time t' with logarithmic seed length $\ell(t)$, BPP $\subseteq$ P. Similarly given a PRG with logarithmic seed length that runs in log space, BPL $\subseteq$ L. Such pseudorandom generators are not known to exist, but this is an approach used to attempt to prove the containments.

3 Space-Bounded Derandomization

Although we do not yet know how to construct a log space computable PRG with $O(\log r)$ seed length, there are nontrivial constructions approaching this goal. We now present a construction based on expanders.

Theorem 1. *There exists an ε-PRG for* BPSPACE(s) *with*

$$\ell(r) = O(\log \frac{r}{s} \cdot (s + \log \frac{1}{\varepsilon})) \tag{5}$$

computable in space $O(\ell(r))$.

Corollary 1. *There is a $\frac{1}{6}$-PRG for BPL with $\ell(r) = O(\log^2 r)$ and computable in space $O(\log^2 r)$, thus* BPL $\subseteq$ DSPACE($\log^2 n$).

This was already known, due to BPL $\subseteq$ NC^2, but this theorem shows it can be done with PRGs as well. [1]

The idea behind this proof is dividing a space-bounded randomized computation into 2^k phases. Each phase uses r' random bits, where $r' = \frac{r}{2^k}$. Since the operation of this machine is bounded by space s, s bits must pass from phase to phase.

By pairing these blocks and using an expander to produce their random bits, we can reduce the overall level of randomness used by the machine. Consider an expander with degree d and $2^{r'}$ vertices. We let $G_{2r'}$ produce $2r'$ pseudorandom bits by choosing a vertex in the expander at random, then moving to a random neighbor (this is equivalent to selecting an edge at random and using its endpoints). $G_{2r'}$ requires a seed length of $r' \log d$ random bits for each block pair; if this is $< 2r'$ we have reduced the amount of randomness. This process is diagrammed in Figure 1.

If the expander used is good enough, the output from the modified block pair will not differ greatly from the output of the original. We rely on the expander mixing lemma to prove this.

Call the distribution of input (output resp.) states to a block pair S_{in} (S_{out}) and the random inputs to the pair ρ_{left} and ρ_{right}. There are two distributions to consider for (ρ_{left},ρ_{right}):

[1] An alternative construction can be used to show that BPL $\subseteq$ DSPACE($\log^{1.5} n$) which is the best known bound.

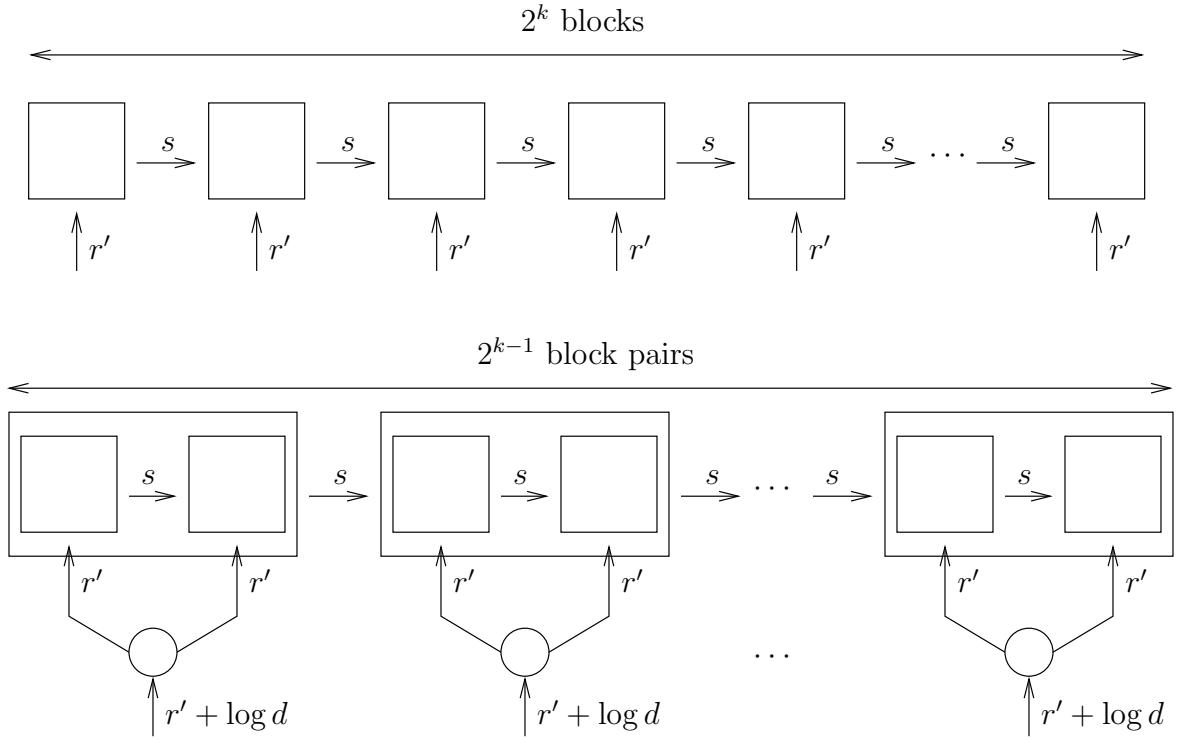


Figure 1: Dividing computation into blocks, with s bits passing between each block. The original computation is shown above, and below it is shown with random bits of adjacent blocks coming from picking adjacent vertices in an expander.

- Random: $U_{2r'}$ - the original randomized input.
- Pseudo-random: $G_{2r'}(U_{r'}, U_{\log d})$ - output from our expander. Note than $G_{2r'}(\rho, \sigma) = (\rho, \sigma\text{-th neighbor of } \rho \text{ in the expander})$.

The following lemma bounds the difference in output distribution between the two scenarios.

Lemma 1. *For any distribution S_{in} on s bits where λ is the second largest eigenvalue of the expander,*

$$|S_{out}(S_{in}, U_{2r'}) - S_{out}(S_{in}, G_{2r'}(U_{r'}, U_{\log d}))|_1 \leq 2^s \cdot \lambda \tag{6}$$

We soon prove this lemma, but for now we finish the description and proof of the PRG given the lemma. We first want to bound the difference in output distribution of running the algorithm on purely random bits versus running the algorithm by grouping pairs of blocks and producing the random bits from the expander. Consider hybrid distributions, where D_i is the distribution formed by using the random distribution for the first $2i$ blocks, then switching to the pseudo-random distribution for the remainder. Thus $D_{2^{k-1}}$ is perfectly random, and D_0 is entirely pseudo-random. The difference between these two distributions is the difference between the randomized algorithm and our pseudorandom version. From the triangle inequality and our key lemma we find:

$$|D_{2^{k-1}} - D_0|_1 \leq \sum_{i=1}^{2^{k-1}} |D_i - D_{i-1}|_1 \leq 2^{k-1} \cdot 2^s \cdot \lambda \tag{7}$$

This provides a bound on the error introduced by the first step of the derandomization. The amount of randomness has been reduced from $2r'$ for each block pair to $r' + \log d$, a savings of roughly r' as d is constant. This is not a large savings but note that we have reduced our original block chain to an easier instance of the same problem - one with 2^{k-1} blocks, each taking $r' + \log d$ random bits. These new computational blocks can be paired, with the $r' + \log d$ random bits being generated by the expander as described above. Pairing blocks recursively (as shown in Figure 2) results in a PRG with the following parameters:

- $\varepsilon < 2^k \cdot 2^s \cdot \lambda$. This bound is found by summing all previous errors.
- $\ell(r) = r' + k \cdot \log d$. Each reduction requires an additional $\log d$ random bits.
- $O(\ell(r))$ space complexity. To compute a given output bit of the PRG, we must compute neighbor relations in a series of expanders. Each of these can be computed in linear space, so the amount of space used at the topmost level dominates. Hence the total space used by the PRG is $O(\ell(r))$.

As defined above r and r' are related through $r' = \frac{r}{2^k}$. The important terms in the parameters for this PRG are λ and d. Any constant degree expander will have a constant λ, which will eventually be overshadowed by 2^s, resulting in $\varepsilon > 1$. To grow λ along with s we begin with a constant-degree constant-λ expander and raise it to the t-th power. Allowing multi-edges in this graph results in a simple expression of the new degree and spectral gap, namely $\lambda(G^t) = (\lambda(G))^t$, and $d(G^t) = (d(G))^t$. Since we want the error of our PRG to be less than 2ε we must satisfy

$$2^{k+s} \cdot \lambda_0^t < 2\varepsilon \tag{8}$$

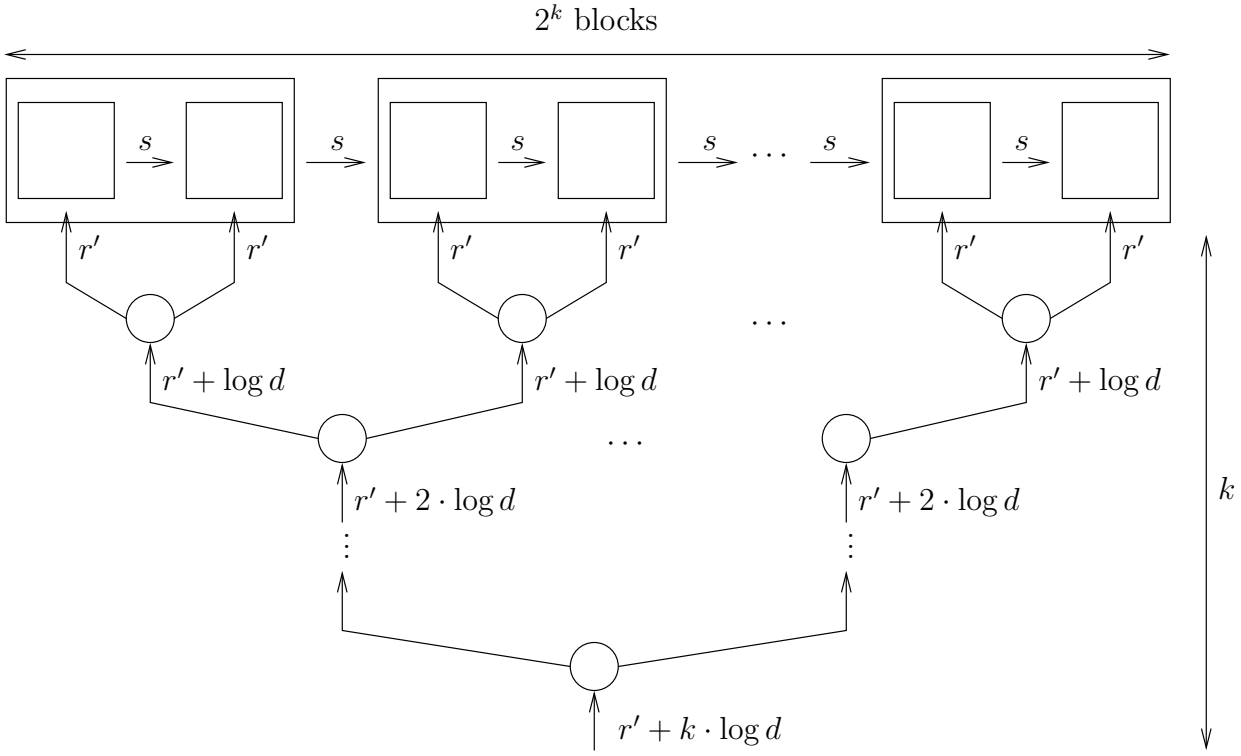


Figure 2: Recursively pairing blocks and applying the expander. k expansions cover the entire computation.

We rearrange to derive the value of t that must be used, and plug in d^t as the degree to determine the seed length

$$t = \Theta(k + s + \log \frac{1}{\varepsilon}) \tag{9}$$

$$\ell(r) = \frac{r}{2^k} + k \cdot \Theta(k + s + \log \frac{1}{\varepsilon}) \cdot \log d \tag{10}$$

We know that $k \leq s$, since there are at most 2^s blocks in our construction and each block uses at least one random bit. Seed length can therefore be defined as

$$\ell(r) = \frac{r}{2^k} + k \cdot \Theta(s + \log \frac{1}{\varepsilon}) \tag{11}$$

The second term grows with k while the first descends. We have remarked before that setting the two terms equal and solving for k gives a result that is minimal to within constant factors. We use $k = \log \frac{r}{s}$. The seed length becomes

$$\ell(r) = O(\log \frac{r}{s} \cdot (s + \log \frac{1}{\varepsilon})) \tag{12}$$

finishing the proof of Theorem 1. All that remains is to prove Lemma 1.

Notice that in our construction each block was treated as a black box. The only connection between blocks was the s bits representing the state of the machine. The algorithm relies on only these s bits being transmitted between blocks, but places no limit on the computations performed by each block individually. This PRG therefore works for any algorithm which can be divided into 2^k blocks with limited communication from block to block, even if each block uses unbounded space.

4 Proof of Lemma 1

We use the expander mixing lemma for this, and so the proof involves finding the appropriate sets to which to apply the lemma.

Proof. First notice that we can prove the lemma by only considering point distributions. Point distributions place all probability in a single point. Any distribution S_{in} is a convex combination of point distributions $X_1, X_2, ..., X_k$. For a randomized process Z, if we consider the quantity

$$\Delta(S_{in}) = \|Z(S_{in}, U_{2r'}) - Z(S_{in}, G(U_{r'}, U_{\log d}))\|_1,$$

we know by the triangle inequality for 1-norm that $\Delta(S_{in}) \leq \Sigma_k \Delta(X_k)$. Hence, we only need to consider a point distribution s_{in}.

We have defined S_{in} and S_{out} as the distributions of inputs to, and outputs from, respectively, a pair of blocks. We now define S_{mid} as the distribution of states when control passes from the first block of a pair to the second. With this distribution we represent the probability of moving from some input state s_{in} to some output state s_{out} using perfectly random bits as follows

$$\Pr[s_{in} \to s_{out}] = \sum_{s_{mid}} \Pr[s_{in} \to s_{mid} \to s_{out}] \tag{13}$$

$$= \sum_{s_{mid}} \Pr[(\rho_{\text{left}}, \rho_{\text{right}}) \in S \times T] \tag{14}$$

Where S and T are defined as:

$$S = S_{s_{in}, s_{mid}} = \{\rho_{\text{left}} | s_{in} \to s_{mid}\} \tag{15}$$

$$T = T_{s_{mid}, s_{out}} = \{\rho_{\text{right}} | s_{mid} \to s_{out}\} \tag{16}$$

In terms of the notation of the expander mixing lemma Equation 14 can be written $\sum_{s_{mid}} \mu(S) \cdot \mu(T)$, as ρ_{left} and ρ_{right} are chosen independently at random.

Under the pseudorandom distribution, the probability of picking an edge between S and T is replaced by $\frac{|E(S,T)|}{dN}$ rather than $\mu(S) \cdot \mu(T)$. Using the expander mixing lemma:

$$\left| \Pr_{(\rho_{\text{left}}, \rho_{\text{right}}) \leftarrow U_{2r'}} [s_{in} \to s_{out}] - \Pr_{(\rho_{\text{left}}, \rho_{\text{right}}) \leftarrow G_{2r'}(U_{\ell(2r')})} [s_{in} \to s_{out}] \right| =$$

$$= \left| \sum_{s_{mid}} \mu(S)\mu(T) - \sum_{s_{mid}} \frac{|E(S,T)|}{dN} \right| \leq \sum_{s_{mid}} \lambda \sqrt{\mu(S)\mu(T)}$$

This inequality represents the probability of generating a single fixed s_{out}. To find the difference in probability over S_{out} we take the sum:

$$|S_{out}(s_{in}, \text{random}) - S_{out}(s_{in}, \text{pseudo})|_1 \leq \sum_{s_{out}} \sum_{s_{mid}} \lambda \sqrt{\mu(S)\mu(T)} \tag{17}$$

By Cauchy-Schwartz we can bound this summation by:

$$\sum_{s_{out}} \sum_{s_{mid}} \lambda \sqrt{\mu(S)\mu(T)} \leq \lambda \cdot \sqrt{\sum_{s_{out}} \sum_{s_{mid}} \mu(S)} \sqrt{\sum_{s_{out}} \sum_{s_{mid}} \mu(T)} \tag{18}$$

Given the definition of S provided above, it should be clear that $\sum_{s_{mid}} \mu(S) = 1$. Similarly, the order of summations in the last term can be reversed, and $\sum_{s_{out}} \mu(T) = 1$. We have therefore demonstrated that:

$$|S_{out}(S_{in}, \text{random}) - S_{out}(S_{in}, \text{pseudo})|_1 \leq \lambda \cdot \sqrt{2^s} \cdot \sqrt{2^s} = 2^s \cdot \lambda \tag{19}$$

□

An alternate proof of Theorem 1 using universal hash functions exists, which shows BPSPACE(s) $\subseteq$ DTISP($2^{O(s)}, s^2$). The proof above is more general, as discussed, due to the sole requirement that the computation can be divided into discrete blocks with limited communication.

The alternate proof shows that BPSPACE(s) $\subseteq$ DSPACE(s^2), which we now already know. By exploiting the properties of that construction, however, the inclusion can be improved to BPSPACE(s) $\subseteq$ DSPACE($s^{\frac{3}{2}}$). Although randomness is conjectured to provide no more than a constant factor improvement in space complexity, an overhead of $\sqrt{s}$ is the best known.

5 Next Lecture

In the next lecture we will look at time-bounded derandomization. In a time-bounded setting no non-trivial derandomizations are known; it is possible (though it would be surprising) that BPP = EXP. However, it is possible to perform non-trivial derandomizations under certain reasonable assumptions. As we will see, if there exists a problem in linear exponential time that requires circuits of linear exponential size then BPP = P. In other words, if non-uniformity doesn't help to speed up computations, neither does randomness.

CS 810: Complexity Theory 2/21/2007

Lecture 15: Time-Bounded Derandomization

Instructor: Dieter van Melkebeek Scribe: Tom Watson

In the last lecture we introduced the notion of a pseudorandom generator (PRG), showed how PRGs can be used for derandomization, and developed a construction of a PRG that fools space-bounded computations. In particular, we developed a PRG with seed length $O(\log^2 n)$ that fools BPL computations. In the time-bounded setting, no nontrivial unconditional PRG constructions are known, but there are constructions that are known to work under certain reasonable complexity-theoretic hypotheses. In this lecture, we will present one such construction, due to Nisan and Wigderson [1]. Under a sufficiently strong (but still reasonable) hypothesis, this PRG allows us to show that BPP = P.

1 Pseudorandom Generators for Time-Bounded Computations

1.1 Distinguishability

Recall that a PRG takes a truly random seed of length $\ell(r)$ and produces a "pseudorandom" string of length r. To be useful, a PRG should be efficiently computable by a deterministic machine. A PRG is called *quick* if it can be computed in time $2^{O(\ell(r))}$, i.e. in time linear exponential in its seed length. We will show that if there exists a language in E with large average-case circuit complexity, then there exists a quick PRG with short seed length that fools time-bounded randomized computations. We will formalize the notion of average-case complexity in Section 2.1. In the next lecture, we will see how to use error-correcting codes to relax our hypothesis from the existence of an average-case hard language to the existence of a worst-case hard language.

The notion of "quickness" may not seem to be efficient enough, and indeed in the cryptographic setting PRGs are typically required to be computable in time polynomial in the seed length. Also, this may not be efficient enough if our goal is merely to reduce the amount of randomness needed by a computation. However, our present focus is full derandomization, achieved by trying all possible seeds and explicitly computing the probability that our algorithm accepts under the pseudorandom distribution. In this setting, we need $2^{\ell(r)}$ time just to look at all possible seeds, and so the factor $2^{O(\ell(r))}$ overhead in computing the PRG's output is just a polynomial overhead in time.

To guage the quality of our PRG construction, we will need measures of how powerful the computations we are trying to fool are allowed to be, and how well we fool these computations. These measures are formalized by the parameters r and ϵ in the following definition.

Definition 1. *An* ϵ-PRG for circuits of size r *is a family of functions* $(G_r)_{r\in\mathbb{N}}$ *where* $G_r : \{0,1\}^{\ell(r)} \to \{0,1\}^r$ *such that for all circuits* C *that take* r *inputs and are of size at most* r*,*

$$\left| Pr_{\sigma\in\{0,1\}^{\ell(r)}}\left[C(G_r(\sigma)) = 1\right] - Pr_{\rho\in\{0,1\}^r}\left[C(\rho) = 1\right] \right| < \epsilon$$

where σ *and* ρ *are chosen uniformly at random.*

Intuitively, this definition means that every circuit of size at most r will have trouble distinguishing whether its input was sampled from the uniform distribution or from the pseudorandom distribution.

There are a few questions about the above definition that naturally present themselves.

(1) Why do we require that our PRG fool circuits when we're really interested in fooling uniform computations? Since BPTIME(t) computations can be mimicked by circuits of size polynomial in t, we will also be able to use such a PRG to fool the uniform computations. We want our PRG to succeed in fooling the computations on all but finitely many inputs, and this is easily captured in the nonuniform setting by constructing a different circuit for each input where the input is hard-wired and the random bits are left as inputs to the circuit. Also, it turns out that our arguments critically use the nonuniformity of the circuits. There are also results that start from a uniform hardness assumption, but those results aren't as strong.

(2) Why do we only require that our PRG fool circuits of linear size? Using the same parameter r for the size of the circuit and its number of inputs will keep the arguments cleaner, and there's no harm in allowing the circuit to take more random bits than it needs. Mimicking a uniform computation with a circuit may yield a circuit that's larger than the number of random bits it needs, but the computation won't be affected by allowing the PRG to provide more random bits.

Recall from the last lecture that a PRG can be used for full derandomization by trying all possible seeds and explicity computing the probability of acceptance of the algorithm under the pseudorandom distribution. The running time becomes the time to run the PRG on a given seed plus the time to run a simulation of the algorithm, times $2^{\ell(r)}$ seeds. Thus if we can get a quick PRG with $O(\log r)$ seed length, then this full derandomization runs in polynomial time, implying that BPP = P. Our ultimate goal is to show that if a sufficiently hard function exists, then such a PRG exists.

1.2 Predictability

When provided with a truly random seed, a PRG produces an output according to some distribution. Since the seed length $\ell(r)$ is ideally much smaller than the output length r, it follows that the pseudorandom distribution can have mass on at most $2^{\ell(r)}$ out of the 2^r strings of length r and will thus be, in some sense, far from the uniform distribution. This is not an issue for us, however; we only want the pseudorandom distribution to be *computationally indistinguishable* from the uniform distribution. The circuits in Definition 1 can be viewed as statistical tests, and we only require that our pseudorandom distribution "pass" certain tests, namely those computable by small circuits.

Our first step will be to show that we can restrict our class of statistical tests even further. We will be interested in circuits that attempt to *predict* the ith bit of a pseudorandom string given the first $i-1$ bits. No predictor exists for the uniform distribution; all circuits succeed in predicting the next bit of the sample with probability exactly $1/2$. In principle, an advantage in predicting the next bit can be gained by the fact that the input is sampled from a pseudorandom distribution; however, intuitively it seems like a lot of computation would be required to do this prediction. We leave it as an exercise to show that a circuit that succeeds with probability at least $1/2 + \epsilon$ in predicting the ith bit from the first $i-1$ bits of a sample from a pseudorandom distribution yields a circuit of essentially the same size that can distinguish between the pseudorandom distribution and the uniform distribution by at least an ϵ amount in the sense of Definition 1. Thus an indistinguishable distribution is also unpredictable. It is conceivable that distinguishing is a much easier task than predicting, but we will now show that, actually, unpredictable distributions are

also indistinguishable from the uniform distribution in a certain sense. Thus we will be able to focus our efforts on constructing a PRG with an unpredictable output distribution.

Theorem 1. *If there exists a circuit C of size at most r such that*

$$\Big|Pr_{\sigma\in\{0,1\}^{\ell(r)}}\big[C(G_r(\sigma))=1\big]-Pr_{\rho\in\{0,1\}^r}\big[C(\rho)=1\big]\Big|\geq\epsilon$$

then there exists an $i\in\{1,\ldots,r\}$ and a circuit P of size at most r such that

$$Pr_{\sigma\in\{0,1\}^{\ell(r)}}\big[P\big((G_r(\sigma))_1,\ldots,(G_r(\sigma))_{i-1}\big)=(G_r(\sigma))_i\big]\geq\frac{1}{2}+\frac{\epsilon}{r}.$$

Proof. Using the distinguisher C, we would like to construct a predictor P. Our first task will be to determine which bit position i will be predicted by P. Consider the hybrid distributions D_i $(i=0,\ldots,r)$ where D_i consists of samples where the first i bits are chosen according to the output distribution of G_r and the remaining bits are chosen uniformly at random. Then D_0 is the uniform distribution on strings of length r, and D_r is the output distribution of G_r. Intuitively, seeing how the circuit C behaves on distributions D_i and D_{i-1} should give us some idea of how good C is at predicting the ith bit from the first $i-1$ bits of a pseudorandom sample, because these distributions are very similar, differing only in the ith component. We can argue this formally. Using the shorthand $Pr_{D_i}[C=1]$ for the probability that C outputs 1 on a sample from distribution D_i, we have

$$\begin{aligned}\epsilon&\leq\Big|Pr_{D_r}[C=1]-Pr_{D_0}[C=1]\Big|\\&=\Big|\sum_{i=1}^{r}\big(Pr_{D_i}[C=1]-Pr_{D_{i-1}}[C=1]\big)\Big|\\&\leq\sum_{i=1}^{r}\Big|Pr_{D_i}[C=1]-Pr_{D_{i-1}}[C=1]\Big|\end{aligned}$$

and thus $|Pr_{D_i}[C=1]-Pr_{D_{i-1}}[C=1]|\geq\epsilon/r$ for some i. We will choose this index i for our predictor. Now we have that

$$Pr\big[C\big((G_r(\sigma))_1,\ldots,(G_r(\sigma))_{i-1},(G_r(\sigma))_i,\rho_{i+1},\ldots,\rho_r\big)=1\big]$$

differs from

$$Pr\big[C\big((G_r(\sigma))_1,\ldots,(G_r(\sigma))_{i-1},\rho_i,\rho_{i+1},\ldots,\rho_r\big)=1\big]$$

by at least ϵ/r, where the probabilities are taken over σ and $\rho_i,\rho_{i+1},\ldots,\rho_r$ chosen uniformly at random. The circuit C appears to be doing a good job of detecting when the ith bit of its input came from the pseudorandom distribution, but it is not quite a predictor yet. In particular, it still takes r input bits, whereas a predictor is only given the first $i-1$ bits of a sample. However, by an averaging argument, there must be some setting $\widetilde{\rho}_{i+1},\ldots,\widetilde{\rho}_r$ to the inputs $\rho_{i+1},\ldots,\rho_r$ such that

$$Pr\big[C\big((G_r(\sigma))_1,\ldots,(G_r(\sigma))_{i-1},(G_r(\sigma))_i,\widetilde{\rho}_{i+1},\ldots,\widetilde{\rho}_r\big)=1\big]$$

differs from

$$Pr\big[C\big((G_r(\sigma))_1,\ldots,(G_r(\sigma))_{i-1},\rho_i,\widetilde{\rho}_{i+1},\ldots,\widetilde{\rho}_r\big)=1\big]$$

by at least ϵ/r, where the probabilities are taken over σ and ρ_i chosen uniformly at random. We can hard-wire these inputs without affecting the circuit size. Note that we are critically using the fact that we are dealing with nonuniform circuits, and so we can handle each value of r separately.

Now for some bit b, our circuit is at least ϵ/r more likely to output b when provided with the first i bits of a sample from the pseudorandom distribution than when provided with the first $i-1$ bits plus a truly random bit. This suggests how to construct a *randomized predictor* P': given the first $i-1$ bits $\pi_1, \ldots, \pi_{i-1}$ of a sample from the pseudorandom distribution, flip a coin to determine ρ_i, evaluate $C(\pi_1, \ldots, \pi_{i-1}, \rho_i, \widetilde{\rho}_{i+1}, \ldots, \widetilde{\rho}_r)$, and if it evaluates to b, assume that our guess was correct and output ρ_i, and otherwise output $\overline{\rho_i}$. More formally,

$$P'(\pi_1, \ldots, \pi_{i-1}) = \rho_i \oplus C(\pi_1, \ldots, \pi_{i-1}, \rho_i, \widetilde{\rho}_{i+1}, \ldots, \widetilde{\rho}_r) \oplus b.$$

The proof of the following claim is left as an exercise.

Claim 1. *$Pr\big[P'\big((G_r(\sigma))_1, \ldots, (G_r(\sigma))_{i-1}, \rho_i\big) = (G_r(\sigma))_i\big] \geq \frac{1}{2} + \frac{\epsilon}{r}$ where the probability is over σ and ρ_i.*

This is exactly the behavior we want from our predictor, but P' still has one undesirable feature—it flips a coin. However, we can again take advantage of the fact that we're in the nonuniform setting and hard-wire ρ_i to some value $\widetilde{\rho}_i$, either 0 or 1, such that the circuit retains its advantage of ϵ/r in predicting the ith bit. This yields a predictor P where $P(\pi_1, \ldots, \pi_{i-1})$ is just

$$\widetilde{\rho}_i \oplus C(\pi_1, \ldots, \pi_{i-1}, \widetilde{\rho}_i, \widetilde{\rho}_{i+1}, \ldots, \widetilde{\rho}_r) \oplus b,$$

which can be expressed as a circuit of the same size as C (possibly with an additional NOT gate, which we assume doesn't increase the size of the circuit). This predictor satisfies

$$Pr\big[P\big((G_r(\sigma))_1, \ldots, (G_r(\sigma))_{i-1}\big) = (G_r(\sigma))_i\big] \geq \frac{1}{2} + \frac{\epsilon}{r},$$

as desired. □

2 The Nisan-Wigderson Construction

2.1 Average-Case Circuit Complexity

We want to construct an ϵ-PRG for circuits of size r. We argued above that if some circuit of size r distinguishes the pseudorandom distribution from the uniform distribution by at least an ϵ amount, then there exists an index i and another circuit of size r that succeeds in predicting the ith bit of a sample from the first $i-1$ bits with advantage at least ϵ/r over the trivial bound of $1/2$. Thus our task is reduced to constructing a PRG such that no small circuit can gain a significant advantage in predicting any bit of a sample from the previous bits.

Intuitively, this suggests that our PRG should generate each bit of its output by applying some hard function, so that we can argue that having a small predictor circuit for some index i would allow us to construct a circuit evaluating the function that was used to generate the ith bit. We now formalize the precise notion of hardness we will need.

Definition 2. *For a language L, the average-case hardness of L at input length m, denoted $H_L(m)$, is the largest s such that no circuit of size at most s can compute L correctly on at least a $\frac{1}{2} + \frac{1}{s}$ fraction of the inputs of length m.*

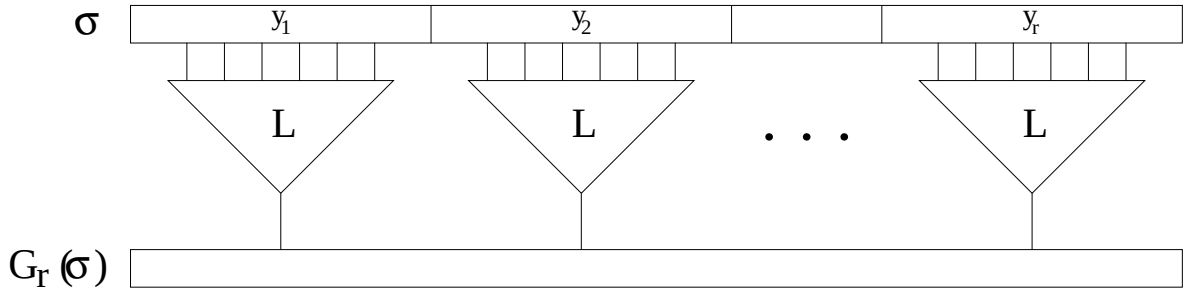
Note that computing L correctly on at least a $1/2$ fraction of the inputs at a given length is trivial—either constant 0 or constant 1 will do the job. An average-case hard function is one that is not only hard to compute exactly, but also hard to compute correctly on noticeably more than half the inputs.

One might wonder why the above definition uses s to refer to both the size of the circuits under consideration and the degree of hardness. The main reason is to keep the number of parameters small, so that our analysis works out cleanly. One might also wonder why we are measuring hardness against *nonuniform* circuits. There is a very good reason for this—our arguments will crucially use this nonuniformity.

Let us gain some intuition about Definition 2. As mentioned in a previous lecture, every predicate on m bits can be computed exactly by a circuit of size at most 2^m, so $H_L(m) < 2^m$ for all L. As s gets smaller, the condition of Definition 2 gets easier to satisfy: the circuits under consideration become more computationally restricted, *and* they're required to compute L on more inputs. Thus $H_L(m)$ serves as a measure of the average-case hardness of L at input length m.

2.2 PRG Construction

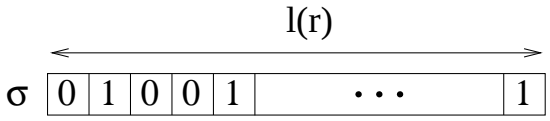
Suppose we have a language L such that $H_L(m) \geq r/\epsilon$. Then no circuit of size r/ϵ, and in particular no circuit of size r, can compute L with probability at least $\frac{1}{2} + \frac{\epsilon}{r}$ over inputs of length m chosen uniformly at random. This suggests the following approach for constructing a PRG: given $y_1, \ldots, y_r \in \{0,1\}^m$ chosen independently and uniformly at random, apply L (viewed as a function producing a bit) to each y_i, yielding an output string of length r. Intuitively, if some bit of the output distribution of this PRG were predictable, then since the samples y_i are chosen independently, such a predictor would have an advantage in computing L. Thus the output distribution of this PRG would be unpredictable and hence, by Theorem 1, indistinguishable from the uniform distribution, as desired.



We will not proceed to formalize this very vague idea because it is seriously flawed. One issue that naturally arises is that computing the output of such a function would require computing L, which is assumed to be hard to compute. However, the length m at which we would be computing L would ideally be much less than the output length r, so the complexity of computing L might not be prohibitive. A much more critical problem is that this construction takes a seed of length mr but only outputs r bits! We want our seed length to be much smaller than r, and certainly not larger. It is trivial to build a PRG when the seed length is at least as large as the output length — we can just output some of the bits of the seed, yielding a uniform output distribution. The reason the above construction requires such a long seed is that all y_i's are to be chosen independently. We

would like to show that by sacrificing some independence of the y_i's, we can drastically reduce the seed length without the quality of the output distribution deteriorating by too much.

We will accomplish this by taking a seed σ of length $\ell(r) > m$ and selecting r subsets S_i $(i = 1, \ldots, r)$ of the bit positions of the seed, and letting $y_i = \sigma|_{S_i}$ be the bits of the seed indexed by S_i. For example, if $S_1 = \{1, 3, \ell(r)\}$ and the seed σ is as illustrated below, then $y_1 = 001$.



The desired subset construction is formalized in the following definition.

Definition 3. *An (m, a)-design of size r over $[\ell] = \{1, \ldots, \ell\}$ is a sequence of subsets $S_1, \ldots, S_r \subseteq [\ell]$ such that $|S_i| = m$ for all i, and $|S_i \cap S_j| \leq a$ for all $i \neq j$.*

We want our PRG output length r to be large, but at the same time we want the pairwise intersections of the S_i's to be small so that the y_i's are "as independent as possble." These two goals are at odds with each other, but the following lemma shows that, in fact, not only do there exist such designs with large r, but the subsets can be efficiently computed.

Lemma 1. *For all r and $m \geq \log r$, there exists an efficiently computable $(m, \log r)$-design of size r over $[\ell]$ where $\ell = O(m^2)$.*

Proof. Assume that m is a prime power. If not, round it up to say the next power of 2. This will affect the value of $\ell = m^2$ by only a constant factor. Now identify $[\ell]$ with $GF(m) \times GF(m)$ and take $S_i = \{(x, q_i(x)) : x \in GF(m)\}$ where q_i is the ith univariate polynomial of degree less than $\log r$. That $|S_i| = m$ for all i is obvious. Then $|S_i \cap S_j| < \log r$ follows from the fact that the polynomials of degree less than $\log r$ represent distinct functions (because $\log r \leq m$—we leave this as an exercise), and so each pair has fewer than $\log r$ points in common. The number of subsets provided is $m^{\log r} \geq r$. Each subset can be computed via simple arithmetic in $GF(m)$, which can be done efficiently enough for our purposes. We will not give a more detailed analysis of the efficiency. □

The condition that $m \geq \log r$ is no problem for us since we will want m to be such that $H_L(m) \geq \frac{r}{\epsilon} \geq r$, and since $H_L(m) \leq 2^m$, we get the constraint $m \geq \log r$ anyway.

We are now in a position to fully specify our PRG. For a given r and m, we can set $\ell(r) = O(m^2)$ and obtain an $(m, \log r)$-design $S_1, \ldots, S_r$ via Lemma 1. Then our PRG G_r will be

$$G_r(\sigma) = L(\sigma|_{S_1})L(\sigma|_{S_2}) \cdots L(\sigma|_{S_r}).$$

It is straightforward to verify that if L is computable in linear exponential time, then this PRG is quick. All that remains is to show that this construction fools circuits of size r if L is sufficiently hard.

Theorem 2. *If $H_L(m) \geq \frac{r}{\epsilon}$ and $\epsilon \leq \frac{1}{r}$, then the above construction is an ϵ-PRG for circuits of size r.*

Proof. We will prove the theorem by contradiction. Suppose that for some circuit C of size r, we have

$$\left| Pr_{\sigma \in \{0,1\}^{\ell(r)}} \left[C(G_r(\sigma)) = 1 \right] - Pr_{\rho \in \{0,1\}^r} \left[C(\rho) = 1 \right] \right| \geq \epsilon.$$

We will show that then $H_L(m) < \frac{r}{\epsilon}$ by exhibiting a circuit of size at most $\frac{r}{\epsilon}$ that solves L at length m on at least a $\frac{1}{2} + \frac{\epsilon}{r}$ fraction of the inputs, thus contradicting the assumed hardness of L.

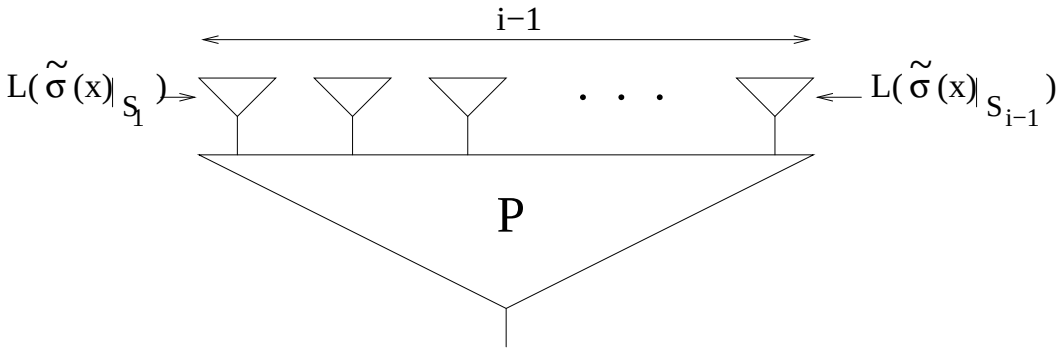
By Theorem 1, there exists an $i \in \{1, \ldots, r\}$ and a circuit P of size at most r such that

$$Pr\left[P\big((G_r(\sigma))_1, \ldots, (G_r(\sigma))_{i-1} \big) = (G_r(\sigma))_i \right] \geq \frac{1}{2} + \frac{\epsilon}{r}.$$

We would like to use P to construct a small circuit that will approximate L well at length m. Intuitively, P seems to be approximating L on input $\sigma|_{S_i}$, and in fact, by another averaging argument we can fix some setting to the bits of σ other than those indexed by S_i such that the predictor P maintains its ϵ/r advantage. Here we are again critically use the fact that we are working with nonuniform circuits. Renaming $\sigma|_{S_i}$ to x and letting $\widetilde{\sigma}(x)$ denote the $\ell(r)$ bits where x fills the positions indexed by S_i and the rest of the positions are fixed as above, we have

$$Pr\left[P\big((G_r(\widetilde{\sigma}(x)))_1, \ldots, (G_r(\widetilde{\sigma}(x)))_{i-1} \big) = L(x) \right] \geq \frac{1}{2} + \frac{\epsilon}{r}$$

where the probability is over x chosen uniformly at random from $\{0,1\}^m$. This is exactly the sort of behavior we would like, but we need to construct a circuit that takes input x, whereas P takes the first $i-1$ bits of G_r's output. We cannot just attach a circuit computing G_r to P, since computing G_r involves computing L exactly. But now we come to the most critical observation of the entire argument: each input to P depends only on at most $\log r$ bits of x, since $|S_i \cap S_j| \leq \log r$ for $j \neq i$, and can thus be computed from x by a circuit of size at most $2^{\log r} = r$. Having the limited pairwise intersections of the subsets in the design is the key to avoiding the inherent complexity of computing L, allowing us to get a contradiction. To the jth input of P, we can attach a circuit of size at most r computing $(G_r(\widetilde{\sigma}(x)))_j = L(\widetilde{\sigma}(x)|_{S_j})$ from x, as illustrated below.



Since $i < r$, we have thus obtained a circuit of size at most $r^2 \leq r/\epsilon$ that succeeds in computing $L(x)$ with probability at least $\frac{1}{2} + \frac{\epsilon}{r}$ over the choice of x. Since $|x| = m$, we conclude that $H_L(m) < \frac{r}{\epsilon}$. We have our contradiction, and the theorem is proved. □

To summarize the above proof, we assumed that our PRG's output distribution was distinguishable from the uniform distribution by a small circuit, used Theorem 1 to obtain a small predictor

circuit for some bit of the pseudorandom distribution, and connected some additional small circuitry to this predictor to convert it into a small circuit that approximated L well, thus showing that L cannot be too hard. We can eliminate the ϵ parameter by setting $\epsilon = 1/r$ to obtain the following clean corollary.

Corollary 1. *If $L \in$ E, m, and r are such that $H_L(m) \geq r^2$, then there exists a quick $\frac{1}{r}$-PRG for circuits of size r with seed length $O(m^2)$.*

We remark that no languages in E are known to satisfy the hardness condition $H_L(m) \geq r^2$ on arbitrary circuits for interesting values of m, say $m = r^{o(1)}$. However, we will see in the next lecture that if there exist languages in E requiring large circuits in the worst case, which is conjectured to be true, then such average-case hard languages do exist.

2.3 Extensions

2.3.1 Constant-Depth Circuits

In a previous lecture we saw average-case hardness results for parity on constant-depth circuits. With such hardness results, the Nisan-Wigderson construction yields unconditional PRGs for constant depth circuits. Supposing we have a distinguisher for the PRG's output distribution, we can obtain a predictor via Theorem 1 without increasing the depth. The final circuit approximating parity is obtained by adding depth-2 circuits computing the first $i - 1$ bits of the PRG's output in DNF or CNF. Thus this PRG succeeds in fooling constant-depth circuits. It has polylogarithmic seed length, and only involves computing the parity of short strings.

2.3.2 Space-Bounded Derandomization

A similar construction to the one discussed today yields PRGs for branching programs that can be used for space-bounded derandomization. In particular, if there exists a language in DSPACE(n) that is average-case hard for linear exponential size branching programs, then this construction yields a (conditional) PRG for space-bounded computations.

2.3.3 Worst-Case to Average-Case Reductions

In the next lecture, we will see how to use error-correcting codes to relax our hardness requirement from average-case hardness to worst-case hardness. We will show that encoding the characteristic sequence of a worst-case hard function with a good error-correcting code can yield an average-case hard function, since a circuit that approximates the "encoded" function could be combined with an efficient decoder to compute the original function on every input. Combining this technique with the results of today's lecture, we can obtain a PRG that fools polynomial-size circuits and has logarithmic seed length, leading to the following result.

Theorem 3. *If there exists a language in* E *requiring linear exponential size circuits, then* BPP = P.

References

[1] N. Nisan and A. Wigderson. Hardness vs. Randomness. *Journal of Computer and System Sciences*, 49:149-167, 1994.

CS 810: Complexity Theory 2/23/2007

Lecture 16: Error-Correcting Codes

Instructor: Dieter van Melkebeek Scribe: Matt Elder

Last lecture, we constructed a family of "quick" pseudo-random generators, based on the assumption that there exists a language $L \in \mathrm{E}$ with high average-case hardness. In this lecture, we extend these results; we show that the worst-case hardness at length m, $C_L(m)$, can be substituted for the average-case hardness at length m, $H_L(m)$.

For most of this lecture, and almost all of the next, we prove this fact via error-correcting codes. Such codes will let us construct a language L' with the average-case hardness very close to the worst-case hardness of a given language L.

1 Circuit Lower Bounds Yield Pseudo-Random Generators

Last time, we found that we could generate efficient pseudo-random generators. More precisely, suppose we have a function $f : \{0,1\}^m \to \{0,1\}$ and an $(m, \log r)$ design $S_1, \ldots, S_r$ over $[\ell]$. We construct the function $G_r : \{0,1\}^\ell \to \{0,1\}$, where $G_r(\sigma) = (f(\sigma|_{S_1}), \ldots, f(\sigma|_{S_r}))$. The key property of G_r is that if $H_{L(f)}(m) \geq r^2$, then G_r is a $(1/r)$-PRG for circuits of size at most r. We also demonstrated a very efficient design when $\ell = O(m^2)$, such that G_r is "quick" - that is, in time linear-exponential in ℓ if $L(f) \in \mathrm{E}$.

If a hard language exists that allows us to construct such a pseudo-random generator, and we have a BPP algorithm A for a problem, then we can use the following deterministic algorithm A' to solve the same problem.

1. Construct the PRG G_r, which takes a random seed of length ℓ and outputs a pseudo-random string of length r.
2. For every seed $\sigma \in \{0,1\}^\ell$, run A, replacing its random bits with $G_r(\sigma)$.
3. Accept if A accepts on most seeds; reject otherwise.

If the complexity of A is small enough that the PRG G_r is known to fool it, then A' is a deterministic algorithm for the problem that A solves. Thus, we can get the following results:

1. If there exists a language L in E such that $H_L(m) \geq m^{\omega(1)}$, then we can build G_r that takes $r^{o(1)}$ input bits, so BPP $\subseteq$ SUBEXP.
2. If there exists a language L in E such that $H_L(m) \geq 2^{m^{\Omega(1)}}$, then we can build G_r that takes $(\log r)^{O(1)}$ input bits, so BPP $\subseteq$ DTIME$(n^{(\log n)^{O(1)}})$.
3. If there exists a language L in E such that $H_L(m) \geq 2^{\Omega(m)}$, then we can build G_r that takes $O(\log r)$ input bits, so BPP $\subseteq$ P.

In implication 3, we need a $(c \log r, \log r)$ design with $\ell = O(c^2 \log r)$ to show that we can build a G_r that takes $O(\log r)$ input bits. We assert that such a design exists, but omit its construction here. As an aside, implication 3 implies the following corollary.

Corollary 1. *BPP* $\subseteq$ *ZPP*$^{\mathrm{NP}} \subseteq \Sigma_2^{\mathrm{P}} \cap \Pi_2^{\mathrm{P}}$.

Proof. By definition, a language is in BPP if it has a poly-time, bounded-error, randomized algorithm. As we've just shown, such algorithms can be derandomized using any function $f\colon \{0,1\}^m \to \{0,1\}$ in E that has hardness $H(f) \geq 2^{\Omega(m)}$.

So, consider χ_f, the characteristic string of the function f. The length of χ_f is 2^m. By a counting argument on the possible characteristic strings of f, we know that most functions on m bits have hardness $H(f) \geq 2^{\Omega(m)}$. So, given a BPP algorithm A, consider the following algorithm in $\mathrm{ZPP}^{\mathrm{NP}}$:

1. Pick a random function $f\colon \{0,1\}^{O(\log r)} \to \{0,1\}$
2. Ask the NP oracle if there exists a circuit of size less than r that computes f. Because f is a function over $O(\log r)$ bits, this is an NP predicate. If there is a small circuit for f, halt and return "?".
3. Otherwise, construct a pseudo-random generator G built from the hard function f. Run the derandomized version of A using G. Accept if it accepts and reject if it rejects.

Since most candidates for f are hard languages, this algorithm halts and fails with low probability; in all other cases, it outputs the correct answer, so this is a $\mathrm{ZPP}^{\mathrm{NP}}$ algorithm. So, if a language can be solved by an algorithm in BPP, then it can be solved in $\mathrm{ZPP}^{\mathrm{NP}}$. Thus, $\mathrm{BPP} \subseteq \mathrm{ZPP}^{\mathrm{NP}}$. □

Next we discuss the converse of these implications; instead of showing that hard languages imply PRGs, we show that PRGs imply hard languages.

2 Pseudo-Random Generators Yield Circuit Lower Bounds

Theorem 1. *If there exists a* ϵ*-PRG* G *computable in* E *that fools circuits of size at most* r*, then there exists a language* $A \in \mathrm{E}$ *with circuit complexity greater than* r*.*

Proof. The idea of this proof is that, given a PRG G, we can construct a language that differentiates between the distribution that G generates and the uniform distribution. Since G is a PRG, this language must have a high circuit complexity.

First, note that the length of the output of G must be larger than the length of its input, and $\epsilon < 1$; the existence of G otherwise is trivial. By only looking at the first $\ell + 1$ output bits of G, we can assume that G takes ℓ bits to $\ell + 1$ bits. Let A be the language $\left\{ G(\alpha) \mid \alpha \in \{0,1\}^{\ell} \right\}$.

Suppose circuit C decides language A. Then,

$$\mathbf{Pr}_{\sigma \in \{0,1\}^{\ell}} \left[C\left(G(\sigma)\right) = 1 \right] = 1, \text{ and}$$
$$\mathbf{Pr}_{\sigma \in \{0,1\}^{\ell+1}} \left[C(\sigma) = 1 \right] \leq \frac{1}{2}.$$

Above, the first line holds by the definition of C. The second line holds because G maps its 2^{ℓ} inputs to no more than 2^{ℓ} distinct outputs. Together, these statements show that C differentiates between the distribution generated by G and the uniform distribution. Thus, since G is known to be undifferentiable by any circuit of size no more than r, the circuit C has size greater than r. This is true for any circuit that decides A, so the language A has circuit complexity greater than r.

To determine if $x \in A$, the naive algorithm is to compute $G(\sigma)$ for all seeds σ and see if x is the output of one of these. As we assume $G \in \mathrm{E}$ and must cycle over 2^ℓ seeds, this computation takes time $2^{O(\ell)}$ - showing that $A \in \mathrm{E}$. □

So, we now have implication both ways: there exist quick PRGs that fool circuits of size r iff there exist functions in E that have circuit complexity greater than r.

3 Error-Correcting Codes

Informally, an *error-correcting code* is a function from a set of *information words* to a set of *codewords* such that, given a codeword with some not-too-large fraction of its bits flipped, the information word can be retrieved. Thus, error-correcting codes encode information in a manner robust against a certain amount of error.

We will use error-correcting codes to relate a language L' that has high average-case circuit complexity to a language L with high worst-case circuit complexity. Suppose that $X_{L|_m}$ is the characteristic sequence of L on inputs of length m, that is, the i^{th} bit of $X_{L|_m}$ is 1 iff the i^{th} m-bit word is in the language L. Now, for some m', let L' be the language whose characteristic sequence on inputs of m' bits is the codeword corresponding to the information word $X_{L|_m}$ in some error-correcting code.

We would like to be able to conclude that if L is worst-case hard for circuits of size s, then L' is average-case hard for circuits of slightly smaller size. The natural way to try to prove this is the following. Suppose we have a circuit C of size s' that computes L' on a large fraction of inputs. Given an input x that we want to compute L on, we want to use C in addition to the decoding procedure for the error-correcting code to compute $L(x)$. The goal is to accomplish this with only a small increase in the size of the circuit. An examination of this process (to be carried out in the next lecture) reveals that the error-correcting code needs to satisfy the following properties:

1. The ECC can handle error rates as high as $1/2 - \epsilon$ for small ϵ.
2. The ECC employs *local decoding*; that is, a single bit of the information word can be derived by examining only a small portion of the codeword.
3. The ECC can encode in polynomial time. (This way, $L \in \mathrm{E}$ implies $L' \in \mathrm{E}$, and $m' = cm$ for some positive constant c.)

Of course, to usefully discuss these properties, we must formally define error-correcting codes. We give appropriate definitions and then consider a number of error-correcting codes to see if we can find one that meets the above goals.

Definition 1. *An (N, K, D)* error-correcting code *(ECC) over the alphabet Σ is a function $E : \Sigma^K \to \Sigma^N$ such that for all distinct $x_1, x_2 \in \Sigma^K$, the Hamming distance[1] between $E(x_1)$ and $E(x_2)$ is greater than D.*

Suppose we process a word $x \in \Sigma^K$ with an ECC, producing $y = E(x)$. Suppose that uncontrolled conditions flip some bits of the code word at random, producing the *received* word $z = \text{perturb}(y)$ and destroying our memory of y. If fewer than D bit flips have occurred between

[1]The Hamming distance between two binary strings (of equal length) is the number of bits in which they differ.

y and z, then z cannot be any legal codeword, so we can detect that bits have been changed. If fewer than $\lfloor (D-1)/2 \rfloor$ bit flips have occurred, then in terms of Hamming distance, z is closer to y than it is to any other legal codeword, so it can be corrected.

The parameters of an ECC that we wish to optimize are:

- The *relative distance* $d = D/N$, which we want to make large.
- The *rate* K/N, which we want to make small.
- The complexity of encoding and decoding, which we want to keep low.

The rate and relative distance are, in a sense, opposed to each other. For example, a very small rate restricts the minimum Hamming distance between two codewords, as it leaves little extra space in Σ^N.

Many of the codes we are interested in belong to a general class of ECCs, called linear error-correcting codes. Intuitively, each codeword bit is a linear function of the message bits. This is define formally as follows.

Definition 2. *An ECC is* linear *if its range is a linear subspace with rank K of $GF(q)^N$ for some prime power q. We designate a linear ECC with square brackets, thus: $[N, K, D]_q$.*

A linear ECC has some nice properties, such as having a generator matrix, which we will not further discuss here. Next, we give several examples of linear ECCs.

3.1 Hadamard Code

The Hadamard code, with $q = 2$, takes the information word $a \in \{0,1\}^K$ to the codeword $(a \cdot x)_{x \in \{0,1\}^K}$. That is, the concatenation of the dot products of a and x across all possible binary words x of length K. For this code, $N = 2^K$ and $d = 1/2$. This relative distance is very good, but the rate is very bad.

Consider how we might decode a received word, after it has been encoded and mangled. Let $r(x)$ be the piece of the received word in position x. Suppose that $\Pr[r(x) = a \cdot x] \geq 3/4 + \epsilon$. If we want the bit a_i, then we can look at two values of x that differ at bit i. We pick x uniformly at random, and look at this bit as well as the bit $x \oplus e_i$. Since each of x and $x \oplus e_i$ are uniformly random:

$$\begin{aligned}
&\Pr[r(x) \oplus r(x \oplus e_i) = a_i e_i] \\
\geq &\Pr[r(x) \text{ is correct}] \Pr[r(x \oplus e_i) \text{ is correct}] \\
\geq &(\frac{3}{4} + \epsilon)^2 \geq \frac{1}{2} + 2\epsilon.
\end{aligned}$$

Thus, the probability that we can decode a single bit by examining just two others is greater than $1/2$. So, as long as the error rate of transmission is slightly below $1/4$, we can correct all errors. Since the relative distance of any ECC is no more than $1/2$, the error-correction rate that this algorithm yields is arbitrarily close to the best rate we can ever have.

Notice that the Hadamard code fails to meet our original goals: although local decoding is possible, the encoding (and therefore time to encode) is exponentially long, and we can handle error rates only up to $1/4$.

3.2 Reed-Solomon Code

The Reed-Solomon code transforms the information word $a \in GF(q)^K$ to the codeword $(P(x))_{x \in GF(q)}$, where P is the polynomial of degree $K-1$ whose coefficients are the "digits" of a (i.e. $a = (a_0, a_1, ..., a_{K-1})$ and $P(x) = a_0 + a_1 x + ...a_{K-1}x^{K-1}$). This code relies on the fact that different polynomials of degree $K-1$ can have at most $K-1$ points of intersection, meaning the distance between distinct codewords is at least $N-K$.

The parameters of the Reed-Solomon code are $N = q$, $K \leq q$, and $d \geq 1 - K/N$. Because the Reed-Solomon code yields an "adjustable" trade-off between the rate and the relative distance, and is fairly easy to encode and decode, it's heavily used in many communications and data-retrieval applications.

But can we use the Reed-Solomon code for the current purposes - does it meet the three goals stated earlier? By setting $K = N2\epsilon$, the code can handle errors up to $\frac{1}{2} - \epsilon$, and both encoding and decoding can be performed in polynomial time. However, decoding is very nonlocal - essentially the entire received word must be examined to retrieve a single bit of the message.

3.3 Reed-Müller Code

The Reed-Solomon code fails for our purposes because the decoding procedure is inherently non-local. The Reed-Müller code corrects this problem by using multi-variate rather than univariate polynomials. We will see in a moment how to take advantage of multi-variate polynomials to perform local decoding. First, consider the representation of the message. In the Reed-Solomon code, we represent the message as the coefficients for a polynomial. For the Reed-Müller code, we will think of the message as an m-variate polynomial with individual degrees less than s. We could use a similar encoding as was used for the Reed-Solomon code: interpret the message as the coefficients of the polynomial. However, this encoding inherently leads to non-local decoding - to recover the coefficients we need to recover the entire polynomial, meaning we would need to examine a large fraction of the received word bits.

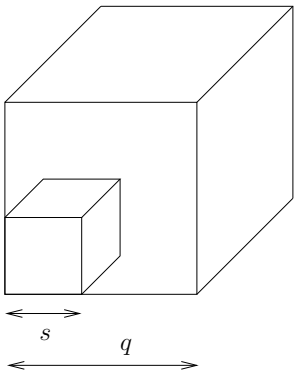


Figure 1: An illustration of the scheme used for the Reed-Müller code for the case of $m = 3$. The message gives the evaluation of a polynomial on a sub-cube of size s^m, and the encoding is the evaluation of an interpolating polynomial on the entire space.

We use a different encoding that will allow local decoding. The information word a is composed of s^m elements from $GF(q)$. We view each element as the evaluation of some m-variate polynomial P on a particular element of $GF(q)^m$ taken from a sub-cube of size s^m. This is illustrated in Figure

1. Encoding is performed by determining the polynomial P from the points given in the message, and then evaluating P on all possible m-tuples over $GF(q)^m$.

It is immediate that for the Reed-Müller code, $K = s^m$ and $N = q^m$. By the Schwartz-Zippel Lemma, the code achieves relative distance $d = 1 - \frac{ms}{q}$.

3.3.1 Encoding

We have specified the correct encoding of the Reed-Müller code above, but it is not immediate that this can be done efficiently: we need to determine the unique polynomial P with each individual degree less than s that interpolates a given set of values on a sub-cube of size s^m. Let $a_i \in GF(q)$ denote the i^{th} value of a. Let ϕ be some invertible map between the integers 1 through s^m and a subcube of $GF(q)^m$ of size s^m. We define want to determine P so that $P(\phi(i)) = a_i$.

We show that we can construct P by building polynomials that interpolate each element of the sub-cube. The polynomial δ_c is defined for an m-variable constant vector c, which is inside the s^m-subcube. We want $\delta_c(c) = 1$ and $\delta_c(x) = 0$ for all other values x inside the s^m-subcube. We can construct δ_c as follows:

$$\delta_c(x) = \alpha \prod_{i=1}^{m} \prod_{\substack{j \neq c_i \\ 0 \leq j < s}} (x_i - j).$$

Here, α is just a normalization constant; we use this so that $\delta_c(c) = 1$. For all values of $x \neq c$ in the s^m-subcube, there is some index i such that $x_i \neq c_i$. By the construction, there must then be a multiplicand in $\delta_c(x)$ of the form $x_i - x_i$, so $\delta_c(x) = 0$. Note that the degree of δ_c in each variable is less than s.

So, we can compose the desired polynomial P as a linear combination of the polynomials δ_c:

$$P(x) = \sum_{i=1}^{s^m} a_i \delta_{\phi(i)}(x).$$

As the degree of each δ_c in each variable is less than s, the same is true of P. We conclude that this formula determines the unique such polynomial interpolating the given points.

Once we have constructed P from the information word a, we produce the codeword b by concatenating the value of $P(x)$ for all values of x in $GF(q)^m$. As an aside, we point out that the the codeword contains an exact copy of the message a (the portion of the codeword where we evaluate P on the elements of the sub-cube that were used to construct P). Codes with this property are called *systematic codes*.

3.3.2 Decoding

To decode, we could query the received word at every position and construct a polynomial that differs from the received word in the fewest number of positions. We do not do this as our goal is to perform as few queries of the received word as possible. Recall that each element of the message corresponds to the evaluation of P on some point in the s^m-subcube. The basic idea is to look at P restricted to a random line through the point corresponding the position in the message we want to decode. P restricted to this line is a univariate polynomial, so if the received word agrees with P on a large fraction of the points on the line, we can recover the values of P on the line (including the point we are interested in). The construction is illustrated in Figure 2.

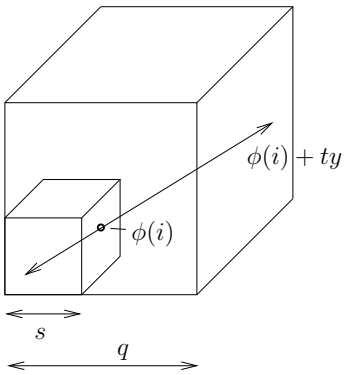


Figure 2: Reed-Müller decoding. To determine $a_i = P(\phi(i))$, construct a line $\phi(i) + ty$ through $\phi(i)$ for a randomly chosen y, then determine the univariate polynomial $P(\phi(i) + ty)$. For $t = 0$, this gives $P(\phi(i))$.

We now formalize the decoding procedure. Suppose r is our received word. Let $r(x)$ denote the value of r in the position corresponding to $x \in GF(q)^m$; and let $P(x)$ denote the analogous position in the correct codeword. To retrieve the i^{th} digit of the message a, we would like to determine the value of $P(\phi(i)) = a_i$. To determine $P(\phi(i))$, we pick a random point $y \in GF(q)^m$, select ms distinct values for $t \neq 0$, and examine $r(\phi(i) + ty)$ at each one. We know that $P(\phi(i) + ty)$ is a polynomial in t of degree at most ms. Suppose $P = r$ for the values we have chosen. In this case, we can determine the polynomial $P(\phi(i) + ty)$, and evaluate it at $t = 0$ to get $P(\phi(i)) = a_i$.

So if $P = r$ on the chosen points, we recover the correct value. We now bound the probability that $P \neq r$ for at least one of the chosen points. Suppose $r(x)$ differs from $P(x)$ in at most γ fraction of locations. As each point we query r on is uniformly distributed, the probability that $r(x) \neq P(x)$ for each query x is at most γ, by our above assumption. So, the probability that $r(\phi(i)+ty) \neq P(\phi(i)+ty)$ for some selected value of t among the $m(s-1)$ values that we selected is at most $(ms)\gamma$. If we assume that $\gamma \leq \frac{1}{3ms}$, the probability we output an incorrect value for a_i is at most $1/3$. Furthermore, we can repeat the above process for different random values of y, and then take the majority vote for the value of a_i. In this way, we can make this decoding algorithm work with probability $1 - \epsilon$ for ϵ as small as we like.

The above analysis only works provided the error rate in transmission is no greater than $1/(3ms)$. Essentially the same techniques can be used with higher error rates, but we stuck to lower error rates in the above to keep the analysis simple. A decoding algorithm for the Reed-Müller code which corrects higher errors is given on Page 14 of [1].

We also point out that the decoding procedure is randomized, whereas we often want to have a deterministic decoding procedure. Since we will have circuits perform the decoding procedure in our worst-case to average-case reduction, requiring randomness will not be a problem - we will be able to hardwire "good" random bits into the circuits.

3.3.3 Parameters

We would like to set the parameters of this code in such a way to meet the three conditions we originally set out to meet. We already have that $N = q^m$, $K = s^m$, and $d = 1 - \frac{sm}{q}$. We want to choose s, m, and q, so that: i) d is close to 1, meaning we can correct even with error rates close to

$\frac{1}{2}$, ii) ms is small, so the number of queries in the decoding procedure is small, and iii) $N = K^{O(1)}$, so the encoding is polynomially long (and thus also polynomial-time computable).

We point out that for d to be positive, we need $sm < q$. Then iii) implies that

$$K^{O(1)} \geq N = q^m \geq (ms)^m = K \cdot m^m,$$

so $m^m \leq K^{O(1)}$. Taking logarithms, we have that $m \log m \leq O(\log K)$, and therefore $m \leq O(\frac{\log K}{\log \log K})$.

ii) combined with the fact that $K = s^m$ means that making m as small as possible will minimize sm, so we set $m = \Theta(\frac{\log K}{\log \log K})$. As $K = s^m$, this means that $s = \Theta(\log K)$.

We have set the parameters so that we get a code with polynomial stretch and requiring only a poly-logarithmic number of queries to locally decode. This is almost good enough for what we want to do. There are two issues that still need to be dealt with: the code is not binary, and (we will see) the distance is not good enough.

3.4 Concatenation of Reed-Müller and Hadamard

To deal with the problem of the Reed-Müller code being a non-binary code, we concatenate it with the Hadamard code. If we start with an $[N, K, d]$ code over an alphabet of q elements, concatenation with the Hadamard code yields a binary $[Nq, K \log q, \frac{d}{2}]$ code. This concatenated code has all of the properties that we want except that the distance is not good enough. Recall that we wanted to be able to handle error rates close to $\frac{1}{2}$. To do this with unique decoding, we need the distance $\frac{d}{2}$ to be close to 1. This is not possible since $d \leq 1$.

In the next lecture, we will see how to get around this problem by relaxing the requirements of uniquely decoding the correct message. The relaxed notion is called *list-decoding*, where we require the decoding procedure to output all messages whose encoding is close to the received word. In the next lecture we will see a list-decoding procedure for the concatenated Reed-Müller/Hadamard code that satisfies all three of our original goals.

References

[1] Madhu Sudan, Luca Trevisian, and Salil Vadhan. *Pseudorandom Generators without the XOR Lemma*. J. of Computer and System Sciences, 62(2):236-266, 2001. Preliminary version: Proc. of 31st ACM STOC, 1999. Accessed at `http://www.cs.berkeley.edu/%7Eluca/pubs/`

CS 810: Complexity Theory 2/26/2007

Lecture 17: Worst-Case to Average-Case Reductions

Instructor: Dieter van Melkebeek Scribe: Tom Watson

In the last lecture we discussed some applications of the Nisan-Wigderson pseudorandom generator, showing that if some language in $L \in \mathrm{E}$ is sufficiently average-case hard for nonuniform circuits, then BPP can be efficiently simulated deterministically, where the efficiency of the simulation depends on the hardness of L. We also introduced the idea of using error correcting codes to allow us to relax our hypothesis from the existence of an average-case hard language to the existence of a worst-case hard language, and we described a local decoding procedure for Reed-Muller codes. Today we will discuss the paradigm of list decoding and describe a local list decoding algorithm for the Hadamard code. Then we will show how local list decoding algorithms can be used to obtain very strong worst-case to average-case hardness reductions in E. Finally, we will introduce the notion of randomness extraction.

1 Worst-Case to Average-Case Reductions via Error Correcting Codes

1.1 Local List Decoding

Suppose there is a language $L \in \mathrm{E}$ such that no family of small circuits can compute L. Our goal is to show that then there exists another language $L' \in \mathrm{E}$ such that not only can no family of small circuits compute L', but no family of small circuits can even succeed in computing L' on noticeably more than half the inputs. Our strategy is to consider the characteristic sequence $\chi_{L|_m}$ at some input length m, which is a string of length 2^m, and encode it using a good binary error correcting code (ECC) to obtain a string of length $2^{m'}$ for some m' that's not too much larger than m. We will then interpret the encoding of $\chi_{L|_m}$ as the characteristic sequence $\chi_{L'|_{m'}}$ of some other language L' at input length m'.

The intuition is that if $L'|_{m'}$ can be solved on noticeably more than half the inputs by a small circuit, then the characteristic sequence of the function computed by this circuit can be viewed as a corrupted version of $\chi_{L'|_{m'}}$. Then the original information word $\chi_{L|_m}$ could be obtained by an efficient decoding procedure, allowing us to solve $L|_m$ on all inputs.

Our ECC needs to have the following properties.

(1) We want to argue that if L' is not sufficiently hard, then we can construct a small circuit computing L. This circuit will be given a string x of length m and required to compute the bit of $\chi_{L|_m}$ corresponding to that string. However, the "received word" $\chi_{L'|_{m'}}$ is exponentially longer than x. This seems to be a problem since traditional decoding algorithms need to look at the entire received word, even to compute a single bit of the information word. We get around this by using a *local decoding* algorithm, which, given the index (x, in our case) of a bit of the information word, computes that bit while only making a few randomized queries to the received word. In the last lecture we described local decoding algorithms for the Hadamard and Reed-Muller codes.

(2) Since we would like L' to be hard to solve on even a little more than half the inputs, we will need to be able to decode a fraction of errors that is as large as $1/2 - \epsilon$ for some small ϵ. This seems problematic since to be able to correct a fraction η of errors, we need the (relative) minimum distance of the ECC to be greater than 2η. This would imply that we need an ECC with minimum distance close to 1, which is a problem since, although we will not argue it, a distance of $1/2$ is the best one can hope for in the case of binary codes. We will get around this by using a *list decoding* algorithm, which, given a received word, computes all information words whose encodings are within a certain distance from it. Since we are dealing with circuits, we will be able to then nonuniformly select which information word is the correct one. Naturally, as the fraction η gets closer to half, the number of codewords within distance η grows, but we will be able to show that this number does not grow too large. We will describe a list decoding algorithm for the Hadamard code today.

(3) Finally, we will require an efficient encoding procedure for our ECC. We will need to show that if L is in E, then L' is also in E, and this will be shown by combining an exponential-time algorithm for L with an efficient encoding procedure for the ECC. In fact, we would like m' to be only a constant factor larger than m. This will allow us to show that the average-case hardness of L' is quite comparable to the worst-case hardness of L. Our ultimate goal is to show that if L has linear exponential worst-case circuit complexity, then L' has linear exponential average-case circuit complexity, since as we saw previously, the existence of such an L' allows us to obtain a quick pseudorandom generator with logarithmic seed length and conclude that BPP = P.

Properties (1) and (2) indicate that we will need an ECC with an efficient *local list decoding* algorithm. We will not give the full details of the necessary construction. All the desired properties can be realized for the concatenation of the Hadamard code with the Reed-Muller code. A local list decoder for the Reed-Muller code can be obtained using the local decoding approach discussed in the last lecture, combined with a list decoder for the Reed-Solomon code. The latter decoder involves a technical procedure for factoring bivariate polynomials. Today we describe a local list decoding procedure for the Hadamard code. The local list decoders for these two codes can easily be combined to form a local list decoder for the concatenation. In Section 1.4, we will show how this leads to the desired worst-case to average-case reduction.

1.2 Error Correcting Code Constructions

Recall that the Hadamard code is a $[2^K, K]_2$ code that encodes an information word $a \in \{0,1\}^K$ as the codeword $((a,x))_{x\in\{0,1\}^K}$. That is, it takes the inner product of a with every bit string of length K and outputs the list of all 2^K results. The minimum distance of this code is $1/2$, which in the case of binary codes is the best distance one can hope for asymptotically. Although it has a very good minimum distance, the Hadamard code has a horrible rate and is of no practical value. However, it is useful in complexity theory. This code can handle up to a $1/4$ fraction of errors if we require unique decoding. This is not good enough to achieve the strong hardness results we're after. We will show that by contenting ourselves with list decoding, we will be able to handle a fraction of errors that is almost half.

The Reed-Solomon code is another useful ECC. It has a minimum distance that is close to 1. However, it is not locally decodable — we need to query at least K positions of the received word in order to reconstruct even one position of the information word. Furthermore, the Reed-Solomon

code requires that the field size be at least as large as the codeword length, which is undesirable since we are interested in codes over $GF(2)$. However, we can handle this by concatenating with a binary code such as the Hadamard code.

The Reed-Muller code is one for which we do have a good local decoding algorithm. Recall that Reed-Solomon encoding is achieved by interpreting the information word as a low-degree univariate polynomial and evaluating it at all points of the field, while Reed-Muller encoding is achieved by interpreting the information word as a low-degree polynomial in ℓ variables (say) and evaluating it at all ℓ-tuples of field elements. Intuitively, we are packing the information into an ℓ-dimensional cube. We can then locally decode by picking a random line through the point in the cube where we want to evaluate the original polynomial, querying the received word at all points along this line, and using a Reed-Solomon decoder to reconstruct a univariate polynomial corresponding to the original polynomial restricted to this line. Using a similar approach together with a list decoder for the Reed-Solomon code, a local list decoder for the Reed-Muller code can be obtained. We will not explore this result in this course.

Finally, concatenating the Reed-Muller code with the Hadamard code yields a code that satisfies our desired properties. Concatenating with the Hadamard code allows us to get a binary code, and the exponential blow-up of the Hadamard code is compensated for by the fact that the field size required by the Reed-Muller code doesn't grow too fast. The locally decodability property is the key for getting a worst-case to average-case reduction The list decodability property is the key for overcoming the upper bound of $1/2$ on the distance of the code in order to acheive very strong worst-case to average-case reductions.

1.3 Local List Decoding of the Hadamard Code

We develop a local list decoding algorithm for the Hadamard code. We will not need to worry about the local decoding aspect; this will be a natural feature of our algorithm. Given a received word $r \in \{0,1\}^{2^K}$ and an error bound η, we wish to find a list of all information words whose encodings are within Hamming distance at most η from r. If $\eta < \delta/2$, where δ is the minimum distance of the code, then this list can contain at most one information word. As η gets larger, the list will also naturally get larger, but we want to show that it does not get too large. Specifically, we wish to be able to handle up to a $1/2 - \epsilon$ fraction of errors in randomized time $poly(\frac{K}{\epsilon})$. Since the received word is of length 2^K, we clearly need random access to it.

Theorem 1. *There is a randomized algorithm that, given random access to a received word $r \in \{0,1\}^{2^K}$, runs in time $poly(\frac{K}{\epsilon})$ and outputs a list of information words that with high probability contains all $a \in \{0,1\}^K$ such that*

$$Pr_{x \in \{0,1\}^K}\left[(a,x) = r(x)\right] \geq \frac{1}{2} + \epsilon.$$

That is, it outputs a list of all information words whose Hadamard encodings are at relative distance at most $1/2 - \epsilon$ from r.

Proof. We focus on obtaining one particular $a \in \{0,1\}^K$ satisfying $Pr_x[(a,x) = r(x)] \geq \frac{1}{2} + \epsilon$. By running the procedure a few more times and concatenating the lists, we can get a list that with high probability contains all information words having the desired level of agreement.

Recall that in the last lecture we described a local unique decoder for the Hadamard code. The idea was to retrieve the ith bit of a by picking a random x and querying $r(x)$ and $r(x + e_i)$ where

e_i is the string with a 1 in the ith position and 0's elsewhere. Since we were assuming that the encoding of a differed from r in at most a fraction of $1/4 - \epsilon$ positions, we were able to conclude by a union bound that with probability at least $1/2 + 2\epsilon$, both x and $x + e_i$ were positions where r was correct, in which case $r(x) + r(x + e_i) = a_i$. By picking several x's independently, and taking the majority vote of the values $r(x) + r(x + e_i)$, we were able to obtain the correct a_i with high probability. In the present settting, however, r may be wrong in a fraction of $1/2 - \epsilon$ positions, so we are only able to conclude that with probability at least 2ϵ, both x and $x + e_i$ are positions where r is correct. Thus obtaining a_i using this idea would require too many samples x. We will now describe a more elaborate approach that uses the power of list decoding to reduce the number of samples needed.

We focus on retrieving one particular component a_i of the information word a. Consider the following idea. Select $x_1, x_2, \ldots, x_t \in \{0,1\}^K$ uniformly at random (for some small t to be determined later), and obtain $2^t - 1$ strings by adding together all possible combinations of the x_j's (except the empty combination). More formally, for the $2^t - 1$ nonzero values of $c \in \{0,1\}^t$, take the string

$$y_c = \sum_{j=1}^{t} c_j x_j.$$

Note that c_j is one bit of c, whereas x_j is a string of length K. Then y_c is just the sum of some of the x_j's, namely those corresponding to the locations of the 1's in c. Now since c is nonzero, it follows that $e_i + y_c$ is uniformly distributed. The proof of the following claim follows since the event under consideration holds whenever $e_i + y_c$ is an index of a position where r agrees with the encoding of a.

Claim 1. *For all nonzero c, $Pr[(a, y_c) + r(e_i + y_c) = a_i] \geq 1/2 + \epsilon$.*

Thus *if we knew the values (a, y_c) for all c*, then we would be able to pick an arbitrary c, query $r(e_i + y_c)$, add (a, y_c) to the result, and conclude that we had a_i with confidence at least $1/2 + \epsilon$, which is much better than the 2ϵ we got in our first attempt. In other words, we are circumventing the inherent unreliability of the two-query approach by assuming we always have the "correct" answer for one of the two queries. We will show later how we can handle this hypothesis using the power of list decoding.

Another issue we need to handle is that we would naturally like to boost our confidence by making not one, but several queries to r. We can just use $e_i + y_c$ for all choices of c, and take the majority vote of the values $(a, y_c) + r(e_i + y_c)$. There are $2^t - 1$ choices for c, and one might wonder if this means we would have to make too many queries. However, it turns out that t can be chosen small enough that this isn't a problem. A more important issue to be alarmed about is the fact that these $2^t - 1$ strings are definitely not fully independent; after all, they were generated using only tK bits of randomness. However, they are *pairwise independent*. To see this, note that if $c_1 \neq c_2$ then there is an index j where they differ. Since c_1 and c_2 are nonzero we have

$$Pr[y_{c_1} = z_1] = Pr[y_{c_2} = z_2] = \frac{1}{2^K},$$

and the equality

$$Pr\big[(y_{c_1} = z_1) \wedge (y_{c_2} = z_2)\big] = \frac{1}{2^K}$$

can be seen by conditioning on the values $x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_t$.

It turns out that pairwise independence is good enough.

Claim 2. *For all i,*

$$Pr[MAJ_i \neq a_i] \leq \frac{1}{2^t \epsilon^2}$$

where $MAJ_i = majority(b_{i,c} : c \neq 0^t)$ and $b_{i,c} = (a, y_c) + r(e_i + y_c)$.

Proof. Let $X_{i,c}$ be the indicator random variable for the event that $b_{i,c} = a_i$, i.e. our guess for a_i is correct when we use c. In order for $MAJ_i \neq a_i$ to happen, it needs to be the case that

$$\frac{1}{2^t - 1} \sum_{c \neq 0^t} X_{i,c} \leq \frac{1}{2}.$$

However, by Claim 1 we know that $E[X_{i,c}] \geq 1/2 + \epsilon$ for all $c \neq 0^t$, and thus $E[\frac{1}{2^t-1} \sum_{c \neq 0^t} X_{i,c}] \geq 1/2 + \epsilon$ by linearity of expectation. It follows that

$$Pr[MAJ_i \neq a_i] \leq Pr\left[\left|\frac{1}{2^t - 1} \sum_{c \neq 0^t} X_{i,c} - E\Big[\frac{1}{2^t - 1} \sum_{c \neq 0^t} X_{i,c}\Big]\right| \geq \epsilon\right].$$

Since the $X_{i,c}$'s are pairwise independent for each i, we can apply Chebyshev's inequality and the fact that every indicator random variable has variance at most $1/4$ to conclude that

$$\begin{aligned} Pr[MAJ_i \neq a_i] &\leq \frac{\sigma^2\big(\frac{1}{2^t-1} \sum_{c \neq 0^t} X_{i,c}\big)}{\epsilon^2} \\ &= \frac{1}{(2^t - 1)^2} \frac{\sum_{c \neq 0^t} \sigma^2(X_{i,c})}{\epsilon^2} \\ &\leq \frac{1}{(2^t - 1)^2} \frac{2^t - 1}{4\epsilon^2} \\ &\leq \frac{1}{2^t \epsilon^2}. \end{aligned}$$

□

Now by a union bound, the probability that there exists an i such that $MAJ_i \neq a_i$ is at most $\frac{K}{2^t \epsilon^2}$. This can be made at most $1/2$ by choosing

$$t = \Theta\Big(\log \frac{K}{\epsilon^2}\Big).$$

To summarize the algorithm up to this point, we first choose $x_1, \ldots, x_t \in \{0,1\}^K$ uniformly at random. We then recover each bit a_i separately (this will allow for local decoding) by forming the $2^t - 1$ queries $r(e_i + y_c)$ corresponding to different c's, for each query guessing that a_i equals $b_{i,c} = (a, y_c) + r(e_i + y_c)$, and taking the majority vote over all these guesses $b_{i,c}$. As argued above, we succeed in recovering the information word a correctly with probability at least $1/2$ over the choice of $x_1, \ldots, x_t$. Since recovering each position a_i involves $2^t - 1 = poly(\frac{K}{\epsilon})$ queries, the entire procedure runs in time $poly(\frac{K}{\epsilon})$, as desired.

However, there is one critical issue we have yet to resolve — the entire procedure assumed we knew the values (a, y_c) for all c, which seems ridiculous since a is what we're trying to find! This is

where list decoding comes in: since our algorithm needs a sequence of values $((a, y_c))_{c \neq 0^t}$, we can try all possibilities for this sequence, run our algorithm for each possibility, and output the list of all information words obtained. Then with probability at least $1/2$, a appears on the list (namely, it appears when we try the correct values for the sequence $((a, y_c))_{c \neq 0^t}$).

There is a problem with this, however: there are $2^t - 1$ different c's, leading to $2^{2^t - 1}$ possibilities and making the running time exponential in $\frac{K}{\epsilon}$. This is easily remedied by recalling that the inner product is *linear*, and so

$$(a, y_c) = (a, \sum_{j=1}^{t} c_j x_j) = \sum_{j=1}^{t} c_j (a, x_j).$$

Thus the $2^t - 1$ values (a, y_c) are uniquely determined by the t values (a, x_j). It follows that we can reduce our list to size $2^t = poly(\frac{K}{\epsilon})$, since we only need to try all possible values for the sequence $((a, x_j))_j$. The overall running time remains $poly(\frac{K}{\epsilon})$.

This explains why we didn't choose $2^t - 1$ strings x independently but rather chose t strings and looked at all combinations of them: with the former approach our list would have been too long — on the order of 2^{2^t} entries — whereas with the latter approach we can get by with a list size of 2^t at the expense of having our $2^t - 1$ samples be only pairwise independent, which as we argued above, is not a big problem. □

As with the local decoding algorithm discussed in the last lecture, the basic idea of the above proof is to try to recover a_i by querying a random location in the received word, querying the location whose index has the ith bit flipped, and XORing the results. In the present setting there are too many errors in the received word for this to be reliable. One key idea is that we can drastically increase the reliability *if we know that one of the two queries is uncorrupted*. Since unique decoding is not required, we can try all possibilities for the "correct" values of these queries. The other key idea is that we can keep the list size small by only choosing a small number of query locations, and deterministically generating the rest of the query locations by adding together all possible combinations. Chebyshev's inequality allows us to conclude that the reliability doesn't deteriorate too much when we do this.

As a corollary to the above result, we note that for all received words r, the number of information words a whose Hadamard encodings agree with r on at least a $1/2 + \epsilon$ fraction of positions is bounded by $poly(\frac{K}{\epsilon})$, the running time of the algorithm.

Finally, we note that the list output by our decoding algorithm may contain information words whose encodings do not have the $1/2 + \epsilon$ agreement with the received word. We can try to weed these out by randomly querying r to ensure that with high probability, r agrees with the encoding in at least a fraction $1/2 + \epsilon - \epsilon'$ locations, for some small ϵ'.

1.4 Worst-Case to Average-Case Reductions

With the Hadamard decoder described in the previous section and a local list decoder for the Reed-Muller code, one can obtain a local list decoder for the concatenation of the two codes. The precise result is stated below, without proof.

Theorem 2. *For each $\epsilon > 0$ and K there exists an error correcting code with the following properties. There is a polynomial-time encoder mapping information words of length K to codewords of length $poly(K)$. The codeword length can be assumed to be a power of 2 when K is. There is*

a randomized algorithm that runs in time poly($\frac{K}{\epsilon}$) and outputs a list of randomized oracle Turing machines $M_1, M_2, \ldots$ that take as input a position in the information word, have oracle access to the received word, and run in time poly($\frac{\log K}{\epsilon}$). These machines have the property that for all received words r and all information words w such that r agrees with the encoding of w in at least a fraction $1/2 + \epsilon$ positions, there exists an i such that M_i^r computes w.

We now show how to use Theorem 2 to get worst-case to average-case reductions.

Theorem 3. *For every $L \in \mathrm{E}$ there exists a language $L' \in \mathrm{E}$ such that*

$$H_{L'}(m) \geq \frac{C_L(m)^{\Omega(1)}}{m^{O(1)}}.$$

Proof. Let L be a language in E. Applying the ECC from Theorem 2 to $\chi_{L|_m}$ yields a string of length $2^{m'}$ for some $m' = O(m)$. We define $\chi_{L'|_{m'}}$ to be this string. We can solve L' in E by taking an input of length m', computing the corresponding length $m < m'$, explicitly writing out $\chi_{L|_m}$, encoding it, and extracting the bit corresponding to our input. Writing out $\chi_{L|_m}$ takes $2^{O(m')}$ since there are 2^m positions, each of which can be computed in $2^{O(m)}$ time since $L \in \mathrm{E}$. Encoding $\chi_{L|_m}$ takes $2^{O(m')}$ time since the ECC from Theorem 2 is polynomial-time encodable. Solving L' incurs an exponential factor blowup in running time, which doesn't take us out of E, but does prevent us from using this technique to get worst-case to average-case reductions for smaller classes.

We will show that

$$H_{L'}(m') \geq \frac{C_L(m)^{\Omega(1)}}{m^{O(1)}},$$

and the theorem will follow from the fact that $m' = O(m)$ (and the fact that $C_L(m)$ is at most exponential, and so changing the input length by a constant factor only makes $C_L(m)$ change by a polynomial factor).

Set $\epsilon = 1/C_L(m)^\alpha$ for some α to be determined later. Suppose there exists a circuit of size s' that, given an input of length m', computes the corresponding bit of $\chi_{L'|_{m'}}$ for at least a $1/2 + \epsilon$ fraction of inputs. Now for some i, the machine M_i^r from Theorem 2 takes an input of length m and outputs the corresponding position of $\chi_{L|m}$ with high probability, i.e. it solves L, provided r is a string of length $2^{m'}$ that agrees with $\chi_{L'|_{m'}}$ on at least a $1/2 + \epsilon$ fraction of positions. This probability may be amplified so that there there is some choice of randomness for which M_i^r solves L on all inputs of length m. By hard-wiring this choice of randomness, we can obtain an oracle circuit of size $(\frac{m}{\epsilon})^{O(1)}$ solving L at input length m. All its oracle gates are queries to $\chi_{L'|_{m'}}$ and can thus be replaced by our hypothesized circuit of size s'. By Theorem 2, this circuit of size $(\frac{m}{\epsilon})^{O(1)} \cdot s'$ computes L exactly provided the oracle subcircuit solves L' on at least a $1/2 + \epsilon$ fraction of inputs, which it does by hypothesis. We conclude that

$$C_L(m) \leq \left(\frac{m}{\epsilon}\right)^\beta \cdot s',$$

for some constant $\beta \geq 1$ (and sufficiently large m). It follows that every circuit of size less than

$$\frac{C_L(m)^{1-\alpha\beta}}{m^\beta}$$

succeeds in computing L' at length m' for less than a

$$\frac{1}{2} + \frac{1}{C_L(m)^\alpha}$$

fraction of inputs. This implies that

$$H_{L'}(m') \geq \min\left(C_L(m)^\alpha, \frac{C_L(m)^{1-\alpha\beta}}{m^\beta}\right).$$

If $1 - \alpha\beta < \alpha$, i.e. $\alpha > \frac{1}{\beta+1}$, then the second term definitely dictates the minimum, so choosing $\alpha < \frac{1}{\beta}$ gives the desired result

$$H_{L'}(m') \geq \frac{C_L(m)^{\Omega(1)}}{m^{O(1)}}.$$

□

Corollary 1. *If there exists a language $L \in$ E with $C_L(m) \geq m^{\omega(1)}$ then there exists a language $L' \in$ E with $H_{L'}(m) \geq m^{\omega(1)}$ and thus there exists a quick PRG with subpolynomial seed length, implying that* BPP $\subseteq$ SUBEXP*.*

Corollary 2. *If there exists a language $L \in$ E with $C_L(m) \geq 2^{m^{\Omega(1)}}$ then there exists a language $L' \in$ E with $H_{L'}(m) \geq 2^{m^{\Omega(1)}}$ and thus there exists a quick PRG with polylogarithmic seed length, implying that* BPP $\subseteq$ DTIME$(n^{\log^{O(1)} n})$*.*

Corollary 3. *If there exists a language $L \in$ E with $C_L(m) \geq 2^{\Omega(m)}$ then there exists a language $L' \in$ E with $H_{L'}(m) \geq 2^{\Omega(m)}$. and thus there exists a quick PRG with logarithmic seed length, implying that* BPP $=$ P*.*

2 Randomness Extraction

We have seen evidence that randomness is not very powerful in terms of reducing the complexity of solving decision problems. We have seen an unconditional pseudorandom generator that fools space-bounded computations, and a conditional pseudorandom generator that fools time-bounded computations under the hypothesis that there exists a language in E requiring linear exponential size circuits. It is conjectured that using randomness can only lead to a polynomial factor savings in time and a constant factor savings in space. This does not mean that randomness is useless in practice. On the contrary, a quadratic speedup achieved with randomness may be very attractive in practice. Additionally, many randomized algorithms are simpler and easier to implement than deterministic algorithms for the same problems.

We now turn to a different question. Most randomized algorithm assume access to a perfect source of unbiased, and more importantly uncorellated, random bits. How do we run such algorithms with access to an imperfect random source? The goal of randomness extraction is to take samples from a weak random source — one where samples may not be uniformly distributed — and generate samples that are close to being uniformly distributed. Such weak random sources will be our models for physical sources of randomness, such as keystrokes or delays over networks.

An *extractor* is an efficient procedure for taking a sample from such an imperfect source and "extracting" the randomness from it, producing an output string that is shorter but much closer

to being uniformly distributed. Such a procedure can be used to run randomized algorithms with weak random sources. The fact that the output distribution of an extractor is only "close" to uniform will have only a small effect on the output distribution of the randomized algorithm.

Before we embark on the task of constructing an extractor, we need to formalize what we mean by the "amount of randomness" contained in our weak random source, and by "closeness to uniform" of the output distribution obtained by applying our extractor to our weak random source. For the latter, we will use the standard measure of statistical distance. For the former, one idea is to use the measure of entropy from physics.

Definition 1. *The* entropy *of a discrete random variable X is*

$$H(X) = E\Big[\log \frac{1}{p_i}\Big] = \sum_i p_i \log \frac{1}{p_i}$$

where the sum is over the range of X, and $p_i = Pr[X = i]$.

However, this measure of randomness does not work in our setting. Indeed, suppose that the range of X is $\{0,1\}^m$ and that for nonzero $x \in \{0,1\}^m$, $p_x = 2^{-(m+1)}$, and the rest of the probability is concentrated on 0^m. Then the entropy measure indicates that X has a fair amount of randomness, but X is useless for simulating a BPP algorithm — if 0^m is in the bad set for a particular input, then the probability of error on that input is greater than half!

Instead, we will require that for X to have a large "amount of randomness", it must be the case that no string is given too much weight. This suggests the following measure.

Definition 2. *The* min entropy *of a discrete random variable X is*

$$H_\infty(X) = \min_i \log \frac{1}{p_i}.$$

Equivalently, $H_\infty(X)$ is the largest value of k such that all outcomes have probability at most 2^{-k} under X.

We will say that a source X with $H_\infty(X) \geq k$ has at least k bits of randomness.

Our goal is to construct extractors such that given a source with min entropy at least k, the output distribution of the extractor is statistically close to the uniform distribution on strings of length as close to k as possible. A good extractor can be obtained by viewing the input sample as the characteristic string of a function and using this function in the Nisan-Wigderson pseudorandom generator construction. We will see more details about this construction in the next lecture.

CS 810: Complexity Theory 2/28/2007

Lecture 18: Randomness Extraction

Instructor: Dieter van Melkebeek Scribe: Jeff Kinne

In the last lecture we briefly introduced the paradigm of randomness extractors. Recall that our analysis of the correctness of randomized algorithms has always assumed a source of random bits that are perfectly uniform and independent. We do not in general have access to such sources in the real world, but we may have access to weak random sources - with some randomness in the source. Randomness extractors are procedures that can be applied to a weak random source to extract out bits that are close to being uniform and independent. We can then run a randomized algorithm on the output of an extractor and have confidence in the answer just as we would if we had access to perfect random sources. Today we formally define randomness extractors, explore some applications, and give a few constructions.

1 Weak Random Sources

Before defining extractors, we quantify what we mean when we say a source has a certain amount of randomness.

Definition 1. *Let X be a random variable on $\{0,1\}^n$. X has* min-entropy *at least k if*

$$(\forall x \in \{0,1\}^n), \Pr[X = x] \leq 2^{-k}.$$

We use $H_\infty(X)$ to denote the min-entropy of X.

We will show in this lecture that given a source with $H_\infty(X) \geq n^{\Omega(1)}$, we can in polynomial time use the source to run polynomial time randomized algorithms. In addition, it can be shown that any source that can be used for randomness extraction, which we define formally in Definition 2, is close to a source with min-entropy $n^{\Omega(1)}$ (see Exercise 2). Hence, min-entropy is the correct notion of randomness for our goal. Therefore, we call a source a *weak random source* if $H_\infty(X) \geq n^{\Omega(1)}$.

There are a number of sources with high min-entropy with simple descriptions.

- *Bit fixing sources.* These are sources where a certain number of bits are fixed adversarially, and the remaining k bits are perfectly random. For such a source, $H_\infty(X) = k$. If $k = n^{\Omega(1)}$, this gives a weak random source.
- *Unpredictable sources.* These are sources where each bit is unpredictable given previous bits, namely for each i, $\Pr[X_{i+1} = 0 | X_0, X_1, ..., X_i] \in [1/2 - \delta, 1/2 + \delta]$ for some $\delta < 1/2$. For $\delta \leq 1/2 - 1/n^{1-\epsilon}$ for any constant $\epsilon < 1$ this gives a weak random source.
- *Flat sources.* These sources correspond to the uniform distribution over some subset S of the range $\{0,1\}^n$. It follows that $H_\infty(X) = \log |S|$, and such a source is a weak random source if $|S| = 2^{n^{\Omega(1)}}$. The following exercise shows that in fact flat sources are a base case for all sources with high min-entropy, and therefore if we can handle flat sources then we can handle all random sources with high min-entropy.

  Exercise 1. *Let X be a random source with $H_\infty(X) \geq k$. Then X is a convex combination of flat sources each on a subset of size at least 2^k.*

2 Randomness Extractor

Given a source of randomness with high min-entropy, we wish to output a distribution that is statistically close to uniform. Ideally, we would like to be able to convert weak randomness into near perfect randomness without using any additional perfect randomness. However, the following shows this is not possible.

Proposition 1. *Let $E : \{0,1\}^n \to \{0,1\}^m$ be a function taking input from a weak random source. There is a weak random source X with $H_\infty(X) = n - 1$ so that when $m = 1$, $E(X)$ is a constant function.*

Proof. In this setting, E outputs a single bit and so must output either 0 or 1 with probability $\geq 1/2$; suppose 0. Define X to be the flat distribution on $S = \{x | E(x) = 0\}$. Then X has min-entropy at least $n - 1$, yet $\Pr[E(X) = 0] = 1$ meaning that the output distribution of E is a constant. □

We think of E in the above as taking a weak random source as input and attempting to output bits that are close to uniform. The proposition shows that even for the weakest setting possible, this cannot be done. We therefore augment E with an additional input that comes from a perfect random source.

Definition 2 (extractor)**.** *$E : \{0,1\}^n \times \{0,1\}^\ell \to \{0,1\}^m$ is a (k, ϵ) extractor if for all X on $\{0,1\}^n$ with $H_\infty(X) \geq k$,*

$$\| E(X, U_\ell) - U_m \|_1 < 2\epsilon \tag{1}$$

where U_ℓ is a uniform variable on ℓ bits and U_m is uniform on m bits.

Note that (1) is equivalent to the following: for every event $A \subseteq \{0,1\}^m$,

$$|\Pr[E(X, U_\ell) \in A] - \Pr[U_m \in A]| < \epsilon. \tag{2}$$

Although we need some additional true randomness to define the extractor, we often will be in the setting where $\ell = O(\log n)$, meaning the amount of true randomness needed is very small. In fact, we will see uses of extractors in the next section that eliminate the need for any true randomness by cycling over all possible strings in $\{0,1\}^\ell$.

There are a number of parameters of Definition 2 that we might wish to optimize. The following bounds can be proven on any possible extractor.

- $\ell \geq \log n + 2 \log \frac{1}{\epsilon} - O(1)$
- $m \leq k + \ell - 2 \log \frac{1}{\epsilon} + O(1)$

We think of these bounds intuitively as: we need at least enough perfect random bits to specify an index into the weak random source, and we can extract out at most as many random bits contained in the combination of the weak source and perfect random source. It can be shown that picking a function at random with $\ell = \log n + 2 \log \frac{1}{\epsilon} + O(1)$ and $m = k + \ell - 2 \log \frac{1}{\epsilon} - O(1)$ with high probability satisfies Definition 2. However, this does not help us in the application we are interested in as the act of picking an extractor at random requires a large amount of perfect randomness. For our application, we want to develop extractors that are computable in deterministic polynomial time.

Exercise 2. *Show that if there is a combination of random source X and function E satisfying (1), then the source must be $O(\epsilon)$ close to a source with min-entropy $H_\infty(X) \geq m - \ell$.*

This fact justifies our choice of H_∞ as the notion of weak randomness.

3 Applications

We give two applications of extractors: to achieve our original goal of simulating randomized algorithms with weak random sources, and to give an alternate proof that BPP $\subseteq \Sigma_2^p$. In each application, we eliminate the need for the perfect randomness by using an extractor where $\ell = O(\log n)$ and cycling over all possible seeds. There are other areas where this is not feasible. For example, in many cryptographic settings, we do not have this luxury. There do exist extractors - called seedless extractors - that can be used in these settings. For these, the extractor takes input from two independent weak random sources and outputs a distribution close to uniform. We do not discuss seedless extractors but only mention their existence.

3.1 Simulating Randomized Algorithms

If we had an extractor that did not have the second input from a perfect random source, simulating a randomized algorithm with the extractor would be trivial. We describe a simulation in this section that removes the need for the perfect random source while giving a simulation that is correct with high probability.

Suppose we have a randomized algorithm M that needs m random bits and an extractor $E : \{0,1\}^n \times \{0,1\}^\ell \to \{0,1\}^m$. Given an input z, we simulate $M(z)$ as follows:

(1) Set $count = 0$.
(2) Let x be a sample from a random source X.
(3) **foreach** $y \in \{0,1\}^\ell$
(4) Let $\rho_y = E(x,y)$.
(5) **if** $M(z,\rho_y) = 1$ **then** $count = count + 1$
(6) **if** $count \geq 2^\ell/2$ **then** Output 1
(7) **else** Output 0

Let us consider the probability this simulation errors. Let B_z be the bad set for z on algorithm M, i.e., $B_z = \{\rho | M(z,\rho) \neq \text{maj}_r(M(z,r))\}$. Then the bad set for our simulation is

$$B'_z = \{x | \Pr_y[E(x,y) \in B_z] \geq 1/2\}. \tag{3}$$

Claim 1. *If E is a $(k, 1/6)$ extractor, then $|B'_z| < 2^k$.*

Proof. Suppose $|B'_z| \geq 2^k$, and let X be the flat source on B'_z. Notice that X has min-entropy at least k. Also, $\Pr[E(X,U_\ell) \in B_z] \geq 1/2$ while $\Pr[U_m \in B_z] \leq 1/3$. So we have a set B_z where the difference in probability assigned between the extractor and uniform is at least $1/6$, contradicting E being a $(k, 1/6)$ extractor. □

Given this claim, we compute the probability our simulation errors assuming E is a $(k, 1/6)$ extractor.

$$\Pr[\text{Simulation errors}] = \Pr_{x \leftarrow X}[x \in B'_z] \leq |B'_z| \cdot 2^{-H_\infty(X)} < 2^{k - H_\infty(X)}.$$

If we use a source X with $H_\infty(X)$ a bit more than k, this probability will be at most $1/3$ (for example, $H_\infty(X) \geq k + 2$ suffices).

Now consider the efficiency of the simulation. We hope that the simulation incurs only a $\text{poly}(m)$ factor overhead in time so that simulating a polynomial time algorithm still takes polynomial time. The time to complete the simulation is the product of: 2^ℓ, the time to compute E, and the time of the original algorithm. Given a $\text{poly}(n)$ computable extractor, the second term is $\text{poly}(m)$ if $H_\infty(X) \geq n^{\Omega(1)}$ because then $n = \text{poly}(m)$. The 2^ℓ term is $\text{poly}(m)$ if $\ell = O(\log m)$, or equivalently $\ell = O(\log n)$ since $n = \text{poly}(m)$. Recall the condition on X is precisely what we stated earlier as the requirement to be a weak random source. So, we see that our choice of weak random sources corresponds precisely with the random sources for which this analysis yields a correct simulation of M in polynomial time. We also remark that extractors with the parameters stated here do exist, and we demonstrate these later in the lecture.

3.2 Alternate Proof of BPP $\subseteq \Sigma_2^p$

For this application, we assume the existence of a $(n/2, 1/6)$ extractor $E : \{0,1\}^n \times \{0,1\}^\ell \to \{0,1\}^m$ computable in polynomial time and with $\ell = O(\log n)$. The existence of such an extractor is proven in the next section.

Given a BPP machine M requiring m random bits and input z, we wish to give a Σ_2^p formula equivalent to the acceptance of $M(z)$. We start by considering the simulation given in the previous section using E on a perfectly random source (one with $H_\infty(X) = n$). We view a sample x from this source as two components of equal length: $x = (x_1, x_2)$ where $|x_1| = |x_2| = n/2$. The number of x on which the simulation fails on a sample from X is $< 2^{n/2}$ by Claim 1. By a counting argument, there is a choice of x_1 so that the simulation when given (x_1, x_2) results in the correct answer for all x_2. Stated formally, for an input z, we have the following

$$z \in L(M) \Rightarrow \exists x_1 \forall x_2 (\Pr_y[M(z; E(x_1, x_2, y)) = 1] \geq 1/2).$$

Because $|y| = O(\log n)$ and assuming $m = n^{\Omega(1)}$, the inside predicate is computable in polynomial time. If we can show that $z \notin L(M)$ implies the negation of the RHS, we will be done. With this goal in mind, we switch the roles of x_1 and x_2, and note that the simulation outputs 0 only when the appropriate probability is less than $1/2$, and get the following:

$$\begin{aligned} z \notin L(M) \Rightarrow &\quad \exists x_2 \forall x_1 (\Pr_y[M(z; E(x_1, x_2, y)) = 1] < 1/2) \\ \Rightarrow &\quad \forall x_1 \exists x_2 \neg (\Pr_y[M(z; E(x_1, x_2, y)) = 1] \geq 1/2). \end{aligned}$$

The first line implies the second because $\exists x \forall y$ always implies $\forall y \exists x$ and $(\Pr_y[M(z; E(x_1, x_2, y)) = 1] < 1/2) = \neg(\Pr_y[M(z; E(x_1, x_2, y)) = 1] \geq 1/2)$.

4 Constructions

We give three different constructions of polynomial time computable extractors.

1. The first construction is based on the construction of pseudorandom generators secure against circuits that uses functions with high circuit complexity. This construction gives $\ell = O(\log n)$ and $m = k^{\Omega(1)}$ for sources with $k = n^{\Omega(1)}$. Note that these parameters are good enough for

the applications of the previous section, but the dependence of the number of output bits to the entropy of the input distribution is still far from optimal.

2. The second construction is based on a random walk on an expander graph. For any constant $\alpha > 0$, this construction gives a $\delta > 0$ and an extractor such that for sources with entropy $k = (1-\delta)n$, $m = (1-\alpha)k$, and $\ell = O(\log n)$. This construction is within constant factors of being optimal but only works for sources with very high min-entropy.

3. The final construction is based on polynomials over a finite field and uses the second construction as a building block. The parameters achieved are the same as the second construction, but it works for any value of k.

The first two we give today, and we give the third in the next lecture.

4.1 Hardness based Extractor

The basic idea here is to take the weak random source and view it as the truth table of a function. Most functions have high circuit complexity and so can be used as the hard function to the pseudorandom generator based on hard functions. We then use the perfect random seed as the seed for this generator. We now elaborate these ideas and derive the extractor.

Recall the pseudorandom generator that we gave in a previous lecture that is secure against circuits. Given a function $f : \{0,1\}^{\Theta(\log r)} \to \{0,1\}$ with non-uniform circuit complexity $C_f \geq r^{\Omega(1)}$, we were able to construct a generator $G_f : \{0,1\}^{\Theta(\log r)} \to \{0,1\}^r$ such that for all circuits D of size at most r,

$$|\Pr_{\sigma}[D(G_f(\sigma)) = 1] - \Pr_{\rho}[D(\rho) = 1]| \leq 1/r.$$

An examination of the proof reveals that the proof relativizes, so that if the function has circuit complexity $C_f^A \geq r^{\Omega(1)}$ even for circuits that are given an A oracle, then the output is indistinguishable even for circuits D^A with oracle access to A. In particular, the output of the generator given such a hard function is indistinguishable to the oracle circuit that just queries the oracle [1]. That is, if f has $C_f^A \geq r^{\Omega(1)}$, then

$$|\Pr_{\sigma}[A(G_f(\sigma)) = 1] - \Pr_{\rho}[A(\rho) = 1]| \leq 1/r. \tag{4}$$

This already looks similar to the condition of (2) for G_f being an extractor. (4) already makes use of a short random seed, but only works for a fixed f. We use the weak random source to pick a function f at random. We view a sample from the weak random source as specifying the truth table of f. Since f is a function on $O(\log r)$ bits, this can be specified in $r^{O(1)}$ bits.

Claim 2. *For any fixed oracle A, most functions $f : \{0,1\}^{\Theta(\log r)} \to \{0,1\}$ have $C_f^A \geq r^{\Omega(1)}$.*

We do not prove this claim or quantify what "most" means.

We are now ready to define the extractor. We view x as the truth table of a function f from $\Theta(\log r)$ bits to 1 bit, and view y as the seed σ for the pseudorandom generator G_f. Then

$$E(x,y) = G_f(\sigma).$$

[1] Note that for an oracle circuit, the size of a query to the oracle is charged as part of the size of the circuit. Otherwise, even very large queries to the oracle would count as only one gate, making even "constant sized" circuits powerful.

Now consider the difference in probability assigned to a set A by this extractor and the uniform distribution. By (4) and the fact that x comes from a weak random source with min-entropy at least k, we split the difference in probability assigned to A based off whether f has high circuit complexity or not, to get

$$\begin{aligned} |\Pr[E(x,y) \in A] - \Pr[U_r \in A]| &\leq \tfrac{1}{r} \cdot (\Pr_{x \leftarrow X}[C_f^A \text{ large}]) + 1 \cdot (\Pr_{x \leftarrow X}[C_f^A \text{ small}]) \\ &\leq \tfrac{1}{r} + 1 \cdot 2^{-k} \cdot (\#\text{ of } f \text{ with } C_f^A \text{ small}). \end{aligned}$$

The term (# of f with C_f^A small) can be quantified, but intuitively can be bounded by considering the number of small circuits. We want $C_f^A = r^{\Omega(1)}$. The number of circuits of size $r^{o(1)}$ is $2^{r^{o(1)}}$, so the number of f with small circuit complexity is bounded by this value. This means we only need to have $k = r^{\Omega(1)}$ to ensure the above probability is less than some constant ϵ, showing that $E(x,y)$ is a (k,ϵ) extractor.

4.2 Expander based Extractor

For this construction, we view the sample from the weak random source as describing a random walk on a d-regular expander graph. Let x be a sample from source X. We view the first portion as indicating an initial vertex x_0 within an expander, and the remaining part of x specifies a path to follow in the graph from this vertex. The perfect random seed y specifies a vertex along this path to output. Therefore, the first m bits of x specify x_0 and the next $t \log d$ specify a path of length t starting from x_0. To index a vertex along this path, $|y| = \log t$. To summarize, we let

$$E(x,y) = y^{th} \text{ point on random walk specified by } x.$$

Before proving this is an extractor, let us specify the relationship of the parameters. By definition, $n \geq m + t \log d$. We said previously that the construction should work for $m = (1-\alpha)k$ for any $\alpha > 0$, and we can pick a δ to set $k = (1-\delta)n$. Then for the δ we pick, we will have $n \geq (1-\alpha)(1-\delta)n + t \log d$. Setting $t = \frac{\alpha n}{\log d}$, we get that $\ell = |y| = \log(t) = O(\log n)$.

We must show that we can pick a δ so that (2) is satisfied. Let $A \subseteq \{0,1\}^m$. We use the Chernoff Bound for random walks on expanders to bound the difference in probability assigned to A by $E(x,y)$ and U_m. Recall that the Chernoff Bound for random walks states that if ρ_i is the i^{th} vertex visited on a random walk in an expander with second largest eigenvalue λ, then

$$\Pr_{x \leftarrow U_n}[\underbrace{|\frac{1}{t}\sum_i^t \chi[\rho_i \in A] - \mu(A)| \geq \epsilon}_{(*)}] \leq \underbrace{\exp(-b(1-\lambda)\epsilon^2 t)}_{(**)}. \tag{5}$$

Splitting the difference in probability assigned to A based off whether (*) holds for a particular x and assuming a weak random source with min-entropy k, we get

$$\leq \underbrace{|\Pr_{x \leftarrow X, y \leftarrow U_\ell}[E(x,y) \in A] - \Pr_{\rho \leftarrow U_m}[\rho \in A]|}_{\text{contribution of good } x \text{ breaking (*)}} \underbrace{\epsilon}_{} + 1 \cdot (\#\ x \text{ satisfying (*)}) \cdot 2^{-k}.$$

To finish, we bound (# x satisfying (*))$\cdot 2^{-k}$:

$$(\#\ x \text{ satisfying (*)}) \cdot 2^{-k} = (2^n \cdot (**)) \cdot 2^{-k} = 2^{\delta n} \cdot (**) = 2^{\delta n} 2^{-\beta n}$$

for some constant β depending on (**). If we pick δ sufficiently smaller than β, we get that $E(x,y)$ is a $(k, 2\epsilon)$ extractor.

4.3 Polynomial based Extractor

This construction is given in the next lecture.

CS 810: Complexity Theory 3/1/2007

Lecture 19: Counting

Instructor: Dieter van Melkebeek Scribe: Chi Man Liu

Last time we introduced extractors and discussed two methods to construct them. In the first part of this lecture, we present an explicit construction of extractors based on finite fields [1]. This construction gives extractors with the same parameters as the second construction discussed last time, except that it removes the constraint on the amount of randomness in the weak source.

In the second part of this lecture, we introduce the class #P of counting problems. Problems we have seen in class so far are decision problems where the output is a single bit. On the contrary, counting problems require integers (in the form of binary strings) as output. We look at some properties of #P, as well as relations between #P and other complexity classes.

1 Extractor from Condenser Overview

Recall that in the previous lecture we presented two constructions of polynomial time computable extractors. The first construction is based on pseudorandom generators secure against circuits. This construction gives $m = k^{\Omega(1)}$ and $\ell = O(\log n)$ for sources with $k = n^{\Omega(1)}$. The second construction is based on expander graphs. For any constant $\alpha > 0$, this construction gives a $\delta > 0$ such that for $k = (1-\delta)n$ it extracts $m = (1-\alpha)k$ bits while using $\ell = O(\log n)$ perfect random bits. In this section, we give a construction based on finite fields [1]. This construction uses the second construction as a building block, and achieves the same parameters except that it works for any value of k.

We restate the definition of an extractor here for reference.

Definition 1 (Extractor)**.** *A (k,ϵ)-*extractor *is a function $E : \{0,1\}^n \times \{0,1\}^\ell \to \{0,1\}^m$ such that for every source X over $\{0,1\}^n$ with $H_\infty(X) \geq k$, $\|E(X,U_\ell) - U_m\|_1 < 2\epsilon$.*

Recall that the second construction requires $k = (1-\delta)n$ for some $\delta > 0$. Given any weak random source, we want to apply a procedure that converts the random source into one suitable for applying the second construction. For an arbitrary source on n bits with min-entropy k, we want to condense randomness of the source, i.e., we want to make the ratio k/n close to 1 while keeping a high min-entropy. This intuitive notion is formalized in the following definition.

Definition 2 (Condenser)**.** *A (k,k',ϵ)-*condenser *is a function $C : \{0,1\}^n \times \{0,1\}^\ell \to \{0,1\}^m$ such that for every source X over $\{0,1\}^n$ with $H_\infty(X) \geq k$, there exists a distribution Y over $\{0,1\}^m$ with $H_\infty(Y) \geq k'$ and $\|C(X,U_\ell) - Y\|_1 < 2\epsilon$.*

Condensers are a natural generalization of extractors. While we require a near-uniform output from an extractor, we only require the output of a condenser to be close to any source with a certain amount of randomness. Note that in the definition of a condenser if we set $k' = m$, we get back an extractor. We give a construction of a condenser with good enough parameters so that the output distribution can be used as the weak random source of the second construction from last lecture.

Theorem 1. *For any constants $\alpha, \delta, \epsilon > 0$, and for all $k \leq n$, there exists a* poly(n) *time computable (k,k',ϵ)-condenser with $k'/k \geq 1-\delta$, $k'/m \geq 1-\alpha$ and $\ell = O(\log n)$.*

Before proving this theorem, we see how it can be used in combination with the expander based extractor. Let X be a source on n bits with $H_\infty(X) \geq k$ for any value k. For any $\alpha > 0$, pick some constant $\alpha' \in (0, \alpha)$. Suppose that extracting $(1-\alpha')m$ bits using the expander based extractor requires $k = (1-\delta')n$ where $\delta' > 0$ is a constant depending on α'. Applying the condenser on X with $\alpha = \delta'$ produces an output Y on $m' \geq (1-\delta)k$ bits that is $O(\epsilon)$ close to a distribution with min-entropy $k' \geq (1-\delta')m'$ while using a seed of length $O(\log n)$. We can now apply the extractor to get a distrbution Z over $m \geq (1-\alpha')(1-\delta)k$ bits which is $O(\epsilon)$ close to uniform while using an additional seed of length $O(\log k)$. To achieve $m = (1-\alpha)k$, we just need to pick $\delta = 1 - (1-\alpha)/(1-\alpha')$. Overall, a seed of $O(\log n)$ has been used, and composing the two constructions takes poly(n) time.

2 Condenser

To prove Theorem 1, we need to find a condenser that outputs a distribution with (or close to) high min-entropy for any input source with a guaranteed amount of randomness. For an arbitrary function $C : \{0,1\}^n \times \{0,1\}^\ell \to \{0,1\}^m$, its output distribution may contain elements with large weights, which is a hindrance to achieving high min-entropy. The following lemma shows that if the set of inputs generating these heavy elements is small enough, the output distribution of C will be close to a distribution with high min-entropy.

Lemma 1. *Let X be a source with $H_\infty(X) \geq k$, and C be a function from $\{0,1\}^n \times \{0,1\}^\ell$ to $\{0,1\}^m$. Define the set of* k'-heavy elements *as*

$$H = \{z \mid \Pr_{\substack{x \leftarrow X \\ y \leftarrow U_\ell}}[C(x,y) = z] > 2^{-k'}]\}.$$

Let the set of bad inputs *be*

$$BAD = \{x \mid \Pr_{y \leftarrow U_\ell}[C(x,y) \in H] > \frac{\epsilon}{2}\}.$$

If $|BAD| \leq \epsilon \cdot 2^{k-1}$, then there exists a distribution Z over $\{0,1\}^m$ with $H_\infty(Z) \geq k'$ such that $\|C(X, U_\ell) - Z\|_1 < 2\epsilon$.

Proof. Each element in X has weight at most 2^{-k} since X has min-entropy at least k. Thus $\Pr[X \in BAD] \leq |BAD| \cdot 2^{-k} \leq \epsilon/2$. Hence

$$\begin{aligned} \Pr_{\substack{x \leftarrow X \\ y \leftarrow U_\ell}}[C(x,y) \in H] &\leq \Pr[X \in BAD] + \Pr[X \notin BAD] \cdot \frac{\epsilon}{2} \\ &\leq \frac{\epsilon}{2} + (1 - \frac{\epsilon}{2})(\frac{\epsilon}{2}) \\ &< \epsilon. \end{aligned}$$

Note that the weight of any element in $\{0,1\}^m \backslash H$ is at most $2^{-k'}$. We can redistribute the weights on H to other elements in $\{0,1\}^m$, resulting in a new distribution Z in which no element has weight greater than $2^{-k'}$, i.e. $H_\infty(Z) \geq k'$. Since the weight needed to redistribute is $\Pr[C(x,y) \in H]$, which is less than ϵ, therefore the difference in probability between $C(X, U_\ell)$ and Y on any set is less than ϵ, so $\|C(X, U_\ell) - Z\|_1 < 2\epsilon$. □

If C is a function such that $|BAD| \leq \epsilon \cdot 2^{k-1}$ for all sources X with min-entropy at least k, then C is a (k, k', ϵ)-condenser. Note that although H depends on the source X, H always satisfies $|H| \leq 2^{k'}$. Thus, if we can show that there are few bad inputs with respect to *any* H satisfying $|H| \leq 2^{k'}$, then we can as well conclude that C is a (k, k', ϵ)-condenser. We have the following lemma.

Lemma 2. *Let H be any subset of $\{0,1\}^m$. Define the* bad set with respect to H *as*

$$BAD_H = \{x \mid \Pr_{y \leftarrow U_\ell}[C(x,y) \in H] > \frac{\epsilon}{2}\}.$$

If $|BAD_H| \leq \epsilon \cdot 2^{k-1}$ for all H with $|H| \leq 2^{k'}$, then C is a (k, k', ϵ)-condenser.

We can now prove our main theorem.

Proof of Theorem 1. We view the sample from the weak random source as specifying a polynomial over a finite field. The extractor evaluates powers of this polynomial at a point specified by the perfect random input and outputs these. We will use the condition given in Lemma 2 to show the construction is a condenser. For any small set in the range, we will construct a non-zero univariate polynomial over samples from the weak random source that evaluates to 0 on samples that hit the small set with high probability. If we can do this while keeping the degree of this polynomial small, this will show that not too many samples can hit the small set in the range with high probability. Previous constructions often use multivariate polyonmials instead of univariate polynomials. While multivariate polynomials generally give lower degrees, univariate polynomials provide us with a better handle on the number of zeroes, which turns out to be useful in our construction.

We now formalize this intuition. Let ℓ be the size of the perfect random seed, n the length of a sample from the weak random source, and m the length of our output. We view $\{0,1\}^\ell$ as $\mathrm{GF}(q)$ for $q = 2^\ell$. We view $\{0,1\}^n$ as $\mathrm{GF}(q)[Y]/g(Y)$ where $g(Y)$ is some irreducible polynomial of degree $n' = n/\ell$ over $\mathrm{GF}(q)$. We view m as being $m' = m/\ell$ elements of $\mathrm{GF}(q)$. Given this setup, we define the condenser C as

$$C(x,y) = (x(y), x^s(y), x^{s^2}(y), x^{s^3}(y), \ldots, x^{s^{m'-1}}(y))$$

for any $x \in \mathrm{GF}(q)[Y]/g(Y)$ and $y \in \mathrm{GF}(q)$, where s is a parameter to be set later. The above formula is interpreted as follows. $C(x,y)$ outputs an m'-vector over $\mathrm{GF}(q)$. The i-th element is computed by treating x as an element of $\mathrm{GF}(q)[Y]/g(Y)$, raising it to the s^{i-1}-th power, then evaluating the resulting polynomial at y.

Let H be any subset of $(\mathrm{GF}(q))^{m'}$ with $|H| \leq 2^{k'}$. Then there exists a nonzero formal m'-variate polynomial Q over $\mathrm{GF}(q)$ that vanishes at all points in H and has degree less than s in each variable. To see why, consider evaluating the polynomial at some point in H. This gives a homogeneous linear equation with the polynomial coefficients as unknowns. There are $|H| \leq 2^{k'}$ such equations and $s^{m'}$ (number of coefficients of Q) unknowns. If

$$s^{m'} > 2^{k'}, \tag{1}$$

such a homogeneous system always has a nontrivial solution, which gives the coefficients of Q. We will pick values at the end of this proof to satisfy Equation 1.

Let $x \in BAD_H$. We treat x as an element of $\mathrm{GF}(q)[Y]/g(Y)$. Consider the formal univariate polynomial $R_x(Y) = Q(x(Y), x^s(Y), \ldots, x^{s^{m'-1}}(Y))$. As all functions in $\mathrm{GF}(q)[Y]/g(Y)$ have degree at most n', $R_x(Y)$ has degree at most $(s-1)m'n'$. Let $y \in \mathrm{GF}(q)$. If $C(x,y) \in H$, then $R_x(y) = 0$ by the construction of C and Q. Hence, from the definition of BAD_H, we know that R_x vanishes on at least $\epsilon q/2$ points in $\mathrm{GF}(q)$. Here we make our second assumption:

$$\epsilon q/2 \geq sm'n'. \tag{2}$$

Then, R_x has more zeroes than its degree, and so it must be the zero formal polynomial. Thus, R_x gives the zero formal polynomial as remainder modulo $g(Y)$. This is equivalent to that R_x is the zero element in $\mathrm{GF}(q)[Y]/g(Y)$.

Let $S(X) = Q(X, X^s, \ldots, X^{s^{m'-1}}) \mod g(X)$ be a univariate polynomial in $\mathrm{GF}(q)[Y]/g(Y)$. From the above discussion, $S(x) = 0$ (the zero element in $\mathrm{GF}(q)[Y]/g(Y)$) for every $x \in BAD_H$. Because the individual degrees of Q were all less than s, S is a nonzero polynomial (because distinct monomials in Q map to distinct monomials in S). The degree of S is at most $(s-1)(1+s+s^2+\cdots+s^{m'-1}) = s^{m'}-1$. Therefore, S has at most $s^{m'}-1$ zeroes, and so $|BAD_H| < s^{m'}$. Our third assumption is

$$s^{m'} \leq \epsilon \cdot 2^{k-1}, \tag{3}$$

which leads to $|BAD_H| \leq \epsilon \cdot 2^{k-1}$, and so by Lemma 2 we can conclude that C is a (k, k', ϵ)-condenser.

We still have to pick the values for the parameters to satisfy our assumptions (Equations 1, 2 and 3 and achieve the claimed parameters of our condenser. By the statement of the theorem and the setup of the condenser, we already have the following: $k' \geq k(1-\delta), m \leq k'/(1-\alpha), q = 2^\ell, m' = m/\ell, n' = n/\ell$. We must show that we can choose s, q, and δ to ensure the assumptions are valid. We consider each in turn. For Equation 1, note that $s^{m'} = s^{m/\ell} = s^{k'/(\ell(1-\alpha))}$. Then Equation 1 is equivalent to $s^{k'/(\ell(1-\alpha))} > 2^{k'}$. By rearranging and using $q = 2^\ell$, this is equivalent to $s > q^{1-\alpha}$. We choose $s = q^{1-\alpha}$ to satisfy Equation 1.[1]

Now consider Equation 2. By substituting our value for s, this is $q\epsilon/2 \geq q^{1-\alpha}m'n'$, which can be rearranged as $q \geq \left(\frac{2m'n'}{\epsilon}\right)^{1/\alpha}$. Notice that $n > n' \geq m'$, so setting $q = \left(\frac{2n^2}{\epsilon}\right)^{1/\alpha}$ satisfies assumption 2. Note that $\ell = O(\log q) = O(\log n)$.

Now consider Equation 3. In fact, if we set $k' = k - 1 - \log(1/\epsilon)$, then with Equation 1 it is straightforward to see that Equation 3 holds.[2]

Finally, consider the efficiency of the construction. Finding an irreducible polynomial $g(Y)$ of degree n' over $\mathrm{GF}(2^\ell)$, as well as arithmetic operations over $\mathrm{GF}(2^\ell)$ and $\mathrm{GF}(2^\ell)[Y]/g(Y)$, can be done in time $\mathrm{poly}(n, \ell) = \mathrm{poly}(n)$. Hence the construction is polynomial-time computable.

□

3 Counting Class #P

All problems we have studied so far are decision problems — problems that require only "yes" or "no" answers. In this section, we introduce a new class of problems that require multiple-bit

[1] In fact, s has to be $q^{1-\alpha} + \zeta$ for some small constant $\zeta > 0$. But for simplicity, we just assume that $s = q^{1-\alpha}$. The rest of the argument still goes through if we increment s by ζ.

[2] Once again, we need to add a small constant to k' to get around the strict inequality.

outputs. In particular, we consider problems of the following form: Given an instance x, what is the number of solutions (or witnesses) to x? This leads us to exploring the class #P of counting problems.

3.1 Examples and Definitions

Before we give the precise definition of the complexity class #P, let us look at an example.

> #SAT: Given a Boolean formula ϕ, find the number of assignments satisfying ϕ.

The above problem is an example of a counting problem. Recall that the corresponding decision problem SAT, which asks whether there exists a satisfying assignment, is in NP. We can construct a NTM M which, on input ϕ, guesses an assignment and accepts if it satisfies ϕ. Note that the number of accepting computation paths of M on ϕ is exactly the number of satisfying assignments for ϕ. This leads us to defining #P in terms of NTMs.

Definition 3. $\#\mathrm{P} = \{f : \{0,1\}^* \to \mathbb{N} \mid$ *there exists a polynomial-time NTM M such that for all input x, $f(x)$ equals the number of accepting computation paths of M on x* $\}$.

We give some examples of problems in #P.

1. #SAT is in #P. This can be seen from the above discussion.

2. The problem #PM, which asks for the number of perfect matchings in a bipartite graph, is in #P. Note that the corresponding decision problem PM (existence of perfect matchings in bipartite graphs) can be solved in polynomial time. This problem arises from statistical physics.

3. For any polynomial-time decidable graph property (for example, connectivity and acyclicity), counting the number of graphs of a given size n that satisfy the property is a problem in #P. These problems appear very often in enumerative combinatorics.

4. Let A be an $n \times n$ matrix with coefficients in $\mathbb{N}$. (Note: not $\mathbb{Z}$.) Define the *permanent* of A as
$$perm(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} A_{i,\sigma(i)},$$
where S_n is the set of all permutations on $\{1, \ldots, n\}$. Note that the permanent of a matrix is very similar to the determinant except that a factor of $(-1)^{sign(\sigma)}$ is multiplied to each permutation for the determinant, where $sign(\sigma)$ denotes the sign of the permutation σ. The problem of computing the permanent of a given matrix A is in #P. This might not be obvious at first sight. In fact we can construct a NTM M that generates computation paths according to the entries of A as follows. On input A, M guesses a permutation σ. For each entry b captured by σ, M creates b computation paths. (If $b = 0$, reject immediately.) Each of these computation paths keeps on expanding, until all n entries have been processed. The total number of computation paths generated is exactly the product of the entries captured by σ. Summing over all permutations, we get the permanent of A. We now argue that M runs in polynomial time. Guessing a permutation takes $O(n \log n)$ steps since we need to specify n numbers and each number has $O(\log n)$ bits. Let m be the largest integer entry

in A. Spawning m computation paths takes $O(\log m)$ steps since only a constant number of computation paths can be spawned in each step. Thus, the total running time of M is $O(n \log n + n \log m)$, which is polynomial in the input length $\Omega(n^2 + \log m)$. As an aside, the number of perfect matchings in a bipartite graph G is equal to the permanent of the "adjacency matrix" of G. [3]

In the above, we restricted our attention to matrices with non-negative entries. If we relaxed this constraint and let the matrices have negative entries, than computing their permanents would not be a problem in #P, simply because the permanents could be negative. In fact, there is another complexity class that captures this problem:

$$\text{GapP} = \{f - g \mid f, g \in \#\text{P}\}.$$

We can show that the unrestricted permanent problem lies in GapP by constructing a NTM M^+ that only accepts permutations giving positive products, and another NTM M^- that only accepts permutations giving negative products.

3.2 Properties of #P

The complexity class #P exhibits some algebraic properties:

1. #P is closed under addition, i.e. for any two functions f and g in #P, their sum $f + g$ also lies in #P. To show that this holds, let M and N be NTMs inducing f and g respectively. Let P be an NTM which, on input x, immediately creates two computation paths. One of the paths runs the computation of M on x; the other runs the computation of N on x. It is easy to see that the total number of accepting computation paths is $f(x) + g(x)$. More generally, uniform exponential sums of #P functions are also in #P, i.e., for any $g \in \#\text{P}$ and constant c, the function
$$f(x) = \sum_{|y|=|x|^c} g(x, y)$$
is also in #P.

2. #P is closed under multiplication. This can be done by running the second NTM at the end of every accepting computation path of the first NTM. More generally, uniform polynomial products of #P functions are also in #P, i.e., for any $g \in \#\text{P}$ and constant c, the function
$$f(x) = \prod_{|y|=c \cdot \log |x|} g(x, y)$$
is also in #P.

Remark. It follows directly from properties (1) and (2) that computing permanents over $\mathbb{N}$ is in #P.

The complexity class GapP also has the above two properties. Furthermore, it is closed under subtraction, a property which #P does not possess.

[3] The "adjacency matrix" here is the same as usual adjacency matrices except that the rows represent vertices in one partition and the columns represent vertices in the other.

3.3 #P and Other Complexity Classes

Most of the complexity classes we have encountered are classes of decision problems. In order to compare #P with them, we need to make #P an oracle. Denote by $P^{\#P}$ the class of decision problems solvable using a polynomial-time DTM with access to a #P oracle, and $P^{\#P[1]}$ the class of decision problems solvable using a polynomial-time DTM that makes at most one query to a #P oracle.

Proposition 1. *The following relations hold:*

(a) $\mathrm{NP} \subseteq \mathrm{P}^{\#\mathrm{P}[1]}$.

(b) $\mathrm{BPP} \subseteq \mathrm{P}^{\#\mathrm{P}[1]}$.

(c) $\mathrm{P}^{\#\mathrm{P}[1]} \subseteq \mathrm{P}^{\#\mathrm{P}} \subseteq \mathrm{PSPACE}$.

Proof. Let L be a problem in NP, and M be a polynomial-time NTM solving L. Then the $\mathrm{P}^{\#\mathrm{P}}$ oracle used in the reduction is simply the function f_M induced by M, and $x \in L$ if and only if $f_M(x) > 0$. This completes part (a).

Part (b) can be proved similarly. Given a polynomial-time probabilistic machine P, we construct a NTM P' that guesses a random bit string (which is polynomial in length) and simulates the computation of P. The oracle used in the reduction is $f_{P'}$, and the oracle machine accepts if and only if the number of accepting computation paths of P' is at least 2/3 of the total number of computation paths.

Part (c) follows from the fact that the entire computation tree of an NTM can be traversed deterministically in polynomial space. □

Remark. The set containments in (a) and (b), as well as $\mathrm{P}^{\#\mathrm{P}} \subseteq \mathrm{PSPACE}$, are conjectured to be proper containments.

We introduce another counting-related complexity class here.

Definition 4. $\oplus\mathrm{P} = \{L \mid$ *there exists a polynomial-time NTM* M *such that for all* x, $x \in L$ *if and only if the number of accepting computation paths of* M *on* x *is odd* $\}$.

It is obvious that $\oplus\mathrm{P} \subseteq \mathrm{P}^{\#\mathrm{P}[1]}$. The following proposition shows a more interesting property of $\oplus$P, which is also seen in the classes P and BPP. The proof is left as an exercise for the reader.

Proposition 2. $(\oplus\mathrm{P})^{\oplus\mathrm{P}} = \oplus\mathrm{P}$.

3.4 #P-Complete Problems

In this section, we present some #P-complete problems.

Theorem 2. #SAT *is* #P*-complete under* $\leq_m^p$.

Proof. (Sketch) In Lecture 3 we proved that SAT is NP-complete. In that proof we took a polynomial-time NTM M and constructed a Boolean formula ϕ_x capturing the computation of M on some input x. It can be verified that the number of assignments satisfying ϕ_x is the same as the number of accepting computation paths of M on input x. □

Remark. The reduction used in the above proof is an example of a *parsimonious reduction*, which is a polynomial-time reduction preserving the number of solutions.

Theorem 3. *Computing the permanent of an integer matrix is* #P*-hard under* $\leq_o^p$.

Proof Idea: It can be shown that given a Boolean formula ϕ with ℓ occurrences of literals, we can construct in polynomial time an integer matrix A such that

$$perm(A) = 4^\ell \cdot \#(\phi), \tag{4}$$

where $\#(\phi)$ denotes the number of assignments satisfying ϕ. Given this construction, a single oracle call is needed to determine $\#(\phi)$. □

Theorem 4. #PM *is* #P*-complete under* $\leq_o^p$.

Proof Idea: This theorem can be proved by reducing integer matrix permanents to #PM, then applying Theorem 3. The reduction has two steps. In the first step, we get rid of all negative entries in the matrix using modular arithmetic. In the second step, we transform the nonnegative integer matrix into a 0/1-matrix (adjacency matrix). The complete proof is left as an exercise. □

Theorem 4 is quite a surprising result, as the corresponding decision problem PM is in P. The reduction from #SAT to #PM is not as simple as the one in Theorem 3. For if that was the case, SAT would be in P as follows: given a formula ϕ, reduce it to a graph G, determine whether G has a perfect matching, then apply Equation 4 above to conclude the satisfiability of ϕ. Each of these steps can be done in polynomial-time.

3.5 Reductions from Counting to Decision Problems

It is clear that any decision problem in NP can be reduced to its corresponding counting problem through a single oracle query. A more interesting question is: Can a counting problem be reduced to its corresponding decision problem through an oracle reduction? If the answer is positive for some #P-complete problem such as #SAT, this would imply that $\#\mathrm{P} \leq_o^p \mathrm{P}^{\mathrm{NP}}$ and so $\mathrm{P}^{\#\mathrm{P}} \subseteq \mathrm{P}^{\mathrm{P}^{\mathrm{NP}}} = \mathrm{P}^{\mathrm{NP}}$. By a theorem in the next lecture, this would give us $\mathrm{PH} \subseteq \mathrm{P}^{\mathrm{NP}}$, resulting in a collapse of PH to the second level. However, for some other problems in #P, we can give a positive answer to the above question. An example is the Graph Isomorphism problem (GI), which has not yet been shown to be either in P or NP-complete. This result is also considered as an evidence for GI not being NP-complete.

Recall that two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if there exists a bijection $f : V_1 \to V_2$ such that for any $u, v \in V_1$, $(u, v) \in E_1$ if and only if $(f(u), f(v)) \in V_2$. We write $G_1 \cong G_2$ if G_1 and G_2 are isomorphic. Define $\mathrm{GI} = \{\langle G_1, G_2\rangle \mid G_1, G_2 \text{ are graphs and } G_1 \cong G_2\}$. The corresponding counting problem is #GI which, given G_1 and G_2, asks for the number of different bijections satisfying the above edge-preserving condition. It is easy to verify that $\#\mathrm{GI} \in \#\mathrm{P}$. A related counting problem is the Graph Automorphism counting problem (#GA) which, given a graph $G = (V, E)$, asks for the number of *automorphisms* of G (edge-preserving permutations on V).

Theorem 5. $\#\mathrm{GI} \leq_o^p \mathrm{GI}$.

Proof. We only show that #GA $\leq_o^p$ GI. Showing #GI $\leq_o^p$ #GA is left as an exercise for the reader.

Let $G = (V, E)$ be a graph with $|V| = n$. The *automorphisms* of G forms a group $\mathrm{Aut}(G)$. Pick a vertex z. Let F_z be the set of all automorphisms fixing z, i.e. $\mathcal{F}_z = \{\pi \in \mathrm{Aut}(G) \mid \pi(z) = z\}$. Let C_z be the set of vertices that z can be mapped to by some automorphism, i.e. $C_z = \{\pi(z) \mid \pi \in \mathrm{Aut}(G)\}$. For each w in C_z, associate with it an automorphism π_w such that $\pi_w(z) = w$. It follows that every automorphism in $\mathrm{Aut}(G)$ can be uniquely decomposed as $\pi_w \circ \sigma$ where $w \in C_z$ and $\sigma \in \mathcal{F}_z$. Thus, $|\mathrm{Aut}(G)| = |C_z||\mathcal{F}_z|$. This also follows from elementary group theory.

To compute $|C_z|$, we make use of the GI oracle. For each $v \in V$, we can decide if $v \in C_z$ using a "coloring" technique. Intuitively, we "color" v with some color to get a colored graph G_1. Likewise, we "color" z with the same color to get G_2. Then, we ask the oracle whether G_1 and G_2 have a color-preserving isomorphism. Since $v \in G_1$ can only be mapped to $z \in G_2$, $v \in C_z$ if and only if $G_1 \cong G_2$. However, GI does not answer queries regarding color-preserving isomorphisms. One way to achieve this coloring effect is by attaching soem rigid graph[4] of size at least $n + 1$ to v and z.

Computing $|\mathcal{F}_z|$ is in fact another instance of #GA: We can color z with a certain color, then count the number of color-preserving automorphisms of the new graph. Since z is the only vertex with that color, all color-preserving automorphisms of the new graph must fix z. Note that this graph is larger than the original graph G since we have attached a rigid graph to z. However, it is in fact an easier instance of #GA because one of the vertices of G has been fixed. When solving this instance, we can fix a vertex $z' \neq z$ which is not in the clique, and compute $|C_{z'}|$ and $|\mathcal{F}_{z'}|$ by coloring z' with another color. This goes on recursively for at most n steps. In each step, a rigid graph of polynomial size is added to the graph, so the final graph still has polynomial size. Each step takes polynomial time, hence the whole reduction is polynomial-time computable. □

4 Next Time

Next lecture we will discuss the relations between #P and the polynomial hierarchy PH. In particular, we will prove that PH $\subseteq \mathrm{P}^{\#\mathrm{P}[1]}$. We will also see that although it is unlikely that PH can handle exact counting, it can be shown that we can do approximate counting in the second level of PH.

References

[1] V. Guruswami, C. Umans and S. Vadhan. Extractors and condensers from univariate polynomials. *Electronic Colloquium on Computational Complexity*, Report TR06-134, October 2006.

[4] Rigid graphs are graphs that have only one automorphism — the trivial automorphism. These graphs do exist.

CS 810: Complexity Theory 3/2/2007

Lecture 20: Alternation vs. Counting

Instructor: Dieter van Melkebeek Scribe: Jeff Kinne

We introduced counting complexity classes in the previous lecture and gave some basic properties, including the relation between counting and decision classes. In this lecture we give results relating counting to the polynomial hierarchy. The first result shows that any #P function can be approximated in the second level of the polynomial hierarchy, giving evidence that approximate counting is not much more difficult than deciding. The second result gives evidence that exact counting is more difficult by showing that the entire polynomial hierarchy can be decided with a single query to a #P function.

The proofs of both results make use of families of universal hash functions. These are small function families that behave in certain respects as if they were random, allowing efficient random sampling. We first introduce universal hash functions, and then prove the two main results.

1 Universal Families of Hash Functions

A universal family of hash functions is a collection of functions. We wish the set of functions to be of small size while still behaving similarly to the set of all functions when we pick a member at random. This is made possible by choosing the appropriate notion of "behaving similarly".

Definition 1. $\mathcal{H} = \{h : \{0,1\}^n \to \{0,1\}^m\}$ *is a* universal family of hash functions *if the following holds. For all* $x_1 \neq x_2 \in \{0,1\}^n$ *and* $y_1, y_2 \in \{0,1\}^m$,

$$\Pr_{h \in \mathcal{H}}[h(x_1) = y_1 \wedge h(x_2) = y_2] = \frac{1}{2^{2m}}$$

where h *is chosen uniformly at random from the functions in* $\mathcal{H}$.

Notice that the probability is the same as if we had picked h from all possible functions. If we fix x_1 and x_2 and pick $h \in \mathcal{H}$ at random, then the random variables $h(x_1)$ and $h(x_2)$ are independent. So we can think of universal hash functions as giving us the ability to produce uniform pairwise independent samples. The definition above can be generalized to define k-universal hash functions that produce k-wise independent samples.

We will make use of a few immediate consequences of the above definition.

- For any $x_1 \neq x_2 \in \{0,1\}^n$ $\Pr_{h \in \mathcal{H}}[h(x_1) = h(x_2)] = \frac{1}{2^m}$.
- For any $x \in \{0,1\}^n$ and $y \in \{0,1\}^m$, $\Pr_{h \in \mathcal{H}}[h(x) = y] = \frac{1}{2^m}$.

To be able to efficiently sample from $\mathcal{H}$, we would like the family to be small. There are a number of ways to achieve this; we give two.

Example: Let $\mathcal{H} = \{h(x) | h(x) = Tx + v$ where T is some $m \times n$ Toeplitz matrix over GF(2) and v is some $m \times 1$ vector over GF(2)$\}$. A Toeplitz matrix is one where entries along each diagonal are all the same. So it takes $(m + n - 1) + m = 2m + n - 1$ bits to specify an element of $\mathcal{H}$, whereas the specification of a random function from all possible functions requires $m2^n$ bits to specify the

truth table. If the correctness of a randomized algorithm only relies on pairwise independence, this universal family of hash functions can be used as a $O(\log r)$ length seed pseudorandom generator to derandomize the algorithm.

We leave it as an exercise to prove that $\mathcal{H}$ satisfies Definition 1. ⊠

Example: Another example is the set of affine functions over a finite field. We leave it as an exercise to prove that $\mathcal{H} = \{h(x) = (ax+b \pmod{2^m}) | a, b \in GF(2^m)\}$ is a universal family of hash functions from $\{0,1\}^n$ to $\{0,1\}^m$ for all $m \leq n$, where by $\pmod{2^m}$ we mean that we are working over the field $GF(2^m)$ (correctness follows from the fact that $GF(2^m)$ is a field). ⊠

1.1 Applications

Before delving into the main results of this lecture, we give some intuition of how universal families of hash functions will be useful. Let $S \subseteq \{0,1\}^n$. Then intuitively, we expect that if we pick $2^m \approx |S|$, a randomly chosen hash function from n bits to m bits will map S to $\{0,1\}^m$ with few collisions. That is, for $h \in_U \mathcal{H}$, with high probability $h(S) \approx \{0,1\}^m$. In the first application, we wish to determine the size of S where S is an NP witness set. We will make use of the above intuition to derive a Σ_2^p predicate that allows us to approximately determine the size of S. In a later application, we wish to use randomness to reduce a satisfiable formula to another one that is uniquely satisfiable (satisfiable by only one assignment). We will use the above intuition to show that by choosing m appropriately and picking h at random, there will with high probability be a unique satisfying assignment that hashes to 0^m.

2 Approximate Counting

In this section we prove that any #P function can be approximated to within a polynomial factor with an oracle for the second level of the polynomial hierarchy.

Theorem 1. *For any $f \in$ #P and for all $a > 0$, there is a function g computable in polynomial time with oracle access to a Σ_2^p language such that for all x,*

$$|f(x) - g(x)| \leq \frac{f(x)}{|x|^a}.$$

Notice that the goal in proving Theorem 1 is similar to the goal in showing that BPP $\subseteq \Sigma_2^p$. There, we needed to approximate the function counting the number of accepting random strings to within a constant factor. Now, we wish to approximate a #P function to within any polynomial factor.

Proof. Let the underlying NTM for f run in time n^c, and let us view it as a polynomial-time verifier. The NTM takes a certificate y of length n^c along with input x. We wish to determine the size of the set S_x of certificates causing the NTM to accept. Consider applying a randomly chosen hash function to the set of possible certificates, with $h : \{0,1\}^{n^c} \to \{0,1\}^m$ for some value m. If 2^m is large compared to S_x, we expect that $h(S_x)$ covers only a small portion of the range. If 2^m is small compared to S_x, we expect $h(S_x)$ to cover a large portion of the range. If we take a small collection of hash functions, they will collectively cover all of $\{0,1\}^m$ provided S_x is roughly the

same size as 2^m. We will construct a Σ_2^p predicate that can be queried to determine if a small set of hash functions covers the range. By querying this language for increasing values of m until we get a negative answer, we get an estimate for $|S_x|$.

First attempt. We mimic the proof that BPP $\subseteq \Sigma_2^p$ to try to determine if $|S_x| \approx 2^m$. There, we showed that if x is accepted by a BPP machine, then there exists a small set of shift vectors $\sigma_1, ..., \sigma_t$ so that shifting the witness set covers the entire set of random strings. Here, we use hash functions rather than shift vectors, and want to see if hashing the witness set by a small set of hash functions covers all of $\{0,1\}^m$.

We first bound the probability that a fixed $z \in \{0,1\}^m$ is not hit by a randomly chosen h. We wish to upper bound $\Pr_{h\in\mathcal{H}}[(\sum_{y\in S_x} \chi_{h(y)=z}) = 0]$. We would like to use the fact that choosing h at random results in pairwise independent samples to use Chebyshev's inequality to bound this probability. To do this, we need to compute the expected value of the sum. By linearity of expectation and the properties of universal hash functions,

$$E_{h\in\mathcal{H}}[\sum_{y\in S_x} \chi_{h(y)=z}] = \sum_{y\in S_x} E_{h\in\mathcal{H}}[\chi_{h(y)=z}] = \sum_{y\in S_x} \frac{1}{2^m} = \frac{|S_x|}{2^m}.$$

Denote this value as E_m. By Chebyshev's inequality and pairwise independence,

$$\begin{aligned}\Pr_{h\in\mathcal{H}}[(\textstyle\sum_{y\in S_x} \chi_{h(y)=z}) = 0] &\leq \Pr_{h\in\mathcal{H}}[|\textstyle\sum_{y\in S_x} \chi_{h(y)=z} - E_m| \geq E_m] \\ &\leq \frac{\sigma^2(\sum_{y\in S_x} \chi_{h(y)=z})}{E_m^2} = \frac{|S_x|\frac{1}{2^m}(1-\frac{1}{2^m})}{E_m^2} < \frac{1}{E_m}.\end{aligned}$$

Then if $|S_x| \geq 2^{m+1}$, the probability that a fixed z is not hit by a randomly chosen h is at most $1/2$. By picking t hash functions independently, this probability is at most $1/2^t$. A union bound over all z shows that if $|S_x| \geq 2^{m+1}$ then $\{0,1\}^m$ is covered by t randomly chosen hash functions with probability at least $1 - \frac{2^m}{2^t}$. On the other side, each hash function covers at most $|S_x|$ elements of the range, so if $t \cdot |S_x| < 2^m$ the probability of covering the range is 0. To sum up,

$$\frac{|S_x|}{2^m} \geq 2 \text{ and } t \geq m \Rightarrow (\exists h_1, ..., h_t)(\forall z \in \{0,1\}^m)[z \in \bigcup_{i=1}^{t} h_i(S_x)] \tag{1}$$

$$\frac{|S_x|}{2^m} < \frac{1}{t} \Rightarrow \neg(\exists h_1, ..., h_t)(\forall z \in \{0,1\}^m)[z \in \bigcup_{i=1}^{t} h_i(S_x)]. \tag{2}$$

We can encode the RHS of the above as a language which we can query to determine if $|S_x| \approx 2^m$ or not. This can be used to get the approximation factor that we desired, but unfortunately evaluating the inner predicate $(z \in \bigcup_{i=1}^{t} h_i(S_x))$ requires an existential quantifier to guess a witness y that is mapped to z by some h_i - meaning we would need a Σ_3^p language.

Second attempt. With a bit of work, we can reduce the complexity of the oracle from Σ_3^p to Σ_2^p. As the inner predicate needs an existential quantifier, this would be achieved if we could swap the order of the first two quantifiers in (1) and (2). Notice that this is not a problem for (1), but doesn't work for (2). As for (2), under the stronger assumption that $t \cdot |S_x| \leq 2^{m-1}$, we have that

$$(\forall h_1, ..., h_t) \Pr_{z\in\{0,1\}^m}[z \in \bigcup_{i=1}^{t} h_i(S_x)] \leq \frac{1}{2}.$$

Thus, if we pick ℓ z's at random (we abbreviate this as $Z = z_1, ..., z_\ell$),

$$(\forall h_1, ..., h_t) \Pr_Z[Z \subseteq \bigcup_{i=1}^{t} h_i(S_x)] \leq \frac{1}{2^\ell}.$$

So if $2^\ell >$ # choices for $h_1, ..., h_t$,

$$\Pr_Z[(\exists h_1, ...h_t)[Z \subseteq \bigcup_{i=1}^{t} h_i(S_x)]] < 1. \tag{3}$$

We conclude that

$$\begin{array}{rcl} \frac{|S_x|}{2^m} \geq 2 \text{ and } t \geq m & \Rightarrow & (\forall Z)(\exists h_1, .., h_t)[Z \subseteq \bigcup_{i=1}^{t} h_i(S_x)], \\ \frac{|S_x|}{2^m} \leq \frac{1}{2t} & \Rightarrow & \neg(\forall Z)(\exists h_1, ..., h_t)[Z \subseteq \bigcup_{i=1}^{t} h_i(S_x)]. \end{array} \tag{4}$$

We now set the parameters and verify that the RHS is a Π_2^p predicate. We set $t = m$ and need to set ℓ so that $2^\ell >$ # choices for $h_1, ..., h_t$. Notice that the running time of the predicate has a factor of ℓ to guess $Z = z_1, ..., z_\ell$, so we also need ℓ to be polynomial. This is where it is critical that we are drawing the h_i from a universal family of hash functions. The examples we gave earlier show that h_i can be specified with $O(n^c + m)$ bits, so setting $\ell = \Theta(m \cdot (n^c + m))$ is good enough to ensure that 2^ℓ is large enough. The inner existential quantifier is now side-by-side with the existential quantifier needed to evaluate the inner predicate, and we conclude that the RHS is a Π_2^p predicate. Finally, the Σ_2^p language that is the complement of the above is equivalent when used as an oracle.

We now see how to use an oracle to the Σ_2 language to approximate $|S_x|$. As mentioned earlier, we query the predicate for each value of $m = 1, 2, ..., n^c$ and determine the first value m^* where the answer to the predicate is negative. By (4), we know that for $m \leq (\log |S_x|) - 1$ the predicate is answered positively; and for $m \geq (\log |S_x| + \log\log |S_x|) + O(1)$ the predicate is answered negatively. Then $(\log |S_x|) - 1 \leq m^* \leq (\log |S_x| + \log\log |S_x|) \cdot O(1)$, which can be rewritten as

$$|S_x| \leq 2^{m^*+1} \leq O(|S_x| \log |S_x|).$$

As $\log |S_x| \leq n^c$, we have an approximation for $|S_x|$ that is within a fixed polynomial factor and is computable in $\mathrm{P}^{\Sigma_2^p}$.

But we would like to be within any polynomial factor, in particular within $1/n^a$ for some constant a. We obtain this by applying the above procedure on a modified predicate. If $f \in \#\mathrm{P}$ is the original function we are trying to approximate, we apply the above algorithm on the function $f' = f^{n^d}$ for a constant d we choose later. By the closure properties of #P, f' is a #P function whenever f is. By the above, our approximation for f' gives

$$f'(x) < 2^{m^*+1} < O(f'(x) \log(f'(x))).$$

Taking a $1/n^d$ power and rearranging, this becomes

$$f(x) \leq 2^{(m^*+1)/n^d} \leq f(x)(O(n^d \log f(x)))^{1/n^d} \leq f(x)(O(n^{d+c}))^{1/n^d}.$$

We have an approximation for $f(x)$ with relative error $< (O(n^{d+c}))^{1/n^d} = 2^{(O(1)+(d+c)\log n)/n^d}$. By looking at the Taylor expansion of this value, we see that it is $1 + \Theta(\frac{\log n}{n^d})$. We can set d large enough so the relative error is at most $1/n^a$. □

A few notes about the above proof.

- If the condition for (3) is strengthened to $2^\ell > 2 \cdot \#$ choices for $h_1, ..., h_t$, then we have

$$\frac{|S_x|}{2^m} \geq 2 \text{ and } t \geq m \Rightarrow \Pr_Z[(\exists h_1, ..., h_t)[Z \subseteq \bigcup_{i=1}^{t} h_i(S_x)]] = 1,$$

$$\frac{|S_x|}{2^m} \leq \frac{1}{2t} \Rightarrow \Pr_Z[(\exists h_1, ..., h_t)[Z \subseteq \bigcup_{i=1}^{t} h_i(S_x)]] \leq \frac{1}{2}.$$

  Plugging this into the argument shows that the approximation can be "computed in $\mathrm{RP}^{\mathrm{NP}}$" in the following sense: there is a randomized machine with oracle access to an NP language which computes an approximation to $f(x)$ where the estimate is never smaller than the true value of $f(x)$ and is never larger than $(1 + \frac{1}{|x|^a}) \cdot f(x)$.

- The language that is used as an oracle is a Π_2^p predicate and remains so even if checking whether $y \in S_x$ requires nondeterminism.

Both of these will play a role in an application of approximate counting (to AM games) later in the semester.

3 Exact Counting

For our second main result, we show the following.

Theorem 2. *Any language in the polynomial hierarchy can be decided in polynomial time with a single oracle query to a* #P *function, namely* $\mathrm{PH} \subseteq \mathrm{P}^{\#\mathrm{P}[1]}$.

We prove this theorem in three parts.

1. We first show $\mathrm{NP} \subseteq \mathrm{RP}^{\text{UNIQUE-SAT}}$. UNIQUE-SAT is the promise problem defined on formulae that have exactly either one or zero satisfying assignments, with the positive instance being uniquely satisfiable formulas. The UNIQUE-SAT oracle is guaranteed to give the correct answer on such formulae, and can act arbitrarily on others. In fact, the RP algorithm given is correct even if the oracle gives inconsistent answers on queries that are outside of the promise.
2. We next use the first part to show that $\mathrm{PH} \subseteq \mathrm{BPP}^{\oplus\mathrm{P}}$.
3. We finish the proof by showing that $\mathrm{BPP}^{\oplus\mathrm{P}} \subseteq \mathrm{P}^{\#\mathrm{P}[1]}$.

3.1 Solving NP with randomness and UNIQUE-SAT oracle

We again use hash functions for this theorem. Consider the NTM for SAT running in time n^c. We look at the set of all possible assignments for the formula given as input and consider applying a hash function on these. The idea is to try to choose the range of the hash function about the same size as S_x. If we can achieve this, we show that a randomly chosen hash function with high probability maps a unique satisfying assignment to 0^m. This gives us a potential UNIQUE-SAT query for the oracle.

We first bound the probability that a randomly chosen hash function maps a unique satisfying assignment to 0^m. Let S_x be the set of satisfying assignments, and let $\mathcal{H}_m$ be a universal family of hash functions from $\{0,1\}^{n^c}$ to $\{0,1\}^m$. The probability that h maps a unique satisfying assignment to 0^m is given by

$$\Pr_{h\in\mathcal{H}_m}[\sum_{y\in S_x}\chi_{h(y)=0^m}=1]=\Pr_{h\in\mathcal{H}_m}[\sum_{y\in S_x}\chi_{h(y)=0^m}\geq 1]-\Pr_{h\in\mathcal{H}_m}[\sum_{y\in S_x}\chi_{h(y)=0^m}\geq 2].$$

For the first term we have

$$\Pr_{h\in\mathcal{H}_m}[\sum_{y\in S_x}\chi_{h(y)=0^m}\geq 1]\geq\frac{|S_x|}{2^m}-\binom{|S_x|}{2}\frac{1}{2^{2m}}$$

by considering the first two terms of the inclusion-exclusion principle expansion of the probability and using pairwise independence of the hash functions. For the second term we have

$$\Pr_{h\in\mathcal{H}_m}[\sum_{y\in S_x}\chi_{h(y)=0^m}\geq 2]\leq\binom{|S_x|}{2}\frac{1}{2^{2m}}$$

by union bound and pairwise independence. Putting these two together and using the fact that $\binom{|S_x|}{2}\leq\frac{|S_x|^2}{2}$ gives us

$$\Pr_{h\in\mathcal{H}_m}[\sum_{y\in S_x}\chi_{h(y)=0^m}=1]\geq\frac{|S_x|}{2^m}\left(1-\frac{|S_x|}{2^m}\right)$$

which is equal to $X(1-X)$ for $X=\frac{|S_x|}{2^m}$. This value is symmetric around $X=1/2$ and achieves its maximum of $1/4$ here. As $\frac{|S_x|}{2^m}$ increases by 2 for each value of m, there is some choice of m causing the probability to be in the range $[1/3,2/3]$ providing $S_x\neq\emptyset$. For that value of m, the probability that a randomly chosen hash function maps a unique satisfying assignment to 0^m is at least $2/9$.

Given the above analysis, the following is the $\mathrm{RP}^{\text{UNIQUE-SAT}}$ algorithm for SAT.

INPUT: formula ϕ.
(2) **foreach** $m=0,1,2,...,n^c$
(3) Pick $h\in\mathcal{H}_m$ at random.
(4) Convert the following into a SAT query and ask the UNIQUE-SAT oracle: is there an assignment y that both satisfies ϕ and $h(y)=0^m$?
(5) **if** Oracle says yes **then** Use self-reducibility to find y, and verify $\phi(y)=1$. If yes, then output "Yes".
(6) Output "No".

Because we are choosing h from a universal family of hash functions, choosing the hash function can be done in polynomial time. The rest of the algorithm also runs in polynomial time. Suppose ϕ is satisfiable. Then for at least one choice of m, with probability at least $2/9$ line (4) corresponds to a uniquely satisfiable formula. Notice that the formula remains uniquely satisfiable when using self-reducibility, so in this case, the algorithm correctly outputs "Yes". If ϕ is not satisfiable, the algorithm always outputs "No". The probability of success on satisfiable formulas can be amplified by repeating the above, so the algorithm is $\mathrm{RP}^{\text{UNIQUE-SAT}}$.

3.2 Randomized reduction of PH to $\oplus$P

Now we can use the fact that NP $\subseteq$ RP$^{\text{UNIQUE-SAT}}$. Because UNIQUE-SAT only cares about formulas with exactly 0 or 1 satisfying assignments, $\oplus$SAT solves UNIQUE-SAT on formulas within the promise. Then the previous section shows that NP $\subseteq$ BPP$^{\oplus \text{P}}$.

Given this, we use the fact that a problem from the first homework relativizes. Namely, NP $\subseteq$ BPP $\Rightarrow$ PH $\subseteq$ BPP relativizes. So NP$^{\oplus \text{P}} \subseteq$ BPP$^{\oplus \text{P}} \Rightarrow$ PH$^{\oplus \text{P}} \subseteq$ BPP$^{\oplus \text{P}}$. We want to show the hypothesis of this. To get this, we notice that NP $\subseteq$ BPP$^{\oplus \text{P}}$ relativizes as well, giving NP$^{\oplus \text{P}} \subseteq$ (BPP$^{\oplus \text{P}}$)$^{\oplus \text{P}}$.

In general, giving an additional oracle O_2 to an oracle machine with access to O_1 can be solved by the base machine by giving it access to oracles O_1 and $O_1^{O_2}$. For the case of (BPP$^{\oplus \text{P}}$)$^{\oplus \text{P}}$, we get that a BPP machine requires access to a $(\oplus \text{P})^{\oplus \text{P}}$ oracle in addition to a $\oplus$P oracle. These can be combined into a single $(\oplus \text{P})^{\oplus \text{P}}$ oracle using the completeness of $\oplus$SAT, and using the fact that $(\oplus \text{P})^{\oplus \text{P}} = \oplus \text{P}$, we get NP$^{\oplus \text{P}} \subseteq$ BPP$^{(\oplus \text{P}^{\oplus \text{P}})} =$ BPP$^{\oplus \text{P}}$. Having achieved NP$^{\oplus \text{P}} \subseteq$ BPP$^{\oplus \text{P}}$, we conclude that PH $\subseteq$ PH$^{\oplus \text{P}} \subseteq$ BPP$^{\oplus \text{P}}$.

We point out that the proof given here of this result is simpler than the original proof. This is a demonstration of the power of relativization.

3.3 Deterministic reduction of BPP$^{\oplus \text{P}}$ to #P

Given a language L in BPP$^{\oplus \text{P}}$, we wish to determine if $x \in L$ by making a single query to a #P function. We first show that we can separate out the randomized portion of L from the counting portion. This will be useful in performing the reduction.

Definition 2. *Let $\mathcal{C}$ be a complexity class. We define the* BP *operator to give a new complexity class of languages, where* $\text{BP} \cdot \mathcal{C} = \{L | (\exists c > 0)(\exists L' \in \mathcal{C})$

$$\begin{array}{l} x \in L \Rightarrow \Pr_{y \in |x|^c}[\langle x, y \rangle \in L'] > 2/3, \\ x \notin L \Rightarrow \Pr_{y \in |x|^c}[\langle x, y \rangle \in L'] < 1/3 \quad \} \end{array}$$

Notice that $\text{BP} \cdot \text{P} = \text{BPP}$, so this is a reasonable definition of the BP operator. The alternative characterization of BPP$^{\oplus \text{P}}$ we use is given by the following.

Claim 1. BPP$^{\oplus \text{P}} = \text{BP} \cdot \oplus \text{P}$.

Proof. We show that for any complexity class $\mathcal{C}$, BPP$^{\mathcal{C}} = \text{BP} \cdot \text{P}^{\mathcal{C}}$. The result then follows by using $\oplus$P as $\mathcal{C}$ and the fact that P$^{\oplus \text{P}} = \oplus \text{P}$.

Consider a BPP machine that can ask queries to a $\mathcal{C}$ language. Without changing the computation, the machine can guess its random bits at the beginning the computation before proceeding. What is left is a P$^{\mathcal{C}}$ predicate. Now consider a BP $\cdot$ P$^{\mathcal{C}}$ language. The BPP$^{\mathcal{C}}$ machine computing the same language simply generates enough random bits to be the second input of the P$^{\mathcal{C}}$ machine and then simulates that machine. □

Now consider a language $L \in \text{BP} \cdot \oplus \text{P}$ which we hope to solve in P$^{\#\text{P}[1]}$. By definition, there is some $f \in \#\text{P}$ such that

$$\begin{array}{l} x \in L \Rightarrow \Pr_{|y|=n^c}[f(x, y) \equiv 1 \pmod 2] > 2/3 \\ x \notin L \Rightarrow \Pr_{|y|=n^c}[f(x, y) \equiv 1 \pmod 2] < 1/3 \end{array}$$

We would like to create a #P function which sums f over all y such that we can detect the gap in probability. The following claim is the main ingredient to make this happen.

Claim 2. *Let $f \in$ #P. Then there is a function $g \in$ #P such that*

$$f(z) \equiv 1 \pmod 2 \Rightarrow g(z, 0^M) \equiv -1 \pmod{2^M}$$
$$f(z) \equiv 0 \pmod 2 \Rightarrow g(z, 0^M) \equiv 0 \pmod{2^M}$$

where $M = 2^m$ is some power of 2.

Before proving this claim, we see how it allows us to complete the construction. Notice that if $f(z) \equiv 1 \pmod 2$, then the last M bits of $-g(z, 0^M)$ are 000...001; if $f(z) \equiv 0 \pmod 2$, the last M bits are 000...000. We would like to sum up $-g(z)$ for all $z = (x, y)$ in such a way that the results do not "spill over" past the last M bits. Because we must sum over 2^{n^c} many y, we need to pick $M > n^c$ to ensure there is no spill over. Namely notice that if $M > n^c$, then

$$x \in L \Rightarrow \textstyle\sum_{|y|=n^c} -g(x, y, 0^M) \pmod{2^M} > \frac{2}{3}2^{|x|^c}$$
$$x \notin L \Rightarrow \textstyle\sum_{|y|=n^c} -g(x, y, 0^M) \pmod{2^M} < \frac{1}{3}2^{|x|^c}.$$

In fact, by picking $M > n^c$ we ensure there is a single bit of the sum that we can check to distinguish between the two cases. Thus, $g'(x) = \sum_{|y|=n^c} g(x, y, 0^M)$ is the #P function that we query once to determine if $x \in L$ or not. That is, we compute $g'(x)$, and check if $-g'(x) \pmod{2^M}$ is $> \frac{2}{3}2^{|x|^c}$ or not. Notice that g' is in fact a #P function because a machine can branch on all y and then compute $g(x, y, 0^M)$ on each branch.

All that remains is the proof of the claim.

Proof of Claim 2. The base of the construction is the function $h(z) = 4z^3 + 3z^4$ which has the nice property that

$$z \equiv -1 \pmod{2^M} \Rightarrow h(z) \equiv -1 \pmod{2^{2M}}$$
$$z \equiv 0 \pmod{2^M} \Rightarrow h(z) \equiv 0 \pmod{2^{2M}}$$

Given this property of h, $g(z)$ is the result of applying h recursively $m = \log M$ times to $f(z)$. The claimed property of $g(z)$ follows from the property of h. Consider the running time of the underlying NTM machine whose number of accepting paths is g. Each application of h increases the running time by a constant factor, so overall g is a factor $2^{O(m)} = M^{O(1)}$ slower than f - thus $g \in$ #P.

All that remains is to prove the claimed property of h. Suppose $z \equiv b \pmod{2^M}$, that is $z = q2^M + b$ for some integer q. Then if we look at the expansion of $h(z)$ and remove all terms with a factor of 2^{2M} or higher power of 2^M, we get $h(z) \equiv 4(3b^2q2^M + b^3) + 3(4b^3q2^M + b^4) \pmod{2^{2M}}$. For $b = 0$, this gives us $0 \pmod{2^{2M}}$, and for $b = -1$, $-1 \pmod{2^{2M}}$. □

4 Next time

This lecture concludes the first part of the course that has focused on the traditional setting of complexity theory - namely efficiently computing relations within various computational models. We focus in the remainder of the course on alternative goals within complexity theory. Many of the lectures will take a brief look at an area within complexity theory - each of which could be studied within an entire course. These lectures will often be something like surveys of the area. In fact, we have one more lecture where we are interested in the efficiency of computing relations: next lecture we discuss the power of quantum computing.

Appendix

A Alternate proof of approximate counting

Here we given an alternate proof of Theorem 1 that uses the notion of isolation.

Proof. Let the underlying NTM for f run in time n^c, and let us view it as a polynomial-time verifier. The NTM takes a certificate y of length n^c along with input x. We wish to determine the size of the set S_x of certificates causing the NTM to accept. Consider applying a randomly chosen hash function to the set of possible certificates, with $h : \{0,1\}^{n^c} \to \{0,1\}^m$ for some value m. If 2^m is small compared to S_x, we expect that there are many collisions between members of S_x. If 2^m is large compared to S_x, we expect few collisions. If we are able to determine the relative number of collisions for each value of $m = 1, 2, ..., n^c$, we can come up with an estimate for $|S_x|$.

These ideas are formalized by using the concept of isolation. Let $\mathcal{H}$ be a universal family of hash functions from $\{0,1\}^{n^c}$ to $\{0,1\}^m$. $y \in S_x$ is *isolated* by $h \in \mathcal{H}$ if for all $y' \in S_x$ not equal to y, $h(y) \neq h(y')$. If S_x is small compared to 2^m, then a large portion of S_x should intuitively be isolated by a randomly chosen h. In this case, only a small number of hash functions should be required to guarantee that each $y \in S_x$ is isolated by at least one of them. On the other hand, if S_x is large compared to 2^m, we will show that no small set of hash functions can isolate each $y \in S_x$.

We now quantify these ideas. We first bound the probability that a fixed $y \in S_x$ is not isolated by a random h.

$$\begin{aligned} \Pr_{h\in\mathcal{H}}[y \text{ not isolated by } h] &\overset{(a)}{=} \Pr_{h\in\mathcal{H}}[\bigvee_{y'\in S_x, y'\neq y} h(y') = h(y)] \\ &\overset{(b)}{\leq} \sum_{y'\neq y\in S_x} \Pr_{h\in\mathcal{H}}[h(y') = h(y)] \overset{(c)}{=} \sum_{y'\neq y\in S_x} \frac{1}{2^m} \overset{(d)}{<} \frac{|S_x|}{2^m}. \end{aligned} \tag{5}$$

(a) is by definition of isolation; (b) is by union bound; (c) is because h is chosen at random from a universal family of hash functions; (d) is summing over all $y' \neq y \in S_x$.

Now consider the probability that for a random choice of t hash functions, $y \in S_x$ is not isolated by any of them. Because the events (y not isolated by h_i) for a fixed y are independent for independently chosen h_i, we have

$$\Pr_{h_1,...,h_t\in\mathcal{H}}[y \text{ not isolated by any of } h_1, ..., h_t] = \left(\Pr_{h\in\mathcal{H}}[y \text{ not isolated by } h]\right)^t < \left(\frac{S_x}{2^m}\right)^t. \tag{6}$$

Now consider the probability that there is at least one $y \in S_x$ not isolated by any of $h_1, ..., h_t$. A union bound gives

$$\begin{aligned} \Pr_{h_1,...,h_t\in\mathcal{H}}[S_x \text{ not isolated by } h_1, ..., h_t] \leq & \sum_{y\in S_x} \Pr_{h_1,...,h_t\in\mathcal{H}}[y \text{ not isolated by } h_1, ..., h_t] \\ & < |S_x| \left(\frac{|S_x|}{2^m}\right)^t = \frac{|S_x|^{t+1}}{2^{mt}} \end{aligned}$$

meaning the probability is less than 1 when $2^m > |S_x|^{(t+1)/t}$. This gives us a method for testing whether 2^m is roughly at least as large as S_x. Namely, for all large enough m, we know there are a choice of $h_1, ..., h_t$ isolating all of S_x. We also would like a method for testing whether 2^m is roughly

at most as large as S_x. Notice that each h_i can isolate at most 2^m elements of S_x, so $h_1, ..., h_t$ can isolate at most $t2^m$. Then if $t2^m < |S_x|$, there can be no $h_1, ..., h_t$ isolating all of S_x.

Now let $t = m$ for simplicity. By the above discussion, for all $m = 1, 2, ..., \log(|S_x|) - \log\log(|S_x|)$ there can be no set of hash functions $h_1, ..., h_m$ isolating all of S_x; while for all $m = 1 + \log(|S_x|), 2 + \log(|S_x|), ..., n^c$ there do exist $h_1, ..., h_m$ isolating all of S_x. This gives us a method to estimate the size of S_x: test the predicate

$$(\exists h_1, ...h_m \in \mathcal{H})(\forall y \in S_x)[\vee_{i=1}^{m}(h_i \text{ isolates } y)] \tag{7}$$

for each value $1, 2, ..., n^c$ and determine the first value m^* for which the predicate evaluates to true.

Claim 3. *(7) is a Σ_2^p predicate.*

We finish the analysis given this claim, then prove the claim. From the discussion above, we know $\log|S_x| - \log\log|S_x| < m^* < 1 + \log|S_x|$ which can be rewritten as

$$\frac{|S_x|}{2\log|S_x|} < 2^{m^*-1} < |S_x|. \tag{8}$$

As $\log|S_x| \leq n^c$, we have an approximation for $|S_x|$ that is within a fixed polynomial factor and is computable in $\mathrm{P}^{\Sigma_2^p}$. We can then use the same method as given in section 2 to make the approximation ratio any polynomial.

All that remains is to verify that (7) is in Σ_2^p. This is the point where we use the fact that we are choosing the h_i from a universal family of hash functions rather than at random. Because we are choosing from $\mathcal{H}$, the initial existential guesses are polynomial in size. We claim that the remaining predicate $(\forall y \in S_x)[\vee_{i=1}^{m} h_i \text{ isolates } y]$ is a coNP predicate. This is realized with the predicate

$$(\forall y \in \{0,1\}^{n^c})(\exists i \in \{1, ..., m\})(\forall y' \in \{0,1\}^{n^c})[y \in S_x \wedge y' \in S_x \Rightarrow h_i(y) \neq h_i(y')].$$

Testing $y \in S_x$ is done in polynomial time by evaluating the NTM when given y as a certificate, and the existential phase can be pushed inside since it is of polynomial size. Hence, this is a Σ_2^p predicate.

We have shown that we can approximate $f(x)$ deterministically using a Σ_2^p oracle. In fact, this can be done using a randomized algorithm with a NP oracle. Consider (7), and let us pick the hash functions at random. For large enough values of m, most hash functions satisfy the inside coNP predicate, while for small enough values of m no set of hash functions can satisfy the predicate. Then the m^* derived by randomly selecting hash functions and querying the inside coNP predicate with high probability still satisfies (8). Notice that the estimate for $|S_x|$ derived errors only in one direction - no choice of random functions can satisfy the inside coNP predicate of (7) for small values of m. □

CS 810: Complexity Theory 3/19/2007

Lecture 21: Quantum Effects

Instructor: Dieter van Melkebeek Scribe: Seeun William Umboh

Today we will take a quick look at the quantum computing model. We will define the quantum model of computing as a variant of the probabilistic model of computing, present a common technique in the design of quantum algorithms, and give currently known upper bounds on the power of quantum computation.

1 Motivation

Quantum computation presents a challenge to the *Strong Church-Turing Thesis*, which says that every physically realizable computing device can be efficiently simulated by the Turing machine model presented in the first lecture. It is not clear at this point that problems solvable in polynomial time on quantum computers can be solved in polynomial time on deterministic Turing machines or randomized machines. One reason is that we know how to factor efficiently on quantum machines but not on classical machines. However, the consensus in the community is that quantum computers cannot solve NP-complete problems and some even think that there is an efficient classical factoring algorithm. Another caveat is that it is still uncertain whether quantum computers are actually physically realizable.

2 Idea

We would like to exploit quantum effects to solve computational problems more efficiently, in terms of time. The key idea for search problems is that we would like to use quantum interference in such a way that the "good" solutions interfere constructively and the "bad" solutions interfere destructively. So, in the end, only the "good" solutions remain.

3 Turing Machine Models

3.1 Probabilistic

It is useful to relate the quantum model to the probabilistic model by viewing the probabilistic model from the perspective of Markov chains. We give an alternate definition of the probabilistic model, and then define the quantum model.

Definition 1 (State). *We represent the state of the probabilistic machine as a probability distribution over configurations using the* "ket" *notation:*

$$\sum_c p_c |c\rangle$$

where $\sum_c p_c = 1, p_c \in [0,1]$. p_c *is the probability of being in configuration* c *and* $|c\rangle$ *is the column vector with zeros everywhere except a 1 at the position representing the configuration* c. c *runs over all configurations.*

Definition 2 (Computation). *The computation on a probabilistic machine consists of 2 phases:*

- *A sequence of local(acting only on a few bits, such as the tapehead, state, etc) linear* [1] *transformations that transform a probability distribution into a probability distribution,* i.e. *stochastic matrices, induced by the transition function* δ [2]*:*

$$\delta : Q \times \Gamma^k \times Q \times \Gamma^k \times \{L, R\} \to [0, 1]$$

- *Final observation of part of configuration* C*:*

$$\Pr[ANSWER\ is\ y] = \sum_c p_c$$

where c *runs over all configurations giving the answer* y*.*

From the Markov chain view, at each point in time, the *state* of the machine is a superposition of all possible configurations, represented by the column vector $\sum_c p_c|c\rangle$. Note that the set of single-configuration vectors $\{|c\rangle\}_c$ then forms a basis for the linear space, over $\mathbb{R}$, of all state vectors. Then, at the end of the computation, we observe the output bit(s) and the probability of observing, say 1, is the probability that the machine is in some configuration giving the output 1. This is so far merely a rephrasing of the probabilistic model we presented in an earlier lecture. Now we move on to the quantum model.

3.2 Quantum

Definition 3 (State). *The state of a quantum machine is defined as a linear superposition of all possible configurations* c

$$\sum_c \alpha_c |c\rangle$$

where $\alpha_c \in \mathbb{C}, \sum_c |\alpha_c|^2 = 1$*.*

Definition 4 (Computation). *The computation on a quantum machine consists of 2 phases:*

- *A sequence of local(acting only on a few bits, such as the tapehead, state, etc) linear transformations that transform a vector* v *with* $\|v\|_2 = 1$ *into a vector* v' *with* $\|v'\|_2 = 1$*,* i.e. *unitary matrices, induced by the transition function* δ*:*

$$\delta : Q \times \Gamma^k \times Q \times \Gamma^k \times \{L, R\} \to \mathbb{C}$$

- *Final observation of part of configuration* C*:*

$$\Pr[ANSWER\ is\ y] = \sum_c |p_c|^2$$

where c *runs over all configurations giving the answer* y*.*

[1] We would like the transformation to depend on the configuration not on the overall distribution.

[2] We usually require the transition probabilities to be efficiently approximable, for example the rationals, to avoid dealing with machines with say, the halting sequence as a transition probability.

Here, we view the state of the quantum machine as a wave function. The coefficients, called *amplitudes*, are vectors in the imaginary plane with length at most 1. The fact that the amplitudes can be negative, allowing destructive interference, is what underlies a lot of the power in quantum computing. Also, note that the probability of being in a particular configuration is now the square of the absolute value of the amplitude associated with it.

One issue we need to consider is that the condition that the matrices induced by δ be stochastic or unitary imposes restrictions on the set of allowable transition functions. In the probabilistic setting though, we only need that the transition function be "stochastic". That is, for a fixed configuration, the sum of the probabilities of the configurations it can move to in one step is 1. In particular, we can show that setting the transition probability to be either 1, 0 or 1/2 is enough. In the quantum setting, the natural thing to do is to have δ be "unitary", in the appropriate sense. However, it turns out that δ has to satisfy some *orthogonality conditions* as well. These conditions are unnatural and so instead of considering quantum machines as Turing machines during algorithm design, we prefer to think in terms of circuits.

4 Circuit Models

We now define circuit models for both probabilistic and quantum computations. In order to satisfy the above conditions, we would like the gates to act on a finite number of bits, in particular it is sufficient that they act on at most 3 bits, and to induce stochastic(unitary) matrices. To this end, we define the following notions:

- The *register* is the analogous notion of Turing machine configuration in the circuit model. The contents of the register reflect the results of the computation so far. We retain the notation $|x_o \dots x_m\rangle$ to denote the contents of the register.
- The *state* of a register is represented as a probability distribution(linear superposition) over all possible contents of the register. Again, we note that the set of vectors $\{|c\rangle\}c$ form a basis for the linear space, over the reals(complex numbers), of all state vectors.
- An *operation* G on an m-bit register is specified by a linear stochastic(unitary) transformation $F : \mathbb{R}^{2^3} \to \mathbb{R}^{2^3} (F : \mathbb{C}^{2^3} \to \mathbb{C}^{2^3})$ acting on 3 distinct bits with indices $j, k, l \in \{1, \dots, m\}$, leaving others unmodified, such that for every $x_1, \dots, x_m \in \{0, 1\}$, applying G on $|x_1, \dots, x_m\rangle$ gives $|y_1, \dots, y_m\rangle$ where $|y_j y_k y_l\rangle = F(|x_j x_k x_l\rangle)$.
- A computation consists of a sequence of operations followed by a final observation of the register.

Note that the operations can be represented by stochastic(unitary) matrices, hence the models defined satisfy that condition.

For uniformity, we can simply require that a single Turing machine can compute the operation at step i, for all i. In addition, for the $T(n)$-time bounded versions of these models, we require that the Turing machine take time at most $T(i)$ to compute the operation at step i.

4.1 Probabilistic

For the probabilistic model, it is sufficient to have classical AND, OR and NOT gates to simulate deterministic computation, and a *coin flip* gate to allow access to a fair coin.

4.1.1 Deterministic Gates

The AND, OR and NOT operations are defined by the transformations F, G, H with $F|xyz\rangle = |xy(x \wedge y)\rangle, G|xyz\rangle = |xy(x \vee y)\rangle, H|x\rangle = |\bar{x}\rangle$, respectively. The matrix for AND is:

$$F = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Remember that the input bits are unaffected by the gate, and since we multiply F from the right, $F_{i,j}$ denotes the probability of the register having contents $|i_2 i_1 i_0\rangle$ if it started in configuration $|j_2 j_1 j_0\rangle$ where l_k is the kth bit of l, from the right.

4.1.2 Coin Flip Gate

The coin flip gate acts only on one bit and its output is 1 or 0 equiprobably:

$$C = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

4.2 Quantum

4.2.1 Deterministic Gates

In the quantum setting, we cannot simply use the matrices for the AND, OR and NOT gates as above, since they do not preserve the 2-norm. We note first that the 2-norm preserving condition for a matrix A is equivalent to it satisfying $A^*A = I$ since $||Ax||_2^2 = x^*A^*Ax = x^*x$ iff $A^*A = I$.[3] Hence, the gate matrices have to be at least invertible. However, the AND and OR gates are not invertible, as more than one input can map to 0 under AND, for example.

For the deterministic gates to be reversible, they need to induce permutation matrices as determinism requires the matrix to have in every column a 1 in exactly 1 entry, and zero everywhere else, and reversibility requires the matrix to have in every row a 1 in exactly 1 entry, and zero everywhere else. We can get around this by introducing additional ancilla bits. Given a boolean function $\varphi : \{0,1\}^k \to \{0,1\}$, we transform it to the reversible version $g : \{0,1\}^{k+1} \to \{0,1\}^{k+1}$, defined by $(x, b) \to (x, b \oplus \varphi(x))$.

So now we can apply this transformation to all the gates of any deterministic classical circuit to obtain a circuit usable by our quantum machine. Given a circuit C computing $f : \{0,1\}^n \to \{0,1\}$, the resulting function after transforming C's gates is $f' : \{0,1\}^{n+m+1} \to \{0,1\}^{n+m+1}$, defined by $(x, 0^m, 0) \to (x, \mathit{garbage}, f(x))$, where the number of ancilla bits m is on the order of the number of gates in C.

Now, we need to have the m ancilla bits set to zero, as otherwise the interference patterns would be different. In particular, the garbage in the ancilla bits may cause certain computation paths to

[3] By A^*, we mean the complex conjugate of A.

not interfere when we would like them to. Since simply resetting the bits to zero is an irreversible operation, we do it by applying f' in reverse, and using an extra bit to preserve the result computed by f':

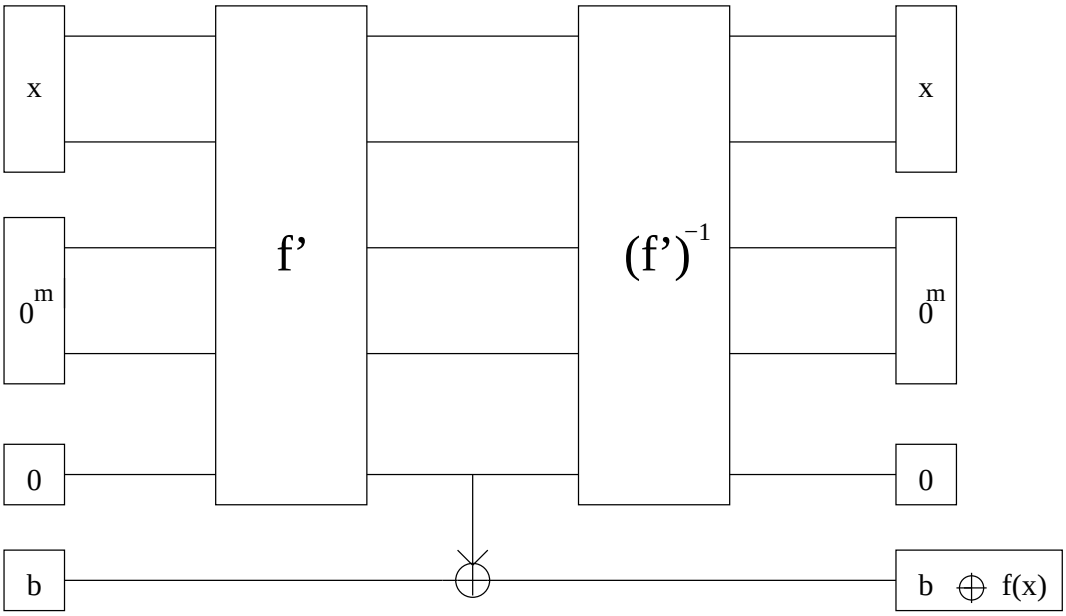


Figure 1: The schematic for the reversible simulation of f.

The complexity of this reversible simulation, denoted as $\tilde{f}$, is a constant factor greater than the complexity for f since we only need to apply the transformation to every gate, and then run the transformed circuit twice.

Note that this also gives us P $\subseteq$ BQP, where BQP is the class of decision problems solvable in polynomial on quantum Turing machines. Since the output of a quantum Turing machine is a random variable, we say that it decides a language L if the probability of deciding correctly the membership of x in L is at least $2/3$.

4.2.2 Hadamard Gate

The quantum analog of the classical coin flip gate is called the *Hadamard gate*. The matrix for this gate is

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Exercise 1. *Verify that the Hadamard matrix is unitary and orthogonal.*

The effect of applying this gate to a single bit is:

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

$$H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

So, if we observe the bit after applying the Hadamard gate, we get 0 or 1 equiprobably, just as in a fair coin flip. However, we also have that $H^2 = I$. So, applying the Hadamard gate twice consecutively in sequence, will give us the original bit, which is not the same result as if we did the same with the coin flip gate. One way to see this is in terms of interference amongst the computation paths.

Hence, we can simulate randomized computation by applying the Hadamard gate to some 0s and then observing those bits. Afterward we can proceed deterministically and use those bits as random bits. Thus, BPP $\subseteq$ BQP.

5 Illustration of Quantum Power

Even though it is widely believed in the community that interference is not enough to solve NP-complete problems, we have managed to use interference to efficiently solve certain problems for which we do not have efficient classical algorithms. An example is factoring, and *Simon's Problem*. While this problem is unnatural, the technique used in solving it underlies many quantum algorithms such as factoring.

Definition 5 (Simon's Problem). *Given a poly-time length-preserving(it maps strings of length n to strings of length n) function $f : \{0,1\}^* \to \{0,1\}^*$ with the promise that*

$$(\forall n)(\exists 0 \neq s_n \in \{0,1\}^n)(\forall x, y \in \{0,1\}^n)[f(x) = f(y) \iff x + y = s_n],$$

where addition is defined as bitwise XOR, find s_n on input 0^n.

In other words, for strings of length n, the function f is exactly 2-to-1 and the 2 strings that map to a certain string differ by a *shift* s_n, and we would like to find s_n. Also note that this is a promise problem.

This problem is in the second-level of the Polynomial Hierarchy since we can guess a shift and then verify for all pairs (x, y) of n-length strings that $f(x) = f(y) \iff x+y = s_n$. f is a poly-time function and we can easily bitwise-XOR x with y and see if we get s_n, so the verification can be done efficiently.

For classical machines, we do not know how to do better than exponential time. Even using randomness, the best algorithm we have is to guess x and y and see if $f(x) = f(y)$. If we get a collision, then we can easily determine s_n. However, the expected number of trials to get a collision is exponential.

On quantum machines though, this problem is solvable in polynomial time. We first start off in the state $|0^n\rangle|0^n\rangle$, where we would like to use the first n bits to encode all possible inputs for f. We achieve this by applying the Hadamard gate to each of the n bits. We denote this operation as $H^{\otimes^n}$. [4] So, now we are in the state $\frac{1}{\sqrt{N}} \sum_x |x\rangle|0^n\rangle$, where $N = 2^n$. Then, we apply $\tilde{f}$ with the first n bits as input bits and the last n bits as the output bits, and get the state $\frac{1}{\sqrt{N}} \sum_x |x\rangle|f(x)\rangle$. Finally, we apply $H^{\otimes^n}$ to the first n bits again and we leave it as an exercise to verify that we end up in $\frac{1}{N} \sum_x \sum_y (-1)^{x \cdot y} |y\rangle|f(x)\rangle$, where $(\cdot)$ denotes inner product. We can rewrite this as $\sum_{y,z} \alpha_{y,z} |y\rangle|z\rangle$, a linear superposition over all possible contents of both registers.

So, the probability of observing y in the first register is $\sum_z |\alpha_{y,z}|^2$. Now, if z is not in the range of f, $\alpha_{y,z}$ is 0. Otherwise, z is in the range and f maps exactly 2 strings to it, so $\alpha_{y,z} =$

[4] In linear algebra, this is the n-fold tensor product of the Hadamard gate matrix.

$\frac{1}{N}[(-1)^{x\cdot y} + (-1)^{(x+s_n)\cdot y}]$ where $z = f(x)$ for some x. Because we can take out the common factor $(-1)^{x\cdot y}$, and since if we sum over all n-length strings, we will count each $f(x)$ twice(as $f(x)$ and $f(x + s_n)$), the probability that a particular y is observed is $\frac{1}{2}\sum_x \frac{1}{N^2}|1 + (-1)^{s_n\cdot y}|^2$. As there are $2^n = N$ possible x's, this is then $\frac{1}{2N}$ if $s_n \cdot y = 0 \mod 2$, and 0 if $s_n \cdot y = 1 \mod 2$.

Therefore, the output of this computation y is chosen uniformly at random from those y such that $s_n \cdot y = 0 \mod 2$. We observe that $s_n \cdot y = 0 \mod 2$ is a linear combination of the components of y where the coefficients are the corresponding components of s_n. This is a linear equation of n variables over $\mathbf{GF}(2)$. So, we can run the above routine an order of n times and collect the output y_i from each run until the equations $s_n \cdot y_i$ form a homogeneous system of rank $n-1$. From elementary linear algebra, it follows that it has a unique non-trivial solution which is s_n.

We have already argued the efficiency of some of the components of the routine: $H^{\otimes^n}, \tilde{f}$. We also know how to solve homogeneous linear systems of equations efficiently. Lastly, we leave the fact that we only need on average $O(n)$ runs of the routine as an exercise:

Exercise 2. *Verify that with high probability, $O(n)$ runs suffice.*

6 Hidden Subgroup Generalization

Definition 6 (Hidden Subgroup Problem). *Given a group G and a poly-time $f : G \to \{0,1\}^*$, and the promise that there exists some subgroup H of G such that $f(x) = f(y)$ iff x and y belong to the same coset of H in G, find generators for H.*

There are several familiar instances of this problem:
Example:[Simon's Problem] The group for Simon's Problem is $G = (\{0,1\}^n, +)$, where $+$ denotes bit-wise XOR. We have f as defined in Definition 5, and so the subgroup we are interested in is $H = \langle s_n \rangle = \{0^n, s_n\}$, since x and y belong to the same coset of H iff $x = 0^n + y = y$ or $x = s_n + y$. ⊠

This next example is critical for the factoring algorithm, although we do not discuss the factoring algorithm here..
Example:[Finding Order r of a mod b] Given a, b relatively prime, we would like to find the order of $a \mod b$. So, we are interested in the group $G = (Z_b, \cdot)$, with $(\cdot)$ denoting multiplication mod b, and the poly-time function $f(x) = a^x \mod b$. Then, H is $\langle r \rangle$ since $f(x) = f(y)$ iff $r|(x-y)$. This can be solved in a way similar to Simon's Problem but we use a general *Fast Fourier Transform* instead of the Hadamard gate, which is just a Fourier Transform over $\mathbf{GF}(2)$. ⊠

Example:[Discrete Log] We leave this as an exercise for the reader. ⊠

Example:[Graph Isomorphism] Consider 2 connected graphs G_1, G_2 on n vertices. In order to cast this as a hidden subgroup problem, we consider the symmetric group $G = S_{2n}$ on $2n$ elements, and the function $f(\pi) = \pi(G_1 \cup G_2)$ which gives the result of applying the permutation π on the vertices of the disjoint union of the 2 graphs. So, H is $\mathrm{Aut}(G_1 \cup G_2)$ since $f(\pi) = f(\sigma)$ if and only if there is some automorphism such that applying the automorphism after applying π gives the same result as applying σ. Note that the graphs are isomorphic if and only if some generator of H swaps G_1, G_2, since we can decompose automorphisms into automorphisms that do not swap G_1, G_2 and those that do. Thus, if we can find the generators for H, we can easily determine if the graphs are isomorphic. ⊠

The main technique in the algorithms for some of these examples is *Fourier sampling*. We set up a linear superposition of all possible inputs to f in the first register, store the output of f in the second register, apply an appropriate Fourier transform and then measure the system. However, this technique only works when the group is abelian, which is the case for factoring, Simon's problem, and discrete logarithm. So, in the case of Graph Isomorphism, since the symmetric group is not abelian, we cannot use the same technique. More specifically, the probability distributions given by Fourier sampling on positive instances of Graph Isomorphism and negative instances are indistinguishable.

7 Upper Bounds

Now that we have seen some of the power of quantum computers, we would like to see if we can upper bound quantum computers. Note that the outcome of a quantum computation depends on the, possibly negative, amplitudes of paths. Thus, the probability distribution is a GapP function. Thus, we get that BQP $\subseteq$ $P^{\#P[1]}$, and also BQP $\subseteq$ PP(this is a class we will define in a future lecture).

The biggest open problem on quantum computation in complexity theory is whether or not BQP is contained within the Polynomial Hierarchy. In the probabilistic setting, approximate counting was good enough to show containment within the Hierarchy. In the quantum setting however, the possibility of interference makes it harder to just rely on approximate counting. In fact, it is conjectured that exact counting is required.

On the physics side, we do not yet know if we can build reliable and scalable quantum computers. So far, 2 models have been proposed. The first uses the spin of a single electron trapped in a silicon lattice. This scales up well once we can implement it on a few bits, since silicon is abundant. However, we are still trying to implement it on those few bits. Another model is the optical lattice. In this model, the electron is trapped in an optical lattice using lasers. While we have managed to implement some limited quantum computers using this model, scaling is a problem since the bottleneck is now laser power.

8 Next Lecture

With today's lecture, we have completed the first part of the course where we look at different models of computation and compare them with regards to time and space resources. In the next lecture we will start the next section of the course, where our goal is to use metrics other than time and space.

CS 810: Complexity Theory	3/21/2007
Lecture 22: Proof Complexity	
Instructor: Dieter van Melkebeek	Scribe: Nathan Collins

In the last lecture we discussed quantum computation. Today we start the second part of the course, where we consider non-standard complexity measures. The standard complexity measures are time and space complexity. Today's complexity measure is the size of the minimum proof of a given theorem.

1 Issues

There are some issues to be resolved

- We all know intuitively what a proof is,[1] but we need a precise definition if we hope to talk about the complexity of proofs. Today we formalize proofs using the familiar notion of NP: A proof is something that is "easily" verified, where "easy" means in time polynomial in the length of the proof. This is like NP, in that the proof is a witness or certificate, but unlike NP, we don't require that the proof be of size polynomial in the size of the theorem in question. In fact, the central goal of proof complexity is to establish the *non*-existence of polynomial size proofs of propositional tautologies.
- We want our proof systems to have two properties:
  - Soundness: No contradiction can be proved.
  - Completeness: All true theorems have a proof.
- We need to decide what sort of things we want to prove.[2] The more the statements under consideration are restricted the more simple the systems will be and (hopefully) the more likely we will be able to prove things about them. Proof complexity considers systems in which we can establish whether or not a logical proposition is a propositional tautology. Since TAUT, the class of propositional tautologies, is co-NP complete, we can answer the NP vs co-NP question by understanding how "difficult" TAUT is. The consensus is that TAUT does not have poly size proofs, although this has only been proven in a few specific proof systems.

2 Definition and Central Theorem

Definition 1 (Propositional proof system)**.** *A* propositional proof system *(PPS) is a poly-time mapping from* $\{0,1\}^*$ *onto TAUT, s.t. if* $f(x) = y$ *then* $x \vdash y \in TAUT$*, where "*$\vdash$*" means "proves."*

[1] One interesting definition is a repeatable experiment in persuasion.

[2] Gödel's Incompleteness Theorems say something like soundness and completeness can't be proved in systems strong enough to express arithmetic. Here we allow other means than the system itself for proving soundness though.

Intuitively, given a tautology y, a PPS ensures there is a proof x that is efficiently checkable by f proving $y \in$ TAUT. We are assuming that we have some encoding of proofs as strings and when x isn't a valid proof we'll map it to a trivial tautology, e.g. T.

Notice that soundness and completeness are implicit in the definition

- Soundness: Since $f(\{0,1\}^*) \subseteq$ TAUT.
- Completeness: Since TAUT $\subseteq f(\{0,1\}^*)$.

Definition 2 (Polynomial bounded PPS)**.** *A PPS is* polynomially bounded *if each $y \in TAUT$ has a proof that is polynomial in length, i.e. there exists a constant c s.t.*

$$(\forall y \in TAUT)(\exists x \in \{0,1\}^{\leq |y|^c}) f(x) = y.$$

With these definitions we can precisely state the theorem (mentioned in the last section) that relates proof complexity to the NP vs co-NP question.

Theorem 1. *There exists a polynomially bounded PPS iff* NP $=$ *co*-NP*.*

Proof. The proof is very easy

- $\Rightarrow$ A polynomially bounded PPS gives a NP machine that decides TAUT. Namely, the machine guesses the proof x and verifies that it is a valid proof by computing $f(x)$ and checking that it equals y.
- $\Leftarrow$ co-NP is $\leq_m^p$-complete for TAUT.

□

So, proof complexity tries to establish NP $\neq$ co-NP by establishing super-polynomial bounds on the proof complexity of TAUT. So far this has consisted of establishing such bounds in a few particular example systems.

3 Proof Systems

Many systems can be cast more easily as *refutation systems*. Since $y \in$ SAT iff $\neg y \in$ TAUT we think of a proof of membership in TAUT as refuting membership of the negation in SAT. DNFs are the natural form for TAUT, as CNFs are the natural form for SAT formulas. Without loss of generality, we focus on refuting CNFs – proving that CNF formulas are not in SAT. There is a general refutation strategy. To refute a formula φ we assume it is true and derive a contradiction. Derivations start from axioms and the assumption that φ is true, and then proceed using a set of sound derivation rules to arrive at a contradiction. The axioms and derivation rules are part of the proof system being used.

The most common refutation systems, and the ones we'll consider here are

- Logic-based: Manipulate boolean formulas.
- Geometry-based: Manipulate linear inequalities over $\mathbb{R}$.
- Algebra-based: Manipulate multivariate polynomials.

3.1 Logic-Based

The most trivial logic-based proof system just uses truth tables for proofs. Proofs in this system are clearly of exponential size. A more interesting class of examples are *Frege systems*, which are defined by a collection of axioms including

- Axioms in the form of easily recognizable tautologies.
- Derivation rules.

Example: A common axiom in Frege systems is the tautology $A \vee \overline{A}$. ⊠

Example: The *cut rule*

$$\frac{A \vee B \qquad \overline{A} \vee C}{B \vee C}$$

is a common derivation rule in Frege systems. The cut rule is read as "any assignment to the boolean variables that satisfies $A \vee B$ and $\overline{A} \vee C$ simultaneously satisfies $B \vee C$ too." Notice that the cut rule is intuitively valid. ⊠

A proof that a formula is not satisfiable in a logic based system is called a refutation.

Definition 3 (Refutation)**.** *A* refutation *of a CNF φ is a sequence of formulas, ending in the empty clause, s.t. each formula is a clause from φ, an axiom, or follows, via derivation, from some previously derived formulas in the sequence. The empty clause is the basic contradiction in Frege Systems.*

Frege systems are usually further classified based on the complexity of the formulas allowed in the proof. The following are possible restrictions

- Every line of the proof is a clause from φ.
- Constant-depth Frege: Every line of the proof is computable by a constant depth circuit.
- Unrestricted Frege: Lines are arbitrary formulas. Earlier in the course we noted that boolean formulas are equivalent to circuits with fanout 1.
- Named sub-formulas: Allowed to have "lemmas" that can be instantiated multiple times. This can significantly decrease proof size. This can be viewed as allowing each line of the proof to be computed by circuits of fanout ¿ 1. Generally, the size and depth of a circuit can be significantly decreased by allowing fanout greater than 1.

3.1.1 Resolution

A commonly studied proof system consists of using only the cut rule.

Definition 4 (Resolution)**.** *A Frege system without axioms in which all proof lines are clauses and the cut-rule is the only derivation is called a* resolution *system.*

Notice that soundness is immediate in a resolution system, since cut can be proved valid, and any proof of the empty clause $\emptyset$ must end with the derivation

$$\frac{x \vee \emptyset \qquad \overline{x} \vee \emptyset}{\emptyset \vee \emptyset = \emptyset},$$

justifying our taking $\emptyset$ to be the basic contradiction. Completeness is harder, since we need to be able to derive the empty clause from any unsatisfiable φ.

Completeness follows from the Davis-Putnam procedure. Here we give an example of the procedure on the unsatisfiable formula

$$\begin{aligned}\varphi =&(x \vee y) \wedge (x \vee z) \wedge (y \vee z)\\ &\wedge(\overline{x} \vee \overline{y}) \wedge (\overline{x} \vee \overline{z}) \wedge (\overline{y} \vee \overline{z}).\end{aligned}$$

The formula φ is not satisfiable since the first line needs two true variables and the second line needs two false variables.

The Davis-Putnam procedure generates a resolution refutation demonstrating the unsatisfiability of φ using a binary-tree search on variable assignments. The tree for φ is given in Figure 1. The internal nodes of the tree are labeled by variables and a path in the tree corresponds to an assignment to the variables labeling nodes on the path. The leaves of the tree are labeled by clauses that are violated by the assignment corresponding to the path that reaches them. Once

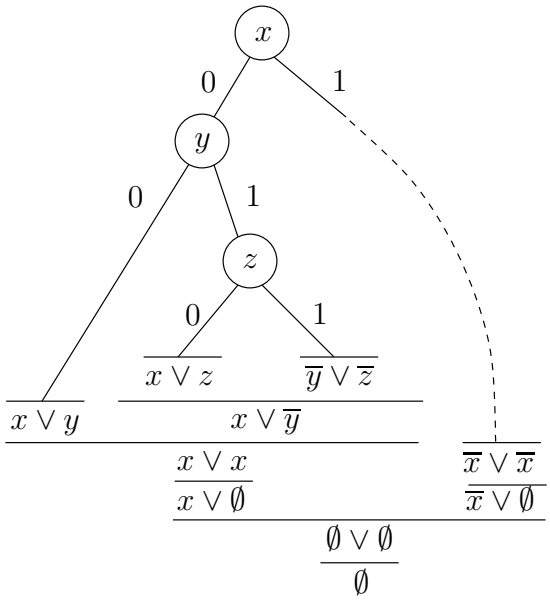


Figure 1: The right hand side of the tree, and corresponding derivations, are analogous to the left hand side and have been omitted. Notice that the derivations mirror the tree.

the tree has been constructed, a resolution refutation is constructed by working inward from the leaves, using the cut-rule on adjacent leaves to collapse the tree.

The Davis-Putnam procedure is used in many automated theorem provers, although it's easy to end up with exponential proofs when using Davis-Putnam to derive a resolution refutation.

3.2 Geometry-Based

The common geometric PPS is called *cutting planes*. A cutting planes system is a set of linear inequalities[3] with integer coefficients.

- Axioms: Unlike the Frege systems above, which had universal axioms, in cutting planes the axioms are specific to the formula in question. More precisely, if $\varphi = \bigwedge_i C_i$ is the CNF we are trying to refute, then there will be one axiom for each clause C_i in φ. For example, for the clause $x \vee \overline{y} \vee z$ we get the inequality $x + (1 - y) + z \geq 1$, which is satisfied by values x, y, and z in $\{0, 1\}$ only when $(x \geq 1) \vee (y \leq 0) \vee (z \geq 1)$.
- Derivation rules:
  - Addition of "same direction" inequalities:
$$\frac{f \geq a \qquad g \geq b}{f + g \geq a + b}$$
  - Multiplication by a positive integer:
$$\frac{f \geq a \qquad c \in \mathbb{N}}{cf \geq ca}$$
  - Rounding:
$$\frac{(f = \sum_i a_i x_i) \geq a \qquad (\forall i) c \mid a_i}{\frac{f}{c} \geq \lceil \frac{a}{c} \rceil}$$
    Notice that rounding is *not* valid if we allow non integer values for the x_i.
- The basic contradiction is $0 \geq 1$.

Soundness is easy like with Frege systems, since the axioms are again "true." Completeness is again non-obvious, and we won't argue it here.

3.3 Algebra-Based

The *polynomial calculus* is a popular algebra-based PPS. In the polynomial calculus the lines in the proofs are multivariate polynomials. The polynomial calculus is based on the following theorem of Hilbert

Theorem 2 (Hilbert's Nullstellensatz)**.** *In any algebraically closed field*[4] *a set $P_1, \ldots, P_n$ of (multivariate) polynomials has no common root iff there exist another set $Q_1, \ldots, Q_n$ of polynomials s.t.*
$$\sum_i P_i Q_i \equiv 1.$$

[3] A linear inequality determines a hyper-half-plane that "cuts" space into two regions, one admissible and one not.
[4] All polynomials over the field have a root in the field. $\mathbb{C}$ is the most common example.

One direction of the proof is easy: If $\sum_i P_i Q_i \equiv 1$ then the P_i certainly don't share a root. The other direction is more difficult.

In the polynomial calculus we arithmetize clauses by producing polynomials that have roots corresponding to satisfying assignments for each clause. For example, for the clause $x \vee \overline{y} \vee z$ we considered above, we get the polynomial $(1-x)y(1-z)$. A refutation resolution consists of finding the polynomials Q_i. This accomplished by starting with polynomials corresponding to the formula, and using a series of multiplications of polynomials to arrive at the constant 1.

- Axioms: One polynomial $x(1-x)$ for each variable x, to force the allowable values taken by the variables to be zero and one.
- Derivation rules: If f and g are polynomials
  - If a and b are polynomials then
$$\frac{f \qquad g}{af + bg}$$
  - If x is a variable then
$$\frac{f}{xf}$$
- The constant polynomial 1 is the basic contradiction.

For the logical and geometric systems above the lines in the proofs are inherently short. In the polynomial calculus we can multiply polynomials though, which can lead to exponential size lines ($\Pi_i(x_i + y_i)$). In the polynomial calculus we let the degree be the measure of a polynomial's size.

4 Lower Bounds

We mentioned earlier that the goal of proof complexity is to establish super-polynomial lower bounds on establishing propositional tautologies. For the systems we have mentioned the best known lower bounds are

- Cutting planes: Exponential.
- Polynomial calculus: Linear degree.
- Unrestricted Frege: Unknown.
- Frege resolution: Exponential.

5 Unsatisfiable CNFs

The following families of unsatisfiable CNFs have proved useful in establishing lower bounds in propositional proof complexity.

5.1 Pigeonhole Principle

- PHP_n^m: The pigeonhole principle for m pigeons and n holes ($m > n$). The CNF has
  - Variables x_{ij} corresponding to the ith pigeon being assigned to the jth hole.
  - Clauses
    - Every pigeon assigned to a hole
    $$\bigwedge_i \bigvee_j x_{ij}$$
    - No hole has two pigeons assigned to it
    $$\bigwedge_k \bigwedge_{i \neq j} \overline{x}_{ik} \vee \overline{x}_{jk}$$

In the Frege resolution system PHP_n^{n+1} has size $2^{\Omega(n)}$ proofs. When $m = n + 1$ the pigeon surplus is minimized and hence this is the case with the highest proof complexity. In general, for arbitrary $m > n$, PHP_n^m has proof complexity of $2^{\Omega(\sqrt[3]{n})}$ in Frege resolution.

In cutting planes, PHP has only polynomial proof complexity.

5.2 Tseitin Formulas

Given a graph $G = (V, E)$ we choose a "charge" function $\sigma : V \to \{0, 1\}$ on the vertices. Once σ has been chosen we ask if there exists a charge assignment γ for the edges s.t. the σ-charge of each vertex is equal to the parity of the γ-charges of the edges incident that vertex. In symbols, we are asking if, given σ,

$$(\exists \gamma : E \to \{0,1\})(\forall v \in V)(\sigma(v) = \bigoplus_{e \in E \text{ s.t. } v \in e} \gamma(e))$$

Exercise 1. *Show that the sum of the vertex charges, $\sum_v \sigma(v)$, must be even if there exists a satisfactory choice γ of edge charges.*

The strongest lower bounds for Tseitin formulas arise when the graphs considered are expanders. Lower bounds of size $2^{\Omega(n)}$ for resolution are achieved by certain expander families.

6 Next Time

Next time we'll talk about more powerful proof systems that are allowed to use randomness.

CS 810: Complexity Theory 3/26/2007

Lecture 23: Interactive Proofs

Instructor: Dieter van Melkebeek Scribe: Seeun William Umboh

Last time, we introduced the notion of proof complexity, where we looked at proof systems for the set of propositional tautologies. We required the proofs to be efficiently verifiable so proofs correspond to non-deterministic computations. So, a language L has short proofs of membership if and only if L is in NP. In particular, if propositional tautologies have short proofs, then NP = coNP.

Today, we relax the notion of proof by allowing randomness in verification, and allow more interaction between the *prover* and *verifier*. In contrast, previously, the verifier was deterministic; the interaction was limited to a single round and was one-way: the prover passes the purported proof to the verifier and the verifier accepts or rejects based on that proof. With the new notion of proof, L has short proofs if and only if L is in PSPACE, and so tautologies have short proofs, unlike in the previous definition of proof.

1 Interactive Proof Systems

Definition 1 (Interactive Proof Systems)**.** *An interactive proof system for L is a* protocol *(P, V), where P is called the* prover*, and V is called the* verifier*. P is an all-powerful(*i.e.*, no restrictions on its computational resources), randomized Turing machine, and V is a randomized Turing machine running in time polynomial in the input length. In a protocol (P, V), we have:*

1. *P, V have read-only access to the input x.*
2. *P, V have read-only access to separate random-bit tapes.*
3. *P can write messages on tape $T_{P\to V}$ which V has read access to.*
4. *V can write messages to P on tape $T_{V\to P}$ which P has read access to.*
5. *$x \in L$ iff V accepts.*
6. *At any one time, exactly one of V and P is active, and the protocol defines how they take turns being active.*

The protocol (P, V) also has to satisfy the following conditions:

- *completeness(easy to prove membership of members):*

$$x \in L \implies \Pr[(V \leftrightarrow P) \text{ accepts } x] \geq c$$

- *soundness(hard to prove membership of non-members):*

$$x \notin L \implies (\forall P')\Pr[(V \leftrightarrow P') \text{ accepts } x] \leq s$$

where the probabilities are over all coin flips of P and V; $(V \leftrightarrow P)$ refers to an interaction between the P and V; c, s are the completeness *and* soundness *parameters of the interactive proof system, respectively. c is normally set to 2/3 and s to 1/3.*

For the soundness condition, we choose to quantify over all provers since the critical issue is that the verifier itself needs to be sufficiently robust even against "malicious" provers that do not follow the protocol. Note that we can also apply the amplification technique to boost the probability of deciding correctly, and the verifier would still be poly-time. For completeness, $c = 1$ means *perfect completeness* and this means that no true statement is rejected, and is something we strive for.

We make the following observations:

1. Randomness for P is not essential. For each turn it takes, since it is all-powerful, it can figure out the coin flips that will maximize the probability of the verifier accepting, and perform the computation corresponding to those coin flips.

2. On the other hand, the randomness of V is essential, as otherwise, we get only NP. If V is deterministic, then it accepts/rejects with probability 1. Then, using the above fact that we can assume P is deterministic, and that $c > 0$, we get that $(V \leftrightarrow P)$ always accepts members of L. Since V can only read and write a polynomial number of symbols, the length of the transcript of the interaction $(V \leftrightarrow P)$ is polynomial in the length of x, and is a polynomial-length certificate of x's membership. In particular, the sequence of responses of P convinces V of x's membership.

3. If we assume perfect soundness, we get NP as well. This is because $x \in L$ iff there exists a sequence of random "questions" by the verifier and a sequence of "answers" by P that convinces V of x's membership. The converse holds by the assumption of perfect soundness. The other direction has a similar proof as above.

4. Without interaction, since the verifier does not receive any information from the prover, we get BPP.

5. By requiring the verifier's random bit tape to be separate from the prover's, the proofs we defined are actually *private coin* interactive proofs. *Public coin* interactive proofs are where the verifier tosses coins and reveals them to the prover after it's turn is up.

The corresponding complexity class is IP $= \{L | L \textit{ has an interactive proof system}\}$.

2 Power of interaction

Let us now illustrate what we can do with interaction and randomness in the verifier. In particular, we would like to be able to obtain short interactive proofs for problems for which we do not know a short standard proof. One example of such a problem is *graph non-isomorphism*(GNI).

Theorem 1. $GNI \in$ IP

Remember that GI is one of the problems that we think are NP-intermediate. Also, GI has short standard proofs. This theorem by itself will not show that the set of tautologies have short interactive proofs(coNP vs IP). GI is probably not NP-complete, and equivalently, GNI is probably not coNP-complete. In fact, we will give some evidence that GI is not NP-complete.

Before we begin with the proof, we give an intuitive analogy for the way the IP protocol for GNI works. One day, Merlin shows a sword to King Arthur and claims that it is Excalibur. However, Arthur is unconvinced since to him it looks exactly like his own sword. So, Arthur would like Merlin to convince him that at least he could tell the difference between Excalibur and Arthur's sword. To this end, Arthur turns around, hides the swords and flips a coin. Based on the coin flip, he produces one of the swords and asks Merlin whether it is the purported Excalibur or Arthur's sword. If Merlin answers incorrectly, then Arthur knows that Merlin had lied. Otherwise, he repeats the procedure until he is convinced.

Note that the usage of the name "Arthur" for the verifier and "Merlin" for the prover is standard in the literature, but for public coin systems. The system below uses private coins.

Proof Sketch. Given 2 graphs (G_0, G_1), we would like to show that they are non-isomorphic. An initial idea, based on the analogy above, is to have V flip a coin, produce one of the graphs G based on the coin flip and ask P which of G_0, G_1 it is. But this is too easy since V could compare G with G_0, G_1 by itself. So, instead we have V pick G at random, permute the vertices of G randomly, and ask P to identify G.

Using $\equiv$ to denote isomorphism, if $G_0 \not\equiv G_1$, then P can identify G by trying all permutations on G_0, G_1 and since $G_0 \not\equiv G_1$ exactly one permutation on exactly one of the 2 graphs give G. Otherwise, there exists permutations π, σ such that $\pi(G_0) = \sigma(G_1) = G$, and so P has no way of identifying which of G_0, G_1 was used to produce G. The best it can do then is flip a coin and guess an answer.

So, this protocol satisfies perfect completeness and has $s \geq 1/2$. We can decrease s by increasing the number of rounds. □

We remark that since the verifier is essentially requiring the prover to determine the result of its coin toss, it is crucial that the protocol uses private coins. We next consider interactive protocols that remain secure even when the verifier's random coins are made public. Although the above protocol for GNI does not work in this setting, there is an alternate protocol that does work with public coins.

3 Bounded-round interactive proof systems

We start with the formal definition of AM. In addition to requiring public coins, we limit the number of rounds of interaction between the verifier and prover.

Definition 2. *AM is a public coin system in which Arthur moves first. He flips some coins to determine what to ask Merlin. Arthur then runs a deterministic predicate on the coins, Merlin's reply and the input.*

Formally, a language L is in AM if for some $V \in \mathrm{P}$:

$$x \in L \implies \Pr_{|z|=n^c}[(\exists y \in \Sigma^{n^c}) V(x, y, z)] \geq 2/3$$

$$x \notin L \implies \Pr_{|z|=n^c}[(\exists y \in \Sigma^{n^c}) V(x, y, z)] \leq 1/3$$

Now we present the AM protocol for GNI.

Theorem 2. $GNI \in AM$

Firstly, we review the *Lower Bound Protocol*. We have some subset $S_x \subseteq \{0,1\}^n$ whose size we would like to estimate. We have a universal family of hash functions from n bits to m bits, where the range of the hash functions is roughly the same as the size of S_x. Then, there is a good chance that a hash function chosen randomly from that family will map S_x "evenly" on $\{0,1\}^m$. In particular, a few hash functions from that family will suffice to cover $\{0,1\}^m$ as an image of S_x. The key point is that if S_x is sufficiently big relative to $\{0,1\}^m$, then it is possible, otherwise we need much more functions.

The modifications we will make to the protocol is that we will pick some l^2 points in the range at random, for some parameter l. And now, we would like to determine if we can pick l hash functions such that each of the selected points is in the image of S_x under one of the chosen hash functions. From lecture 20, this gives a Σ_2^p predicate.

We can obtain from this an AM protocol to distinguish between the case when $|S_x| \geq 2^m$ and the case when $|S_x| \leq 2^{m-1}$. Arthur picks some points p_i in the range uniformly at random and reveals them to Merlin. Merlin then responds with hash functions h_i and points p_i' in S_x along with certificates of their membership in S_x. Finally, Arthur verifies the membership of points p_i' in S_x and that the image of p_i' under the hash functions indeed covers all the points p_i.

Applying some modifications to the analysis of the original protocol gives us the following results: the AM protocol accepts with probability 1 in the case that $|S_x| \geq 2^m$, and in the other case accepts with probability at most 1/3.

Proof sketch. Define $S_i = \{H | H \equiv G_i\}$ and $S = S_0 \cup S_1$.

The key property that we use is that if $G_0 \equiv G_1$, then $S_0 = S_1$ so $|S| = |S_0| = |S_1|$, and otherwise, $|S| \geq 2\min\{|S_0|, |S_1|\}$.

Since for GNI, we would like to distinguish between 2 sets of size differing by at least a factor of 2, we can adapt the above AM protocol for GNI. However, we need to know the sizes of S_0, S_1, S to use it. By group theory, $|S_i||Aut(G_i)| = n!$ since S_i are the cosets of the subgroup $Aut(G_i)$ in the symmetric group on n points.

Define $T_i = \{(H, \pi) | H \equiv G_i \text{ and } \pi \in Aut(G_i)\}$ and $T = T_0 \cup T_1$. By the above, $|T_i| = n!$. So, if $G_0 \equiv G_1$, $|S| = n!$ and otherwise $|S| = 2n!$.

Since we have a constant factor gap in the sizes, we can use approximate counting to distinguish the 2 sets, such that the larger set corresponds to the non-isomorphic case. We modify it into an AM protocol as above and we are done. □

Note that the above protocol has perfect completeness. So, one question we may ask is whether the conditions of public coins and perfect completeness place any restrictions on the power of interactive proof systems. The answer is negative and in fact:

Theorem 3. *For any interactive proof system with $r(n)$ rounds, there exists an equivalent interactive proof system with:*

1. *Perfect completeness using $r(n)$ rounds.*
2. *Perfect completeness and public coins using $r(n) + 2$ rounds.*

We omit the proof but the idea for (1) is that we can use the hashing technique to eliminate error on the positive side.

Now, we focus on bounded-round public coin systems.

Definition 3. *MA is a public coin system in which Merlin moves first, and then Arthur flips some coins and runs a deterministic predicate on the coins, Merlin's message and the input.*

Formally, a language L is in MA if:

$$x \in L \implies (\exists y \in \Sigma^{n^c}) \Pr_z[V(x, y, z)] \geq 2/3$$

$$x \notin L \implies (\forall y \in \Sigma^{n^c}) \Pr_z[V(x, y, z)] \leq 1/3$$

We can extend AM and MA in a similar way that Σ_2^p extends NP: AMA, MAM, ... This gives a hierarchy of classes based on the number of rounds allowed, and whether Arthur or Merlin goes first.

However, unlike the polynomial hierarchy, this hierarchy collapses. It may seem surprising since Merlin behaves like an existential quantifier, and Arthur behaves like a universal quantifier with restricted power. We now prove the key ingredient for the collapse:

Theorem 4. MA $\subseteq$ AM

One of the main ideas behind the proof is that MA and AM are somewhat similar to a randomized NP machine. For example, in MA, the prover provides a proof and the verifier verifies it probabilistically instead of deterministically like in NP.

Proof. For $L \in$ MA, we have:

$$x \in L \implies (\exists y \in \Sigma^{n^c}) \Pr_{|z|=n^c}[V(x, y, z) = 1] \geq 2/3$$

$$x \notin L \implies (\forall y \in \Sigma^{n^c}) \Pr_{|z|=n^c}[V(x, y, z) = 1] \leq 1/3$$

where V is the underlying deterministic predicate that the verifier evaluates.

So, to get MA $\subseteq$ AM, we would like to swap the choice of the purported proof and the choice of the random string. For the case where $x \in L$, an examination of the condition on the RHS reveals this swap is not a problem. But for the case when $x \notin L$, the swap of quantifiers does not work out. We solve this problem by using amplification and the union bound to our advantage. In particular, for a fixed y, we amplify V such that when $x \notin L$, $Pr_z[V(x, y, z)] \leq \frac{1}{3} \cdot 2^{-n^c}$. Given this, we can now swap the quantifiers, to get

$$x \in L \implies \Pr_{|z|=n^d}[(\exists y \in \Sigma^{n^c})V(x, y, z) = 1] \geq 2/3$$

$$x \notin L \implies \Pr_{|z|=n^d}[(\exists y \in \Sigma^{n^c})V(x, y, z) = 1] \leq \frac{1}{3} \cdot 2^{-n^c} \cdot 2^{n^c} = \frac{1}{3}$$

where n^d is some polynomial of n^c.

This gives us an AM protocol and we are done. □

Equipped with this theorem, intuitively we can switch the order of the first 2 turns in MAM to get AMM and then merge the 2 Merlin turns into 1, and thus MAM = AMM = AM. Similarly, AMA = AAM = AM. We omit the formal proof.

In fact, we can prove something stronger for bounded-round protocols. By the above process, we can halve the number of rounds needed:

Theorem 5. *For any poly-time $r(n)$ with $r(n) \geq 2$,* $\mathrm{AM}(2r(n)) = \mathrm{AM}(r(n))$.

Proof. Omitted. □

Corollary 1. $\mathrm{AM}(k) = \mathrm{AM}$ *for all constant k.*

Thus, this corollary and Theorem 3 give us that private coin systems with a constant number of rounds are equivalent to AM. Note that this does not imply that IP collapses to AM because we would need a super-constant number of applications of the above. Intuitively, this is due to a polynomial factor blow-up in the verification procedure when we apply the above. So, when we apply it only a constant number of times, the verification still runs in polynomial time, but not if we require super-constant number of applications.

3.1 Relationship with other complexity classes

We have the following facts:

1. $\mathrm{MA} \subseteq \Sigma_2^p$
2. $\mathrm{AM} \subseteq \Pi_2^p$
3. $\mathrm{AM} \subseteq \mathrm{NP/\,poly}$
4. Under reasonable derandomization hypotheses, AM = NP.

(1) follows from the fact that Arthur's computation is a BPP computation on Merlin's proof and the input, and we can simulate BPP using approximate counting with a Σ_2^p predicate P on (x, y). Then, since we can simulate Merlin's computation with an existential quantifier, we get $(\exists y \in \Sigma^{n^c})[P(x, y) = 1]$. Merging the existential quantifier with the one in P, we get a Σ_2^p predicate. (2) follows similarly, but we use the fact that $\mathrm{BPP} \subseteq \Pi_2^p$ instead.

(3) follows using a proof similar to the one for $\mathrm{BPP} \subseteq \mathrm{P/\,poly}$. We use amplification to obtain a random string that will work for all the inputs of a fixed length, and then use that string as an advice. We leave the details as an exercise.

The intuition behind (4) is that AM is somewhat similar to NP. The proof is by relativizing the proof that E has $2^{\Omega(n)}$-size circuits implies that P = BPP. (4) also suggests that GI is not NP-complete, since otherwise GNI would be coNP-complete and since $GNI \in \mathrm{AM} = \mathrm{NP}$ under reasonable hypotheses, that would imply that NP = coNP. In fact, we get a collapse of PH if GI is NP-complete even without a derandomization assumption.

Theorem 6. *GI is not NP-complete unless $\Sigma_2^p = \Pi_2^p$.*

Proof. Suppose that GI is NP-complete. Then, GNI is coNP-complete.

Let L be a Σ_2^p language. Then, we have that

$$x \in L \iff (\exists y \in \Sigma^{n^c})(\forall z \in \Sigma^{n^c})[< x, y, z > \in V]$$

for some constant c and a polynomial-time predicate V. Since GNI is coNP-complete and $GNI \in \mathrm{AM}$, we can reduce the $(\forall z \in \Sigma^{n^c})[< x, y, z > \in V]$ coNP predicate as an AM protocol. Since we can express the existential quantifier as a Merlin phase, we have that $\Sigma_2^p \subseteq \mathrm{MAM}$. But because all constant-round interactive proof systems collapse to AM, $\Sigma_2^p \subseteq \mathrm{AM} \subseteq \Pi_2^p$. Thus, $\Sigma_2^p = \Pi_2^p$. □

4 coNP is in IP

One of the main reasons we have for studying the class IP is to see if we can obtain short proofs using interactive proof systems in settings where short standard proofs are unlikely, such as tautologies. While it is unlikely that coNP $\subseteq$ AM, since the above proof would then show that $\Sigma_2^p = \Pi_2^p$, we do have that coNP $\subseteq$ IP. Note that this is one of the few non-trivial results in complexity theory that does not relativize.

Theorem 7. *The language* $L = \{(\varphi, \#SAT(\varphi)) | \varphi \in \{0,1\}^*\}$ *is in* IP*, where* $\#SAT(\varphi)$ *denotes the number of assignments satisfying a SAT clause* φ*.*

Note that L is the decision variant of the counting problem for SAT.

Proof(first half). For L, we are given (φ, k) and we would like to verify using an interactive proof system that $\#SAT(\varphi) = k$. One of the key ingredients is arithmetization, which we looked at briefly in the last lecture.

Assume that φ has n variables and m clauses, and is 3-CNF. So, we would like to transform $\varphi(x_1, \ldots, x_n)$ into a polynomial over the integers, $\tilde{\varphi}(x_1, \ldots, x_n)$ such that $\tilde{\varphi}$ agrees with φ on $\{0,1\}^n$ and $\tilde{\varphi}$ has degree at most m in every variable and easy to evaluate as well.

Now we show how to transform a clause C. For example, if $C = (x_1 \vee \bar{x_2} \vee x_3)$, then after transformation we have $\tilde{C} = 1 - (1 - x_1)x_2(1 - x_3)$. Then, to construct $\tilde{\varphi}$, we use $\tilde{\varphi} = \prod_{j=1}^{m} \tilde{C_j}$. Since each variable occurs at most once in each clause, the construction gives a polynomial with degree at most m in every variable, and also $\tilde{C_j} = 1$ if C_j is satisfied and $\varphi = 1$ if and only if every $\tilde{C_j} = 1$. Thus, $\tilde{\varphi}$ agrees with φ on $\{0,1\}^n$. Also, $\tilde{\varphi}$ is easy to evaluate since we just have to go through all the transformed clauses $\tilde{C_j}$ and make sure they all evaluate to 1.

Then, to count the number of satisfying assignments, we use $\sum_{x_1=0}^{1} \cdots \sum_{x_n=0}^{1} \tilde{\varphi}(x_1, \ldots, x_n)$. □

5 Next Lecture

For next time, we will finish the above proof. Then, we will present a *sumcheck protocol* with perfect completeness to verify that degree d polynomials have d zeros and then show that IP = PSPACE. We will also introduce PCP, the class of languages with *probabilistically checkable proofs* and the power of being able to "interrogate" multiple provers with different "questions", with each prover able to access only its own histories. This is akin to police interrogation in which there are multiple suspects, each of which is isolated from the others, and the police can ask different questions to each.

CS 810: Complexity Theory 3/28/2007

Lecture 24: Probabilistically Checkable Proofs

Instructor: Dieter van Melkebeek Scribe: Tom Watson

In this lecture we continue our discussion of interactive proof systems. We develop a nonrelativizing proof technique, encapsulated in the so-called *sumcheck protocol*, and use it to prove that $\mathrm{P}^{\#\mathrm{P}} \subseteq \mathrm{IP}$. Building on this technique, we strengthen the result and show that IP = PSPACE, i.e., polynomial space exactly captures the power of interactive proof systems. Then we introduce the notion of *multiple prover interactive proof systems* and relate them to *probabilistically checkable proofs*, in which the interaction between prover and verifier is eliminated but the proof is allowed to be very long. There are some extremely profound and involved results in these areas, which we briefly discuss.

1 Interactive Proof Systems for #P

Two lectures ago we discussed proof complexity, in which the central question is whether coNP statements have short proofs. In the last lecture we generalized our notion of proof to include both randomness in the verification and interaction with a prover. This led to the notion of an *interactive proof system*, in which there are two parties interacting to decide whether a given input is in a particular language. There is a *prover* who is computationally unrestricted, and therefore knows whether the input is in the language, but it is the (skeptical) probabilistic *verifier* who gets the final say. We can think of the prover as trying to get the verifier to accept, and we want it to be the case that if the input is in the language, then some prover can, in fact, convince the verifier to accept with probability at least some value c (informally, we can prove everything that's true, the *completeness* property) and that if the input is not in the language, then *no* prover, not even a devious one, can convince the verifier to accept with probability greater than some value s (informally, we cannot prove anything that's false, the *soundness* property).

We gave examples of interactive proof systems for the Graph Nonisomorphism problem, which is a problem for which we don't know of short, efficiently verifiable classical proofs (although under certain reasonable complexity theoretic assumptions, they are known to exist). We now demonstrate that not only do all coNP statements have short proofs in the present sense, but (presumably) many more languages have short proofs. We establish the exact power of interactive proof systems by showing that IP = PSPACE. We point out that the proof is nonrelativizing. As a step toward proving this result, we now develop an interactive proof for the problem of deciding whether the number of satisfying assignments to a given CNF formula is exactly some number k. This allows us to establish that $\mathrm{P}^{\#\mathrm{P}} \subseteq \mathrm{IP}$.

Our interactive proof system involves two main ingredients. The first is arithmetization of the given formula ϕ, as discussed in the last lecture. This is a simple procedure that produces a multivariate polynomial $\widetilde{\phi}(x_1, \ldots, x_n)$ that agrees with ϕ on the Boolean cube. This is the step that prevents our proof from relativizing; we leave it as an exercise to argue formally why this is the case. This polynomial has two very important properties: every variable has degree at most m in it, where m is the number of clauses in ϕ, and it can be efficiently evaluated (despite possibly having exponentially many monomials when expanded). These are the only two properties we need in order to apply the second main ingredient, the sumcheck protocol, which we describe next.

1.1 Sumcheck Protocol

Our goal is to design a verifier to decide whether the number of satisfying assignments to ϕ is equal to some number k_0. This is equivalent to checking the following identity:

$$\sum_{x_1=0}^{1}\sum_{x_2=0}^{1}\cdots\sum_{x_n=0}^{1}\widetilde{\phi}(x_1,x_2,\ldots,x_n)=k_0. \tag{1}$$

We don't have time to evaluate $\widetilde{\phi}$ for all 2^n settings of $x_1,\ldots,x_n$, so we enlist the help of the prover. The sumcheck protocol is a method for doing this that is secure against cheating provers.

The key idea is to consider the univariate polynomial $g(x)$ defined by

$$g(x)=\sum_{x_2=0}^{1}\cdots\sum_{x_n=0}^{1}\widetilde{\phi}(x,x_2,\ldots,x_n).$$

This is the same as the expression in Equation (1) except that we don't sum over x_1. Since x_1 has degree at most m in $\widetilde{\phi}$, $g(x)$ has degree at most m. If we knew $g(x)$, then we would be able to check Equation (1) by checking $g(0)+g(1)=k_0$, using the observation that

$$g(0)+g(1)=\sum_{x_1=0}^{1}\sum_{x_2=0}^{1}\cdots\sum_{x_n=0}^{1}\widetilde{\phi}(x_1,x_2,\ldots,x_n).$$

We don't have $g(x)$, but we can ask the prover to send it to us. We know that $g(x)$ has at most $m+1$ coefficients, which is manageable, but it's conceivable that these coefficients are prohibitively huge numbers. This turns out not to be an issue, but we postpone the discussion of this.

Now the prover sends us some polynomial $g'(x)$, which may not be equal to $g(x)$ if the prover is cheating. We check that $g'(x)$ has degree at most m and that $g'(0)+g'(1)=k_0$ and reject if either does not hold. Now if Equation (1) is true, then the prover can just send us the correct polynomial $g'(x)=g(x)$ and everything will be fine. If Equation (1) is not true, then if the prover wants to have a prayer of getting us to accept, then it's forced to send an incorrect polynomial $g'(x)\neq g(x)$. Since the two polynomials agree in at most m places, we can pick a random number ξ_1 from a reasonably small set I and with high probability, $g'(\xi_1)\neq g(\xi_1)$. If we had $g(x)$ then we could just evaluate $g(\xi_1)$ and $g'(\xi_1)$ and with high probability catch the cheating prover. Of course, the whole point is that we don't have $g(x)$, but we have now reduced our original problem, verifying Equation (1), to a simpler instance of the same problem, namely verifying

$$\sum_{x_2=0}^{1}\cdots\sum_{x_n=0}^{1}\widetilde{\phi}(\xi_1,x_2,\ldots,x_n)=k_1 \tag{2}$$

where $k_1=g'(\xi_1)$. By sending us $g'(x)$, the prover has implicitly claimed that Equation (2) holds. It had no way of knowing what number ξ_1 we would pick, though, which gives us an edge. If Equation (1) holds, then Equation (2) holds when the prover just sends us the correct polynomial. If Equation (1) does not hold, then with high probability Equation (2) does not hold, regardless of the prover's behavior. In either case we are back to where we started, except that now there are only $n-1$ variables being summed over. We iterate this process until our task is reduced to the problem of verifying

$$\widetilde{\phi}(\xi_1,\xi_2,\ldots,\xi_n)=k_n \tag{3}$$

for some numbers $\xi_1, \ldots, \xi_n \in I$ and k_n. Now we use the other property required of $\widetilde{\phi}$, namely that is can be evaluated efficiently, to explicitly check that Equation (3) holds. We accept if it holds and reject otherwise. This is the only time in the protocol when we are willing to accept.

Theorem 1. *The sumcheck protocol is a public-coin interactive proof system that verifies Equation (1) with perfect completeness and polynomially small soundness, provided $\widetilde{\phi}$ has degree at most m in every variable and can be evaluated in polynomial time.*

Proof. That the protocol is public-coin is clear — in each step the verifier uses its randomness only to pick the number ξ_i, and then it sends ξ_i to the prover. That the protocol has perfect completeness is also straightforward to see — if Equation (1) holds then the prover can just send us the correct polynomial $g'(x) = g(x)$ in every phase and the consistency check $g'(0) + g'(1) = k_i$ will always pass and all the implicit claims of the form of Equation (2) will be true, so the final check of Equation (3) will also pass, leading to acceptance with probability 1.

Now we argue the soundness. Suppose Equation (1) does not hold, but nevertheless we accept. In the last phase, the claim we're checking, Equation (3), is true (otherwise we would reject), but in the first phase, the claim we're checking, Equation (1), is false. Thus there must be some phase where the current claim transitions from being false to being true. In this phase, the prover sends us some $g'(x) \neq g(x)$ (since otherwise we would reject after seeing that the consistency check $g'(0) + g'(1) = k_i$ fails). For the claim generated by this phase to be true, it must be the case that $g'(\xi_{i+1}) = g(\xi_{i+1})$, which happens with probability at most $m/|I|$. The probability that we accept is at most the probability that this happens in some phase, which by a union bound is at most $mn/|I|$. We can choose $|I|$ to be polynomially large to make the soundness polynomially small.

The only thing left to verify is that the protocol runs in polynomial time. The final check can be done in polynomial time by hypothesis, so the only issue is that the coefficients of the $g(x)$ polynomials might become too large. One can argue that the coefficients do not become too large. We can avoid that analysis by letting I be the integers mod p for a small prime p, and letting all computations take place in the field of integers mod p. Now we need to worry about the soundness, though; it could be the case that the originally claimed value k_0 is congruent to the true value mod p, and in that case the prover can make the verifier always accept. However, the difference between k_0 and the true value cannot have too many prime factors (certainly fewer than n). The density of primes is great enough that picking p at random from the set of primes of polynomial magnitude (and hence logarithmic bit length) affects the soundness by only a polynomially small amount. Thus the overall soundness is still polynomially small, and computations mod p can certainly be done efficiently. Selecting p at random is not a problem since the verifier can enumerate all primes in the desired range by brute force. □

Corollary 1. *Every language in $\mathrm{P}^{\#\mathrm{P}}$ has a public-coin interactive proof system with perfect completeness, and in particular $\mathrm{P}^{\#\mathrm{P}} \subseteq \mathrm{IP}$.*

Proof. We can simulate a $\mathrm{P}^{\#\mathrm{P}}$ machine, and whenever it makes an oracle query, reduce the query to a #SAT instance, have our prover tell us how many satisfying assignments it has, and run the sumcheck protocol to verify this. The resulting protocol still runs in polynomial time. If the input is in the language, then the prover can follow the protocol, leading to acceptance with probability 1. If the input is not in the language, then for us to accept, the prover must cheat on at least one query, and this slips by us with exponentially small probability per query. By a union bound, the soundness of this protocol is still exponentially small. □

2 Interactive Proof Systems for PSPACE

Using the sumcheck protocol, we can now characterize the power of interactive proof systems.

Theorem 2. *Every language in* PSPACE *has a public-coin interactive proof system with perfect completeness, and in particular* PSPACE $\subseteq$ IP*.*

Corollary 2. IP = PSPACE*.*

Proof. The inclusion PSPACE $\subseteq$ IP follows from Theorem 2. The inclusion IP $\subseteq$ PSPACE is left as an exercise; it just involves a space-efficient simulation of an interactive proof system. □

In the last lecture we mentioned that every interactive proof system can be converted to one with perfect completeness without increasing the number of rounds, and also to one with both perfect completeness and public coins with only an increase of two in the number of rounds. In particular, requiring perfect completeness and public coins does not decrease the power of IP. Theorem 2 and Corollary 2 provide an alternative proof of the latter fact: every language in IP is in PSPACE by Corollary 2, and every language in PSPACE has a public-coin interactive proof system with perfect completeness by Theorem 2.

We now prove the main result of this section.

Proof of Theorem 2. One idea is to show that $TQBF \in$ IP by arithmetizing quantified formulas and using a protocol similar to the sumcheck protocol. Arithmetizing a universal quantifier, however, results in a squaring of the degree, which could lead to very high degree polynomials. This can be remedied by using an extra step that reduces the degree of the polynomial after arithmetizing a universal quantifier. This approach does lead to a proof that PSPACE $\subseteq$ IP, but we go a different route, employing the divide-and-conquer strategy used in our proof that NSPACE(s) $\subseteq$ DSPACE(s^2).

In the sumcheck protocol, our goal was to count the number of satisfying assignments to a CNF formula, or equivalently the number of accepting computation paths of a nondeterministic machine. Now we are given a deterministic machine running in space s and wish to determine the number of accepting computation paths, which is either 0 or 1. The computation paths, however, could be exponentially long, so we can't work directly with a formula expressing the whole computation. We instead work with polynomials derived from the divide-and-conquer approach of verifying a computation tableau. This involves another application of the main idea of the sumcheck protocol — in a divide-and-conquer step, we do not just randomly pick a half of the tableau to recurse on, because this only gives us a probability 1/2 of catching a cheating prover. Instead we capture this with a polynomial where plugging in 0 corresponds to checking one half of the tableau and plugging in 1 corresponds to checking the other half, and we plug in a random value from a much larger domain. In this way we sort of "handle both halves at once", and catch a cheating prover with high probability.

We now pursue this intuition formally. We assume for simplicity of notation that the configurations of the machine M we wish to simulate are bit strings of length s, and that the machine makes exactly 2^s transitions, where 2^s is a power of 2 (thus the computation tableau has $2^s + 1$ rows). This last assumption can be justified by assuming that once M reaches a halting configuration, it just keeps transitioning to that same configuration forever. We also assume there is a unique accepting

configuration. For $\ell = 0, \ldots, s$ we define the following predicate on pairs of configurations:

$$P_\ell(C_0, C_1) = \begin{cases} 1 & \text{if } C_0 \vdash_M^{2^\ell} C_1 \\ 0 & \text{otherwise} \end{cases}.$$

That is, $P_\ell(C_0, C_1)$ indicates whether executing 2^ℓ transitions of M starting from configuration C_0 leads to configuration C_1. These predicates satisfy the following recurrence: P_0 is just specified by the transition function of M, and for $\ell > 0$,

$$P_\ell(C_0, C_1) = \sum_{C_{1/2} \in \{0,1\}^s} P_{\ell-1}(C_0, C_{1/2}) P_{\ell-1}(C_{1/2}, C_1). \tag{4}$$

Thus, the claim that we need to verify is

$$P_s(C_0, C_1) = k_s, \tag{5}$$

where C_0 is the unique initial configuration determined by the input, C_1 is the unique accepting configuration, and $k_s = 1$.

Our first step is to arithmetize these predicates to obtain multivariate polynomials (in fact, polynomials in exactly $2s$ variables) $\widetilde{P}_\ell(C_0, C_1)$ such that $\widetilde{P}_\ell$ agrees with P_ℓ on the Boolean cube. We have the following fact.

Claim 1. *Under an appropriate encoding of configurations, there exists a $2s$-variate polynomial $\widetilde{P}_0(C_0, C_1)$ that agrees with $P_0(C_0, C_1)$ on the Boolean cube, can be evaluated in polynomial time, and such that the degree of every variable is $O(1)$.*

We leave the proof of Claim 1 as an exercise. It just involves summing over all possible tape head positions and transitions, and expressing that the contents of the configurations are consistent with that transition by multiplying together many simple checks.

The polynomial $\widetilde{P}_0$ and Equation (4) immediately specify all the polynomials $\widetilde{P}_\ell$. Moreover, in each $\widetilde{P}_\ell$, every variable has degree $O(1)$. This can be argued by induction, using the fact that in Equation (4), the only variables whose degrees increase are $C_{1/2}$, but these are summed out. The constant degree property is not critical, though; it turns out that our argument would go through if the degrees of the variables were larger.

The claim we need to verify is now

$$\widetilde{P}_s(C_0, C_1) = k_s. \tag{6}$$

Now looking at Equation (4), we see that if $\widetilde{P}_{s-1}$ were easy to evaluate, then we would be in exactly a position to use the sumcheck protocol to verify our claim and be done. We don't know how to evaluate $\widetilde{P}_{s-1}$ efficiently, but we observe that actually, the sumcheck protocol reduces the problem of verifying

$$\sum_{C_{1/2} \in \{0,1\}^s} \widetilde{P}_{s-1}(C_0, C_{1/2}) \widetilde{P}_{s-1}(C_{1/2}, C_1) = k_s$$

to the problem of verifying

$$\widetilde{P}_{s-1}(C_0, \gamma) \widetilde{P}_{s-1}(\gamma, C_1) = \kappa_s \tag{7}$$

for some particular values γ and κ_s.

It looks as if we've almost reduced our original problem, checking Equation (6), to a simpler instance of the same problem. However, Equation (7) involves the product of two evaluations of $\widetilde{P}_{s-1}$, so it is not quite the same as our original problem. The two factors correspond to the two halves of the computation tableau, and if we were to recurse on both halves, then we would be no better off than just simulating the entire computation from start to finish. The following is the key idea that leads to an exponential savings in running time.

Define $h(x)$ to be a univariate polynomial such that $h(0) = (C_0, \gamma)$ and $h(1) = (\gamma, C_1)$. More precisely, h is a $2s$-tuple of univariate polynomials, one for each bit of the output. Each bit is a simple linear function of x (since C_0, γ, and C_1 are fixed). Now we're interested in checking

$$\widetilde{P}_{s-1}(h(0))\widetilde{P}_{s-1}(h(1)) = \kappa_s,$$

which seems similar to the consistency check $g'(0) + g'(1) = k_0$ in the sumcheck protocol. Here we set $g(x) = \widetilde{P}_{s-1}(h(x))$. Since each of the $2s$ components of h has degree at most 1, and each of the $2s$ variables in $\widetilde{P}_{s-1}$ has degree $O(1)$, $g(x)$ is a univariate polynomial of degree $O(s)$. If we had $g(x)$ then we could just check that $g(0)g(1) = \kappa_s$ and be done. Like in the sumcheck protocol, we ask the prover for the coefficients of $g(x)$ and it gives us some $g'(x)$ of degree $O(s)$. We check that $g'(0)g'(1) = \kappa_s$ and reject if this does not hold. In the event that $g(0)g(1) \neq \kappa_s$, this forces the prover to give us an incorrect polynomial $g'(x) \neq g(x)$. In order to catch a cheating prover, we pick a random $\xi \in I$, and now with high probability, $g'(\xi) \neq g(\xi)$. We can't detect this immediately since we can't evaluate $g(\xi)$, but we have reduced our original problem to a simpler instance of the same problem. By sending $g'(x)$, the prover has implicitly claimed that

$$\widetilde{P}_{s-1}(h(\xi)) = k_{s-1}, \tag{8}$$

where $k_{s-1} = g'(\xi)$. Now k_s has been replaced by k_{s-1}, (C_0, C_1) has been replaced by $h(\xi)$, and ℓ has been reduced from s to $s-1$. We can iterate this process until we've reduced the problem to checking

$$\widetilde{P}_0(\alpha) = k_0, \tag{9}$$

where α is some $2s$-tuple of numbers (not necessarily bits) and k_0 is some number. By Claim 1, this can be done efficiently. We accept if Equation (9) holds and reject otherwise. This is the only time in the protocol when we are willing to accept.

As in the case of the sumcheck protocol, we need to worry about whether the coefficients become too large. In this case, they can become very large, which necessitates performing all computations modulo some small prime. Since the number of accepting computation paths is either 0 or 1, the soundness is unaffected, regardless of which prime we use.

We now recap the protocol and argue its correctness. The main outline consists of s iterations, each of which reduces the problem of verifying the value of $\widetilde{P}_\ell$ on some input to the problem of verifying the value of $\widetilde{P}_{\ell-1}$ on some other input, and then a final verification of the value of $\widetilde{P}_0$ on some input, which can be done efficiently. Each iteration consists of running the sumcheck protocol with the identity (4) to reduce the problem of verifying the value of $\widetilde{P}_\ell$ on some input to the problem of verifying the value of the product of $\widetilde{P}_{\ell-1}$ on two other inputs. This is then reduced to the problem of verifying the value of $\widetilde{P}_{\ell-1}$ on a single input by interpolating through the two inputs with a univariate polynomial $h(x)$ and setting the single input to be $h(\xi)$ for a randomly chosen ξ.

Like the sumcheck protocol, this protocol is public-coin and has perfect completeness. Now if the original claim, Equation (6), is false, then in order for us to accept, the current claim must

transition from being false to being true in some iteration. By a union bound, the soundness is at most s times the probability that this happens in any given iteration. Within an iteration, there are two steps: we obtain an intermediate claim via the sumcheck protocol, and then we obtain the claim for the next iteration using the interpolating polynomial $h(x)$ evaluated at a random ξ. The probability that the intermediate claim becomes true is at most $O(s/|I|)$ by Theorem 1 (since there are $O(s)$ variables and each has constant degree). The probability that the claim transitions from false to true in the second step is at most $O(s/|I|)$ since the $g(x)$ has degree $O(s)$. By a union bound, the probability that the claim transitions from false to true in a given iteration is also $O(s/|I|)$, and so the soundness of the protocol is $O(s^2/|I|)$. Since s is only polynomial, $|I|$ doesn't have to be too large to make the soundness small, and thus we don't have to pick the prime p to be very large. We can find an appropriate p by brute force. □

3 Probabilistically Checkable Proofs

3.1 Multiple Prover Interactive Proof Systems

We bridge the gap between interactive proof systems and probabilistically checkable proofs by introducing *multiple prover interactive proof systems*, in which there are multiple provers that the verifier may query throughout the protocol, and they are allowed to cooperatively decide on any strategy before the protocol begins, *but they may not communicate during the protocol*. A multiple prover interactive proof system is said to decide a language L with completeness c and soundness s if for all $x \in L$ there exist provers that make the verifier accept with probability at least c, and for all $x \notin L$, no provers can make the verifier accept with probability more than s. Define the class MIP to be the set of languages decided by polynomial-time multiple prover interactive proof systems with any number of provers (though the verifier can clearly only query polynomially many) and say completeness 2/3 and soundness 1/3.

The additional power afforded by having multiple provers is similar to a strategy used by the police to question suspected accomplices: interrogating the suspects separately and checking the consistency of their stories improves the chances of being able to catch them lying. Just as we quantified the exact power of interactive proof systems in Corollary 2, one can quantify the exact power of multiple prover interactive proof systems as follows.

Theorem 3. MIP = NEXP.

We omit the proof of Theorem 3, which uses similar ingredients as our proof of Theorem 2 but is more involved. We note, however, that this theorem and the fact that NP $\subsetneq$ NEXP (which follows from the nondeterministic time hierarchy) imply that we can decide *provably* more languages with multiple prover interactive proof systems than with classical proofs (i.e., NP).

Since we believe that PSPACE $\subsetneq$ NEXP, it seems that multiple prover interactive proof systems are strictly more powerful than single prover interactive proof systems. Does each additional prover yield more power? The following theorem, which we prove shortly, shows that the answer is no. Define MIP[k] to be the set of languages decided by multiple prover interactive proof systems with k provers.

Theorem 4. MIP = MIP[2].

We are now ready to study the very influential notion of probabilistically checkable proofs, in which the proof is entirely written down and the element of interaction is removed, but the proofs are in principle allowed to be very long.

3.2 Probabilistically Checkable Proofs

Definition 1. *A* probabilistically checkable proof system *with completeness c and soundness $s < c$ for a language L is a polynomial-time probabilistic oracle Turing machine V such that if $x \in L$ then there exists a proof* Π *such that*

$$\Pr[V^{\Pi}(x) \text{ accepts}] \geq c,$$

and if $x \notin L$ then for all proofs Π*,*

$$\Pr[V^{\Pi}(x) \text{ accepts}] \leq s.$$

Typical settings of the completeness and soundness parameters are $c = 1$, $s = 1/2$.

The key difference between a probabilistically checkable proof system and an interactive proof system is that in the former, the prover is a fixed oracle — its responses to all possible queries must be fixed for a given input and are not allowed to change after the verifier starts computing. With probabilistically checkable proofs, we are usually interested in the amount of randomness and number of queries needed by the verifier. This motivates the following definition.

Definition 2. *The class* $\mathrm{PCP}(r(n), q(n))$ *is the set of languages decidable by a probabilistically checkable proof system where the verifier uses at most $r(n)$ random bits and queries at most $q(n)$ bits of the proof.*

Theorem 5. $\mathrm{PCP}(r(n), q(n)) \subseteq \mathrm{NTIME}(2^{O(r(n)+q(n))} n^{O(1)})$.

Proof. Consider a probabilistically checkable proof system where the verifier uses $r(n)$ random bits and makes $q(n)$ queries. For each random bit sequence, the number of locations of the proof that could ever get queried is $2^{q(n)}$ (due to the possibly adaptive nature of the verifier). Thus over all proofs and all random strings, there are at most $2^{r(n)+q(n)}$ locations that ever get queried (for a given input). A nondeterministic machine can guess the addresses and answers to these queries in time $2^{r(n)+q(n)} n^{O(1)}$. It can then run over all random bit sequences, simulate the verifier for each one (looking up the answers to queries in the guessed table), and explicitly compute the probability of acceptance of the verifier for the guessed proof. This takes time $2^{r(n)} n^{O(1)}$. If the probability of acceptance is at least the completeness of the probabilistically checkable proof system then the machine accepts, and otherwise it rejects. □

We now draw the connection between multiple prover interactive proof systems and probabilistically checkable proofs. Note that the following result immediately implies Theorem 4 and shows that $\mathrm{MIP} = \mathrm{PCP}(poly(n), poly(n))$.

Theorem 6. $\mathrm{MIP} \subseteq \mathrm{PCP}(poly(n), poly(n)) \subseteq \mathrm{MIP}[2]$.

Proof. We first argue the inclusion $\mathrm{MIP} \subseteq \mathrm{PCP}(poly(n), poly(n))$. For a given multiple prover interactive proof system, we construct a probabilistically checkable proof system where the proof is interpreted as a complete specification of the provers' strategies, i.e., a list of their responses for

every possible message history. The verifier can just simulate the multiple prover interactive proof system, looking up the provers' responses by querying the provided proof. The completeness and soundness are the same as for the multiple prover interactive proof system.

For the inclusion $\mathrm{PCP}(poly(n), poly(n)) \subseteq \mathrm{MIP}[2]$, we can have the verifier just simulate the given probabilistically checkable proof system, using one of its provers to answer queries to the proof. The only problem is that this prover is allowed to be adaptive, whereas in the probabilistically checkable proof system, the entire proof is written down from the beginning. We use a second prover to try to catch the first prover if it behaves nonadaptively. However, the second prover is also allowed to be adaptive, so we're only safe asking it one question. Specifically, when the simulation is over, if the verifier decides to accept then before doing so it picks one of the queries it made to the first prover uniformly at random, makes the same query to the second prover, and only proceeds to accept if their answers agree. Since the provers cannot communicate during the protocol, the second prover's possible responses form a nonadaptive strategy.

Now if the input is in the language, then the two provers can just respond according to the proof specified by the probabilistically checkable proof system, leading to acceptance to probability at least the completeness of that system. If the input is not in the language, then we can compute the probability of acceptance as follows, where q is the number of queries made and A denotes the event that for at least one query asked of the first prover, its response differs from the response we would get if we asked the second prover.

$$\begin{aligned}\Pr\left(\text{accept}\right) &= \Pr\left(\text{accept} \;\middle|\; A\right) \cdot \Pr\left(A\right) + \Pr\left(\text{accept} \;\middle|\; \overline{A}\right) \cdot \Pr\left(\overline{A}\right) \\ &\leq \Pr\left(\text{accept} \;\middle|\; A\right) + \Pr\left(\text{accept} \;\middle|\; \overline{A}\right) \\ &\leq \left(1 - \frac{1}{q}\right) + s.\end{aligned}$$

In the last inequality, $\Pr\left(\text{accept} \;\middle|\; A\right) \leq 1 - 1/q$ follows from the fact that we catch the provers' inconsistency with probability at least $1/q$, and $\Pr\left(\text{accept} \;\middle|\; \overline{A}\right) \leq s$ follows from the soundness property applied to the proof defined by the second prover's responses.

Now the original probabilistically checkable proof system can be amplified so that s is exponentially small and c is exponentially close to 1. Then since q is only polynomially large, the soundness $1 - 1/q + s$ of this multiple prover interactive proof system is non-negligibly less than its completeness c. We can repeat the whole protocol a few times and see whether the average number of accepted runs is less than or at least the midpoint between $1 - 1/q + s$ and c. Intuitively, there shouldn't be issues with adaptivity when we rerun the protocol from scratch since the queries made in previous runs give the provers no information about what will happen in future rounds. More formally, each possible execution of the first $k - 1$ runs defines a proof for the original probabilistically checkable proof system, namely, the proof specified by the responses of the second prover in the kth run given the execution of the first $k - 1$ runs. The soundness and completeness properties apply to each of these exponentially many proofs corresponding to the different executions of the first $k - 1$ runs. Now suppose the input is not in the language. Then the probability of acceptance on the kth run given any particular execution of the first $k - 1$ runs is at most $1 - 1/q + s$ and hence the random variable indicating acceptance on the kth run is at most some indicator random variable that is 1 with probability exactly $1 - 1/q + s$ conditioned on each execution of the first $k - 1$ runs. It follows that the indicator random variables thus defined for each of the runs are fully independent and all have expectation $1 - 1/q + s$, so by a Chernoff bound the soundness is driven down exponentially. Similarly, the completeness is improved. □

3.3 The PCP Theorem

By the nondeterministic time hierarchy theorem, we know that there exist languages in NEXP that do not have short, efficiently verifiable proofs. However, Theorems 3 and 6 imply that all languages in NEXP have efficiently verifiable proofs, where we allow the verifier to randomly spot check a few locations of the proof. Can we scale this result down to get a similar result for NP? That is, can we replace classical proofs with ones where it is only necessary to spot check a few random locations? This is the content of the famous PCP Theorem, which is possibly the most celebrated and involved result in all of theory of computing.

Theorem 7 (The PCP Theorem). $\text{NP} = \text{PCP}(O(\log n), O(1))$.

The inclusion $\text{PCP}(O(\log n), O(1)) \subseteq \text{NP}$ follows immediately from Theorem 5. The other direction is much harder. We do not prove this result in this course, but we illustrate the flavor of the proof by proving the following weaker result, which uses some of the same techniques needed for the PCP Theorem.

Theorem 8. $\text{NP} \subseteq \text{PCP}(poly(n), O(1))$.

In the probabilistically checkable proof system we design for Theorem 8, the proof is an exponentially long encoding of a classical proof. Nevertheless, this result captures what is perhaps the most surprising aspect of the PCP Theorem, namely that only a constant number of bits of the proof need to be queried to verify its correctness.

One of the major applications of the PCP Theorem is hardness of approximation results. Many important combinatorial optimization problems are NP-hard to solve exactly, and for many of these problems the PCP Theorem implies that it is NP-hard even to approximate them within certain factors. In fact, for some problems we have tight results, i.e., approximation algorithms and hardness results with essentially matching approximation factors. Thus the PCP Theorem has allowed us to characterize the approximability of these problems. The basic reason the PCP Theorem leads to hardness of approximation results is that the prover is trying to solve an optimization problem — that of maximizing the probability that the verifier accepts — and this theorem introduces a non-negligible gap in this probability between the cases where the input is in the language and not in the language.

In the next lecture, we will see concrete examples of how the PCP Theorem leads to hardness of approximation results. We will also prove Theorem 8. The proof involves encoding classical proofs with the Hadamard code and using harmonic analysis to analyze the properties of the resulting probabilistically checkable proof system.

4 The Power of the Honest Prover

In this section we address the following question and its implications. For a given interactive or probabilistically checkable proof system, if the input is in the language, then how powerful does a prover have to be to convince the verifier of this fact? We would like it to be the case that the honest prover isn't required to perform computational tasks that are more difficult than solving the original problem from scratch. That is, ideally, the honest prover can answer queries in polynomial time with access to an oracle for the language being decided. We have seen several examples where this is the case.

- This is the case for our private-coin interactive proof system for Graph Nonisomorphism: in the case that the two input graphs are non-isomorphic, all the prover has to do is determine which of the two is isomorphic to the graph provided by the verifier, which can be done with two queries to an oracle for Graph Nonisomorphism.

- This is also the case for our sumcheck protocol for #P functions. The prover is only ever required to find the coefficients of a univariate polynomial obtained by summing an easily evaluatable multivariate polynomial over all 0-1 settings of all but one variable. The prover can plug in a particular value for the free variable and use a #P oracle to find the value of the desired polynomial on that input. (The basic reason is the closure of #P under uniform exponential summations. The #P oracle just needs to nondeterministically guess the settings of the other variables, evaluate the multivariate polynomial, and generate a number of accepting branches equal to the outcome.) Since the desired polynomial is of low degree, the prover can try a few different inputs and reconstruct the polynomial from its values on those inputs.

- Finally, the prover for our PSPACE protocol can compute its responses in polynomial space. After all, the proof of the inclusion IP $\subseteq$ PSPACE involves simulating the interactive proof system and determining the strategy that causes the verifier to accept with the highest probability. The prover can follow this strategy.

We have also seen some examples for which it is open whether an honest prover requires more computational resources than are required to solve the problem at hand.

- This is the case for our public-coin interactive proof system for Graph Nonisomorphism.

- Theorems 1 and 2 immediately yield interactive proof systems for coNP. It is open whether an honest prover for either of these protocols can compute its responses with a coNP oracle. The most efficient honest prover we know of is the one for the sumcheck protocol, which can compute its responses with an oracle for #P.

Why do we care about the power of the honest prover? We give two applications below.

4.1 Relations Among Complexity Classes

Recall the following theorems.

Theorem 9. *If* NP $\subseteq$ P/poly *then* PH $= \Sigma_2^p$.

Theorem 10. *If* PSPACE $\subseteq$ P/poly *then* PSPACE $= \Sigma_2^p$.

Theorem 11. *If* EXP $\subseteq$ P/poly *then* EXP $= \Sigma_2^p$.

We proved Theorem 9 in an earlier lecture and left Theorems 10 and 11 as exercises. We can now prove the following stronger version of Theorem 10.

Theorem 12. *If* PSPACE $\subseteq$ P/poly *then* PSPACE $=$ MA.

Proof. Recall that in the last lecture we proved that MA $\subseteq \Sigma_2^p$, so this is indeed a stronger result. Let L be a language in PSPACE. We know that there is an interactive proof system for L in which the honest prover's responses are computable in polynomial space. Assuming PSPACE $\subseteq$ P/ poly, there is a polynomial-size circuit implementing the honest prover's strategy (i.e., given a transcript of a message history, it computes each bit of the honest prover's next message). We have Merlin send this circuit to Arthur, who can then carry out the protocol by himself, evaluating the circuit to determine the prover's responses. If the input is in the language then, as we argued above, there is a circuit that makes Arthur accept with probability 1. If the input is not in the language then no prover, and in particular no polynomial-size circuit, can make Arthur accept with probability greater than the soundness of the interactive proof system for L. This proves that $L \in$ MA. □

Since the power of interactive proof systems only extends as far as PSPACE, the above argument does not immediately yield an analogous strengthening of Theorem 11. However, the power of multiple prover interactive proof systems allows us to prove the desired result in much the same way.

Theorem 13. *If* EXP $\subseteq$ P/ poly *then* EXP = MA.

Proof of Theorem 13. Let L be a language in EXP. Theorem 3 implies $L \in$ MIP, and it turns out that the strategy of the honest provers for this protocol can be implemented in exponential time. Assuming EXP $\subseteq$ P/ poly, there exist polynomial-size circuits implementing the strategy of each of the honest provers in this multiple prover interactive proof system. Merlin sends these circuits to Arthur, who then carries out the protocol by himself, using the circuits to determine the provers' responses. This establishes that $L \in$ MA. □

Natural questions that arise are whether analogous results hold for NEXP and NP. We cannot use this technique to establish an analogous result for NEXP since it is open whether the honest provers for the NEXP protocol given by Theorem 3 can compute their responses in polynomial time given oracle access to NEXP. The question of whether an analogous result holds for NP is also open.

Conjecture 1. *If* NP $\subseteq$ P/ poly *then* PH = MA.

Given a language in coNP, if we knew an honest prover for the interactive proof system given by Theorem 1 or Theorem 2 could compute its responses in polynomial time with an oracle for NP, then an almost identical argument to the proof of Theorem 12, together with the fact that coNP $\subseteq$ MA implies PH = MA, would establish Conjecture 1. In fact, it would suffice for the honest prover to use an oracle for any language in PH, since we know that NP $\subseteq$ P/ poly implies PH $\subseteq$ P/ poly. However, the best honest provers we know of for languages in coNP require the power of counting.

4.2 Program Verification

As a second application of the power of the honest prover, we study the problem of verifying correctness of programs. The program verification problem in its full generality is undecidable, so we set a more modest goal. We are interested in verifying the correctness of a program P on a specific input x, by running P on x and related inputs. It's not reasonable to expect the verification procedure to detect that P is incorrect when it's only wrong on inputs totally unrelated to x. We

also want the verification procedure to use less resources than the trivial procedure that solves the problem from scratch and explicitly checks whether P is correct on x. In our model, we assume oracle access to P, so we are not charged for running it as a black box procedure. Formally, we have the following definition.

Definition 3. *An* instance checker *with completeness c and soundness $s < c$ for a language L is a probabilistic polynomial-time oracle Turing machine C such that for all x, if $P = L$ (i.e., the program is correct) then $\Pr[C^P(x) \text{ accepts}] \geq c$, and if $P(x) \neq L(x)$ (i.e., the program is incorrect* on the given input*) then $\Pr[C^P(x) \text{ accepts}] \leq s$.*

Typical settings of the completeness and soundness parameters are $c = 1$, $s = 1/2$.

We can now demonstrate a very strong connection between instance checkers and probabilistically checkable proof systems.

Theorem 14. *A language L has an instance checker with completeness c and soundness s if and only if both L and $\overline{L}$ have a probabilistically checkable proof system with completeness c and soundness s such that the honest prover's responses are computable in polynomial time with oracle access to L.*

Proof. We first argue the forward direction. Assume that L has an instance checker with completeness c and soundness s. Note that this implies that $\overline{L}$ has an instance checker with the same parameters; it is the same as the instance checker for L but it flips the answers to the queries it makes. Thus by symmetry, it suffices to show that L has a probabilistically checkable proof system with the same parameters such that an honest prover's responses are computable in polynomial time with oracle access to L. We simply run the instance checker on our input x and use our proof to answer the instance checker's queries. The correct proof consists of the correct answers to the queries to L. When $x \in L$, completeness follows by the completeness of the instance checker. For $x \notin L$, we would like to use the soundness property of the instance checker. This only holds for oracles P with $P(x) \neq L(x)$, so we have the verifier first ask if x is in L before running the instance checker. If the proof claims $x \notin L$, we reject immediately. If the proof claims $x \in L$, we proceed as before. Completeness is preserved, and now for the case when $x \notin L$, a cheating proof is forced to falsely claim that $x \in L$ to get the verifier to accept. Then soundness follows by the soundness of the instance checker.

Now we argue the other direction. Assume that L and $\overline{L}$ have probabilistically checkable proof systems as in the statement of the theorem. Then by Theorem 6, each language has a multiple prover interactive proof system with at least as good of parameters and where the honest provers' responses are computable in polynomial time with oracle access to the language. We design an instance checker for L. On input x, we ask our oracle whether $x \in L$. If the oracle says yes, then we run the multiple prover interactive proof system for L and otherwise we run the one for $\overline{L}$. In simulating one of the verifiers, we respond to queries by running the appropriate honest prover's strategy using our oracle. We accept if and only if the verifier we're simulating accepts. Suppose that our oracle agrees with L. Then we run a multiple prover interactive proof system such that x is in the language decided by it, and all the responses to the prover queries agree with the honest prover's strategy, so we accept with probability at least c. Suppose that our oracle disagrees with L on x. Then we run the wrong multiple prover interactive proof system; x is not in the language decided by it. In this case, it doesn't matter how we respond to the prover queries; we accept with probability at most s. □

Corollary 3. *There exist instance checkers with perfect completeness for Graph Nonisomorphism, every* PSPACE*-complete language, every* #P*-complete function (provided we extend our definition of instance checker to function problems in the natural way), and every* EXP*-complete language.*

Proof. We noted at the beginning of Section 4 that the honest prover's strategy for the private-coin interactive proof system for Graph Nonisomorphism can be implemented in polynomial time with an oracle for Graph Nonisomorphism. This protocol also has perfect completeness. There is trivially an interactive proof system with perfect completeness and soundness 0 for Graph Isomorphism, where the honest prover's strategy is just the problem of finding an isomorphism between two graphs if one exists. As noted in a previous lecture, this problem is solvable in polynomial time with oracle access to (the decision version of) Graph Isomorphism. The first result now follows from the proof of Theorem 14 (bypassing the probabilistically checkable proof systems in the statement of the theorem and directly applying the above interactive proof systems).

Similarly, the result holds for every PSPACE-complete language using the interactive proofs for PSPACE developed in Section 2 and the fact that all polynomial-space computations can be carried out in polynomial time with an oracle for a PSPACE-complete language.

The third result holds using an argument similar to the proof of Theorem 14 where the instance checker first asks its oracle what the value of the function is on the given input x, then parsimoniously reduces x to a #SAT formula and runs the sumcheck protocol. The honest prover's strategy can be implemented in polynomial time with oracle access to #SAT and hence with oracle access to L by completeness.

For the final result, recall from the proof of Theorem 13 that every EXP-complete language has a multiple prover interactive proof system where the honest provers' responses are computable in exponential time and hence in polynomial time with oracle access to an EXP-complete language. This protocol also has perfect completeness. Since EXP is closed under complement, the result then follows from the proof of Theorem 14. □

CS 810: Complexity Theory	3/30/2007
Lecture 25: Harmonic Analysis	
Instructor: Dieter van Melkebeek	Scribe: Nathan Collins

In the last lecture we discussed probabilistically checkable proofs and the PCP theorem (which we did not prove). Today we give an alternate (equivalent) version of the PCP Theorem that is more useful for making hardness of approximation arguments. We also give two examples of hardness of approximation results, for MAX-3-SAT and MAX-IND-SET. Finally, we prove one inclusion on a weaker version of the PCP Theorem stated in the last lecture. One part of this proof uses discrete harmonic analysis, so we introduce enough discrete harmonic analysis to present that part of the argument.

1 PCP Theorems and Hardness Approximation

Recall the PCP Theorem stated last time:

Theorem 1 (The PCP Theorem)**.** $\text{NP} = \text{PCP}(O(\log n), O(1))$

A $\text{PCP}(r(n), q(n))$ for the language L is a probabilistic oracle TM V (the verifier) s.t.

- If $x \in L$ then $(\exists \Pi) \Pr[V^{\Pi}(x)] = 1$.
- If $x \notin L$ then $(\forall \Pi) \Pr[V^{\Pi}(x)] \leq 1/2$.

where the quantification is over proofs and the probability is over random coins flipped by the verifier. The verifier uses $r(n)$ random coins and makes $q(n)$ queries to the proof Π.

1.1 An Approximation Equivalent to the PCP Theorem

To see the relationship between the PCP Theorem and hardness of approximation we prove that the following theorem is equivalent to the PCP Theorem.

Theorem 2. *There exists $\alpha < 1$ and a poly-time computable f from 3-SAT to 3-SAT s.t. if a 3-CNF x is* not *in 3-SAT then the fraction of clauses of the 3-CNF $f(x)$ that can be simultaneously satisfied is less than α, and if x is in 3-SAT, then so is $f(x)$.*

Theorem 3. *The* PCP *Theorem is equivalent to Theorem 2.*

Proof. ($\Downarrow$): Assuming the PCP Theorem, there exist constants q (the $O(1)$ number of queries to the proof) and c (the constant in the $O(\log n)$ number of random bits used), and a poly-time verifier V, such that for each 3-CNF x

- If $x \in$ 3-SAT then there exists a proof Π

$$\Pr_{\rho \in \{0,1\}^{c \log n}}[V^{\Pi}(x)] = 1. \tag{1}$$

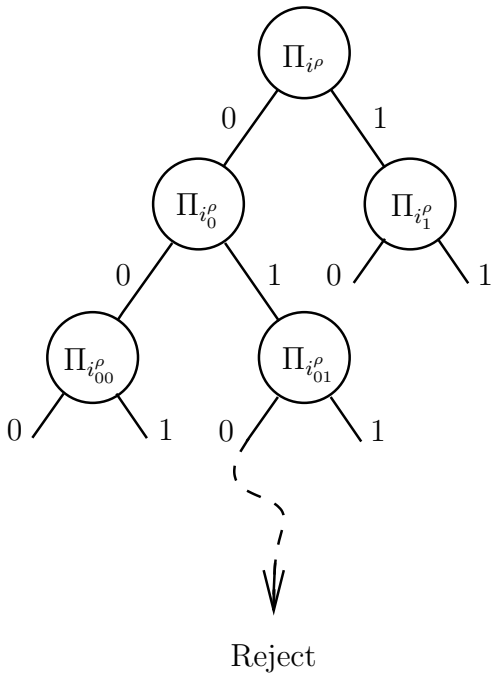


Figure 1: The bits in Π queried by V^Π_ρ depend only on ρ and bits of Π previously queried. The nodes are labeled by the queried proof bit and the outgoing edges are labeled by the nodes value.

- If $x \notin$ 3-SAT then for all proofs Π

$$\Pr_{\rho \in \{0,1\}^{c \log n}}[V^\Pi(x)] \leq 1/2. \tag{2}$$

Let ρ denote a random bit string in $\{0,1\}^{c \log n}$ and V_ρ denote V supplied with random bits ρ. Once ρ is fixed, the value of $V_\rho(x)$ is completely determined by the (up to) q bits of the supplied proof Π that $V_\rho(x)$ queries. The first bit that $V_\rho(x)$ queries is independent of Π, and in general the k^{th} bit queried is determined by the values of the previous $k-1$ bits queried. Figure 1 illustrates a decision tree for this procedure, in which the index $i^\rho_{b_1,\ldots,b_k}$ is the $k+1^{st}$ proof bit that is queried when the first k bits queried had the values $b_1, \ldots, b_k$.[1]

Now we construct a CNF representation of the decision tree. Let the variable x_i represent the value of the i^{th} bit in Π. Then the disjunction of all rejecting path "signatures" gives a DNF that is true iff $V^\Pi_\rho(x)$ rejects, where by path signature we mean the conjunction of variables and variable negations that correspond to the query indices and values corresponding to that path. As Figure 2 illustrates, negating that DNF gives a CNF which is false iff $V^\Pi_\rho(x)$ rejects, and is hence equivalent to $V_\rho(x)$. Since there are at most 2^q paths, and each path has length at most q, the CNF constructed has at most 2^q clause, each of length at most q.

Let C_ρ denote the 3-CNF corresponding to the CNF constructed in the last step, gotten by expanding each ($\leq q$)-clause into at most q 3-clauses. Then C_ρ has at most $q2^q$ clauses and we

[1]The indices that are queried for a fixed Π can be written as $i^\rho, i^\rho_{\Pi_{i^\rho}}, i^\rho_{\Pi_{i^\rho}\Pi_{i^\rho_{\Pi_{i^\rho}}}}, i^\rho_{\Pi_{i^\rho}\Pi_{i^\rho_{\Pi_{i^\rho}}}\Pi_{i^\rho_{\Pi_{i^\rho}\Pi_{i^\rho_{\Pi_{i^\rho}}}}}}}$, etc.

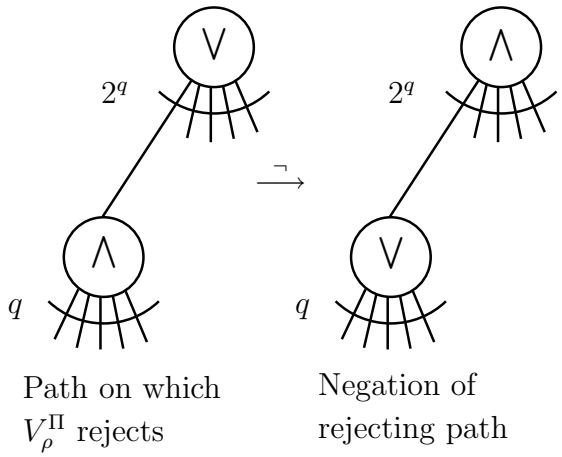


Figure 2: Since the set of paths is prefix free, V_ρ^Π rejects iff the query indices and values of a rejecting path are realized by Π. We form a DNF of rejecting path "signatures" and negate it to get a CNF that is true precisely when no rejecting path is realized.

define $f(x) = \bigwedge_\rho C_\rho$, where the top two levels of ANDs are contracted to form a CNF. Then $f(x)$ has at most $n^c q 2^q$ 3-clauses and either

- $x \in$ 3-SAT: Then $f(x) \in$ 3-SAT too, since the PCP system has perfect completeness (Equation (1)), and hence there exists Π that makes $V_\rho^\Pi(x)$ accept and C_ρ be satisfied for all ρ.
- $x \notin$ 3-SAT: Then $f(x) \notin$ 3-SAT, since the PCP system has soundness 1/2 (Equation (2)), and hence for all Π the computation $V_\rho^\Pi(x)$ rejects and C_ρ is unsatisfied for at least one half of all ρ. If C_ρ is unsatisfied then at least one of its clauses is unsatisfied, and so the proportion of $f(x)$'s clauses that are unsatisfied in this case is at least $(1/2)(1/q2^q) > 0$, giving $\alpha = 1 - 1/q2^{q+1} < 1$

($\Downarrow$, alternative construction): Let ρ denote a random bit string in $\{0,1\}^{c \log n}$ and V_ρ denote V supplied with random bits ρ. For each $\pi \in \{0,1\}^q$ simulate V_ρ on x, supplying bits from π as responses to V_ρ's proof queries, and keeping track of the proof bit indices that V_ρ queries. I.e., the first time V_ρ queries a proof bit, record the index queried and supply the first bit in π. On each subsequent query to the proof, check if the queried index has been queried previously and if so supply the same bit of π as before. Each time a *new* proof index is queried, supply the next unused bit in π. Let $V_\rho^{[\pi]}$ denote the simulated machine.

Each proof index queried by $V_\rho^{[\pi]}$ can depend on the values of the previously queried proof bits, but in any case $V_\rho^{[\pi]}$ queries at most q proof bits, so a q-bit π is sufficient for our simulation. Let the variable y_i represent the i^{th} bit in a hypothetical proof Π. For each $\pi \in \{0,1\}^q$ we can form a disjunction $\psi_{\rho,\pi}$ that is false iff the variables corresponding to the proof indices that $V_\rho^{[\pi]}$ queries agree with the values in π. E.g., if $q = 5$, $\pi = 01101$, and $V_\rho^{[\pi]}$ queries indices i_1, i_2, and i_3, then $\psi_{\rho,\pi} = y_{i_1} \vee \overline{y}_{i_2} \vee \overline{y}_{i_3}$.

We can now form a CNF φ that is satisfied iff its variables are set in such a way that they

correspond to a proof that makes V_ρ accept for each $\rho \in \{0,1\}^{c \log n}$. I.e.,

$$\varphi = \bigwedge_{\neg V_\rho^{[\pi]}(x)} \psi_{\rho,\pi},$$

where the conjunction is taken over all $\rho \in \{0,1\}^{c \log n}$ and $\pi \in \{0,1\}^q$ that cause $V_\rho^{[\pi]}$ to reject x.

If x is in 3-SAT, then Equation (1) says that there exists a satisfying assignment for φ, since a proof Π that makes V_ρ accept x for all ρ can't include the bits that make any *included* $\psi_{\rho,\pi}$ fail. If x isn't in 3-SAT, then Equation (2) says that any assignment will correspond to a proof Π that makes V_ρ reject for at least $1/2$ of all ρ's. In terms of our $\psi_{\rho,\pi}$ defined above, this says that we fail to satisfy some $\psi_{\rho,\pi}$ included in the conjunction φ, for at least $1/2$ of all ρ's. Now, there are at most $2^q n^c$ clauses in φ, so the fraction of clauses that are unsatisfied is at least $\frac{(1/2)n^c}{2^q n^c} = 1/2^{q+1}$

So far, we have $\leq_m^p$ reduced 3-SAT to SAT with "$\alpha = 1 - 1/2^{q+1}$." To finish this direction of the proof, we note that we can efficiently convert φ into an equivalent 3-SAT ${}_3\varphi$, where each clause C in φ corresponds to at most q 3-clauses in ${}_3\varphi$, and whenever an assignment fails to satisfy C it also fails to satisfy at least one of the q 3-clauses corresponding to C. So, completeness is preserved, and whenever x is not in 3-SAT, the fraction of clauses that any assignment fails to satisfy is at least $(1/q)(1/2^{q+1})$, and so we have $\alpha = 1 - 1/(q2^{q+1}) < 1$.

($\Uparrow$): This direction is easier. Suppose that f is a reduction and $\alpha < 1$ as in Theorem 2. Given a 3-CNF x let V^Π calculate $f(x)$ and treat Π as a Boolean assignment for $f(x)$. Since V can only access $O(\log n)$ proof bits, it can't simply check all of $f(x)$'s clauses. However, with $O(\log n)$ random bits, V can choose a clause of $f(x)$ at random and then check if Π satisfies it, accessing at most 3 bits of Π. This procedure has perfect completeness, since if $f(x)$ is satisfiable then Π can be a satisfying assignment. If $f(x)$ is unsatisfiable, then with probability α the clause V chooses of $f(x)$ is not satisfied by Π, giving soundness $\leq \alpha$. Our definition of PCP requires the soundness be $\leq 1/2$, so can we repeat this procedure $\lceil \frac{\log 1/2}{\log \alpha} \rceil$ times (the error is one-sided). This shows that 3-SAT $\in$ PCP$(O(\log n), O(1))$ and the PCP Theorem follows from the NP-completeness of 3-SAT. □

1.2 Implications of the PCP Theorem on Hardness of Approximation

Theorem 2 can be used to derive approximation bounds on certain NP optimization problems. Examples include

- MAX-3-SAT: If a poly-time approximation algorithm A for MAX-3-SAT existed which gave approximations to a factor better than α, then we could determine membership in 3-SAT by running A on $f(x)$ and accepting iff $A(f(x)) \geq (1-\alpha)\#f(x)$,[2] giving P = NP. We conclude that approximating MAX-3-SAT to within α is NP-hard.

  In fact, it turns out that for all $\varepsilon > 0$, it is NP-hard to $(7/8 + \varepsilon)$-approximate MAX-3-SAT in poly-time. This follows from the existence of a PCP$(O(\log n), 3)$ for the problem E-3-LIN (exactly 3 variables, linear equations) that has completeness $c = 1 - \varepsilon$ and soundness $s = 1/2 + \varepsilon$. An instance of E-3-LIN is a collection of linear equations: $x_{i,1} \oplus x_{i,2} \oplus x_{i,3} = b_i$ where the b_i are constants (either 0 or 1). The instance is in the language if there is a setting of the variables that satisfies each equation. The PCP for E-3-LIN works by generating three

[2] $\#f(x)$ denotes the number of clauses in $f(x)$.

indices i_1, i_2, and i_3, and a bit b, and verifying that $\Pi_{i_1} \oplus \Pi_{i_2} \oplus \Pi_{i_3} = b$. With this PCP system $s \geq 1/2$ since for random values $\Pi_{i_1} \oplus \Pi_{i_2} \oplus \Pi_{i_3} = b$ is satisfied half of the time. Also, it's unlikely to have $c = 1$ with this system, since deciding whether the system of equations has a solution can be solved in polynomial time using Gaussian elimination.

The approximation result for MAX-3-SAT follows from the PCP for E-3-LIN by converting each linear equation from an E-3-LIN instance into four clauses such that if the original equation is not true with a given assignment then at least one of the four clauses must be false. [3]

- MAX-IND-SET: Given a 3-CNF x, form a graph G corresponding to $f(x)$ as follows: For each 3-clause c in $f(x)$ add 7 nodes to G, where the nodes are labeled by c and one of the 7 possible satisfying assignments to c's 3 variables. Add an edge to G between any two nodes that together correspond to a conflicting variable assignment. Then for any clause c the 7 nodes corresponding to c form a clique, and so any anti-clique in G includes at most 1 node from each of these 7 node clusters. When $f(x)$ is satisfiable, we can choose 1 node from each cluster, so the max independent set has size the number m of clauses in $f(x)$. When $f(x)$ is not satisfiable, we can satisfy at most αm of $f(x)$'s clauses, and so the maximum independent set will have size at most αm. So, an α-approximation of MAX-IND-SET would imply P = NP.

  Using more "technology" we can get a $\frac{1}{n^{1-\varepsilon}}$-approximability bound, for some $\varepsilon > 0$, where n is the number of nodes in the graph. Note that this is a strong result since we can trivially get a $1/n$ approximation by picking a single node.

That's all we are going to say about hardness of approximation as it relates to PCPs.

2 A Weaker PCP Theorem

The proof of "the" PCP Theorem is quite elaborate, and we won't see it in this class, but we will prove a weaker PCP Theorem today. This result gives a flavor of the stronger PCP theorem - we are able to verify a proof by querying only a constant number of bits in the proof. The result we prove is weaker than "the" PCP theorem as we use a polynomial number of random bits.

Theorem 4. NP $\subseteq$ PCP($poly(n), O(1)$).

Proof. The idea is to somehow convert certificates of membership in an NP language into something easily verified by a PCP system. To do this we'll use quadratic equations and Hadamard coding. Given some 3-CNF φ we form an equivalent collection of quadratic equations and ask if they can be simultaneously satisfied. E.g., we translate the clause $c = x \vee \overline{y} \vee z$ into the polynomial equation $(1-x)y(1-z) = 0$ and introduce a new variable $\xi = xy$ to reduce the degree from 3 to 2. Working over $\mathbb{Z}_2$ we have an equivalent system of two equations: $y \oplus yz \oplus \xi \oplus \xi z = 0$, and the new equation $\xi \oplus xy = 0$.

So, suppose we have converted φ into a system of N quadratic equations in n variables $x_1, \ldots, x_n$. Notice that all the equations will have zero on the right hand side, and denote the k^{th} left hand

[3] See lecture 3, page 6, from CS 880 in 2004 for a more detailed explanation of the hardness of approximation result for MAX-3-SAT.

side by Q_k. Then for each $k \in [N]$ we have $Q_k = \bigoplus_{i,j\in[n]} q_{kij} x_i x_j$ for some coefficients q_{kij}. The satisfiability of φ is thus reduced to finding a solution to the system $Q_k = 0$, for $k \in [N]$.

Given a candidate assignment $a = (a_1, \ldots, a_n)$ of the x_is we have

$$\Pr_{\rho\in\{0,1\}^N}[\underbrace{\bigoplus_{k\in[N]} \rho_k Q_k(a) = 1}_{(*)}] = 1/2, \tag{3}$$

if a fails to satisfy $Q(a) \equiv 0$, since $\bigoplus_{k\in[N]} \rho_k Q_k(a)$ is the inner product of $Q(a)$ (a *non-zero* vector) with a random vector ρ. If a is a satisfying assignment, then the probability in (3) is zero.

Define $b_{ij} = a_i a_j$ for all $i, j \in [n]$. Then any quadratic in $\{a_i\}$ is linear in $\{b_{ij}\}$. Let the proof π be a Hadamard encoding of b. Then checking $(*)$ amounts to querying one position in π: the position corresponding to the vector with ij^{th} entry given by $\bigoplus_{k\in[N]} \rho_k q_{kij} b_{ij}$. For soundness we need to be able to reject π's that don't Hadamard encode a b as defined above. To probabilistically check that a given π is such a Hadamard code we perform the following tests

- Test 1: Probabilistically check that π is close to a Hadamard encoding of some b. Once we know π is close to a Hadamard code we can use local decodability.
- Test 2: Probabilistically check that $b_{ij} = b_{ii} b_{jj}$. This is true for b as above since $x = x^2$ in GF(2).

To perform Test 1 we choose $x, y \in \{0,1\}^{n^2}$ at random and verify that $\pi(x) \oplus \pi(y) = \pi(x \oplus y)$. Completeness of this test is clearly 1. If π is *not* close to any valid Hadamard code, i.e.

$$\forall b \in \{0,1\}^{n^2} \Pr_{x\in\{0,1\}^{n^2}}[\pi(x) \neq \langle b, x\rangle] \geq \gamma \tag{4}$$

for some $\gamma > 0$, then

$$\Pr[\text{Test 1 fails}] \geq \gamma. \tag{5}$$

To prove that (4) implies (5) we will use harmonic analysis, and we postpone that proof until the end of the current proof. So, assume Test 1 passes with high probability.

To perform Test 2, we define matrices A and B by $A_{ij} = b_{ii} b_{jj}$ and $B_{ij} = b_{ij}$, choose $x, y \in \{0,1\}^{n^2}$ at random, and check that

$$x^\top A y = x^\top B y.$$

If $A = B$, i.e. if b is of the desired form, then Test 2 passes with probability 1. If $A \neq B$, then the probability that Test 2 fails is at least 1/4, since if $A \neq B$, then $x^\top A \neq x^\top B$ with probability at least 1/2, since A and B differ in at least one column (think inner products again), and whenever $x^\top A \neq x^\top B$, we have $x^\top A y = x^\top B y$ with probability exactly 1/2. This test requires only three queries to the proof, since

$$x^\top A y = \bigoplus_{i,j} x_i A_{ij} y_j = (\bigoplus_{i,j} x_i b_{ii})(\bigoplus_{i,j} b_{jj} y_j)$$

and

$$x^\top B y = \bigoplus_{i,j} x_i y_j b_{ij},$$

and so we can query π at the positions corresponding to the vectors with ij^{th} entry $\delta_{ij} x_i$, $\delta_{ij} y_j$, and $x_i y_j$, respectively, where $\delta_{ij} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$ is a Kronecker delta function. □

3 Harmonic Analysis

We now develop enough harmonic analysis to prove the implication (4) implies (5) in the proof of the weak PCP Theorem. Hopefully people have seen some form of *continuous* harmonic analysis on $\mathbb{R}$ or $\mathbb{C}$, where functions are approximated using sin and cos (which are called *harmonics*). Today we describe a *discrete* theory for functions

$$f : G \to \mathbb{C},$$

where G is a group. In this discrete theory the harmonics are *characters* of the group, i.e. homomorphisms from G into the multiplicative group of complex numbers $\mathbb{C}^\times$. Recall that a *homomorphism* is a function that respects the group operation, i.e. $\phi : G \to \mathbb{C}^\times$ is a character of G if for all $g, h \in G$ we have $\phi(gh) = \phi(g)\phi(h)$. Notice that if $g \in G$ has finite order, which is true of all $g \in G$ if G is finite, then $\phi(g) \in e^{i\mathbb{R}}$ the unit circle, since $\phi(g)^{\mathrm{ord}(g)} = 1$.

3.1 Properties of Group Characters

Before we get to our application of discrete harmonic analysis, we list without proof a few general properties of group characters for finite groups

- The group characters form an orthonormal set with respect to the inner product

$$\langle f_1, f_2 \rangle = \frac{1}{|G|} \sum_{g \in G} \overline{f_1(g)} f_2(g),$$

  where $x \mapsto \overline{x}$ is complex conjugation.

- There at most $|G|$ characters, since the space of functions $G \to \mathbb{C}$ is a $\mathbb{C}$-vector space of dimension $|G|$. For many groups the number of characters is equal to the size of $|G|$, in which case we biject G with its characters and write χ_g for the character corresponding to $g \in G$, and then the characters form an orthonormal basis for the space of $G \to \mathbb{C}$ functions. I.e., given $f : G \to \mathbb{C}$, there exist *Fourier coefficients* $\{\hat{f}(g)\}_{g \in G}$ such that

$$f = \sum_{g \in G} \hat{f}(g) \chi_g.$$

  The Fourier coefficients are easy to calculate since the χ_g are orthonormal, namely

$$\hat{f}(g) = \langle \chi_g, f \rangle.$$

- Let $\hat{f} = g \mapsto \hat{f}(g)$ denote the vector of f's Fourier coefficients. Then the map $f \mapsto \hat{f}$ "respects" the inner product, i.e. given $f_1, f_2 : G \to \mathbb{C}$ we have

$$\langle f_1, f_2 \rangle = |G| \langle \hat{f}_1, \hat{f}_2 \rangle,$$

  which yields a Parseval identity

$$\|f\|_2^2 = |G| \|\hat{f}\|_2^2. \tag{6}$$

- For the *convolution* of f_1 and f_2, defined by

$$(f_1 * f_2)(z) = \frac{1}{|G|} \sum_{x+y=z} f_1(x) f_2(y),$$

we get the relation

$$\widehat{f_1 * f_2} = \hat{f}_1 \hat{f}_2.$$

This last equality allows for the reduction of some $O(n^2)$ computations to $O(n \log n)$ computations in signal processing.

3.2 Completing the Proof of the Weak PCP Theorem

That's all we are going to say about harmonic analysis in general. For $G = (\{0,1\}^n, \oplus)$, or equivalently $(\{\pm 1\}^n, \cdot)$, where equivalence comes from the mapping $x \mapsto -1^x$, all elements are order 1 or 2, and so the characters are of the form $G \to \{\pm 1\} \subseteq \mathbb{C}$. In this case, we have characters

$$\chi_g(x) = (-1)^{\langle g,x \rangle}, \tag{7}$$

where the inner product in the exponent is given by

$$\langle x, y \rangle = \bigoplus_{i \in [n]} x_i y_i.$$

Such χ_g are characters since $\langle g, x \oplus y \rangle = \langle g, x \rangle \oplus \langle g, y \rangle$. The fact that $\langle g, x \rangle$ is the x^{th} bit of the Hadamard code for g provides the connection with our previous work. Since our characters take values in $\{\pm 1\} \subseteq \mathbb{R}$, we can drop the complex conjugation in the definition of inner product for functions $G \to \mathbb{C}$, and we get

$$\begin{aligned} \hat{f}(g) = \langle \chi_g, f \rangle &= \mathop{\mathbb{E}}_{x \in G}[\chi_g(x) f(x)] \\ &= \Pr_x[f(x) = \chi_g(x)] - \Pr_x[f(x) \neq \chi_g(x)] \\ &= 1 - 2 \Pr_x[f(x) \neq \chi_g(x)], \end{aligned} \tag{8}$$

and more generally

$$\delta_{gh} = \langle \chi_g, \chi_h \rangle = \mathop{\mathbb{E}}_{x \in G}[\chi_g(x) \chi_h(x)]. \tag{9}$$

We now complete the proof of the Weak PCP Theorem, by proving that if for all $b \in \{0,1\}^{n^2}$

$$\Pr_{x \in \{0,1\}^{n^2}}[\pi(x) \neq \langle b, x \rangle] \geq \gamma$$

for some $\gamma > 0$, then

$$\Pr[\text{Test 1 fails}] \geq \gamma.$$

We prove the contrapositive via a calculation, where we use the $\{\pm 1\}$ domain for the values taken by π

$$\begin{aligned}
&1 - 2\Pr[\text{Test 1 fails}] \\
&= \Pr[\text{Test 1 passes}] - \Pr[\text{Test 1 fails}] \\
&= \mathop{\mathbb{E}}_{x,y}[\pi(x)\pi(y)\pi(x \oplus y)] \\
&= \mathop{\mathbb{E}}_{x,y}[(\sum_{g_1} \hat{\pi}(g_1)\chi_{g_1}(x))(\sum_{g_2} \hat{\pi}(g_2)\chi_{g_2}(y))(\sum_{g_3} \hat{\pi}(g_3)\chi_{g_3}(x)\chi_{g_3}(y))]
\end{aligned}$$

by switching to the Fourier domain and using the fact that χ_{g_3} is a homomorphism

$$= \sum_{g_1,g_2,g_3} \hat{\pi}(g_1)\hat{\pi}(g_2)\hat{\pi}(g_3) \mathop{\mathbb{E}}_{x}[\chi_{g_1}(x)\chi_{g_3}(x)] \mathop{\mathbb{E}}_{y}[\chi_{g_2}(y)\chi_{g_3}(y)]$$

by linearity of expectation, the independence of x and y, and the fact that $\mathbb{E}[\chi_g(x)] = 1$

$$= \sum_{g} (\hat{\pi}(g))^3$$

by (9)

$$\begin{aligned}
&\leq \max_g \hat{\pi}(g) \sum_g (\hat{\pi}(g))^2 \\
&= \max_g \hat{\pi}(g) \qquad (10)
\end{aligned}$$

since $\sum_g (\hat{\pi}(g))^2 = 1$, by (6)

$$= 1 - 2\Pr_x[\pi(x) \neq \chi_{g_M}(x)]$$

by (8), where g_M maximizes (10). Rearranging, we get

$$\Pr_x[\pi(x) \neq \chi_{g_M}(x)] \leq \Pr[\text{Test 1 fails}] \qquad (11)$$

which finishes the proof, since by (7) we can rewrite (11) as

$$(\exists g_M \in \{0,1\}^{n^2}) \Pr_x[\pi(x) \neq \langle g_M, x \rangle] \leq \Pr[\text{Test 1 fails}],$$

if we again interpret π to take values in $\{0,1\}$, and this is the contrapositive of what we set out to prove.

CS 810: Complexity Theory 4/11/2007

Lecture 26: Cryptographic Primitives

Instructor: Dieter van Melkebeek Scribe: Chi Man Liu

In this lecture, we give an overview of the complexity-theoretic aspects of cryptography. We focus on confidentiality and look at one cryptographic primitive: the one-way function. Existence of one-way functions is still an open question. We give some examples of candidate one-way functions, and relate the existence of one-way functions to some other complexity-theoretic hypotheses, in particular NP $\not\subseteq$ P/poly.

1 Overview

Cryptography is the study of secure interaction between two or more parties. In the context of complexity theory, "parties" usually refer to efficient machines, and "interaction" refers to digital communication. There are several common interpretations for "secure".

1. *Confidentiality*: transforming data into other forms in order to prevent unauthorized parties from knowing the content by looking at the communication.
2. *Integrity*: ensuring that the received message comes from the right party, and its content has not been changed.
3. *Authenticity*: verifying the identity of trusted parties.

Confidentiality, integrity and authenticity are the traditional goals of cryptography. Other goals, such as privacy, have also been studied. In this lecture, we focus on confidentiality — hiding information from others.

2 Confidentiality

We consider a very simple model. Suppose that Alice wants to send a message M to Bob over some channel. An eavesdropper, Eve, tries to figure out what Alice has sent to Bob by intercepting the communication between them. Therefore, to secure the message against Eve, Alice has to massage M into something else before sending it to Bob. After receiving the message, Bob recovers the original content of M by some operation. In order for this scheme to work, either Alice or Bob (or both) must know something more than Eve does; otherwise, Eve would have simulated Bob's actions to recover M.

Formally, Alice runs an encryption algorithm E with some encryption key K_E on the plaintext M. The algorithm outputs a ciphertext C, which is sent to Bob over the channel. At Bob's side, a decryption algorithm D with decryption key K_D is run on the ciphertext C. We expect the decryption algorithm to output M. We need to pick E, D, K_E and K_D such that Eve is unable to recover M by just looking at C.

We look at two different settings. The first one is *private key systems* (also known as symmetric key systems) where $K_E = K_D$. The second one is *public key systems* (also known as asymmetric

key systems) where $K_E \neq K_D$. Moreover, we require that K_E can be easily computed from K_D but not the other way round.

2.1 Private Key Systems

In a private key system, there is only one key K, which is used both for encryption and decryption. Note that although only one key is used, the encryption algorithm may be different from the decryption algorithm.

A very simple private key system is the *one-time pad*. The encryption algorithm outputs $E_K(M) = M \oplus K$. The decryption algorithm outputs $D_K(C) = C \oplus K$. We pick K uniform at random from $\{0,1\}^n$, where n is the length of the message. No matter how much computational resources the eavesdropper has, she cannot correctly determine the plaintext M with a probability higher than 2^{-n}. Information theoretically, this is the best we can hope for. In other words, the one-time pad system has perfect security.

One drawback to the one-pad time system is that the key needs to have the same length as the message. If we want to send a message of size n, we have to generate n random bits. We can relax the notion of security from information-theoretic to complexity-theoretic. Instead of requiring the system to be secure against eavesdroppers with unlimited computational resources, we only require it to be secure against computationally bounded adversaries, e.g. polynomial-time machines. We achieve this by using pseudorandom generators with short seed length that fool computationally bounded adversaries. With a such PRG, we can first pick a short random string and then apply the PRG to get a long pseudorandom string K which is used as the key.

The PRGs we consider here are different from the PRGs we used for derandomization. Instead, we look at *cryptographic PRGs* (CPRGs). There are a few differences between CPRGs and PRGs for derandomization.

- *Efficiency.* When we discussed time-bounded derandomization, we only require PRGs to be computable in linear exponential time. However, we expect the generation of encryption keys to be efficient, hence linear exponential time wouldn't work for CPRGs. We want the running time of CPRGs to be polynomial.

- *Strength.* When derandomizing certain complexity classes, we know the "adversary" (the target to be fooled), e.g. circuits of quadratic size for the case of BPP. In our cryptographic setting, the adversary we are trying to fool is a polynomial-time computing device. However, we do not know the degree of that polynomial. The CPRG runs in time $O(n^c)$ for some fixed constant c, yet it has to fool any $O(n^d)$-time adversary with arbitrarily large constant d. This implies that our "hardness versus randomness" technique applied to constructing quick PRGs for circuits wouldn't work for CPRGs.In later discussions, we focus on CPRGs against nonuniform models.

- *Seed length or stretch.* In derandomization, we want the seed length to be $O(\log r)$, i.e. the stretch is at least linear exponential. In cryptography, we only require the stretch to be polynomial.

We will talk more about CPRGs in the next lecture.

Other examples of private key systems include the Data Encryption Standard (DES) and the Advanced Encryption Standard (AES). These systems have been shown to be secure under some specific complexity-theoretic hypotheses.

2.2 Public Key Systems

In a public key system, the encryption key K_E and the decryption key K_D are different in general. K_E can be computed by K_D efficiently, but not the other way round. Here is how it works. First, Bob picks K_D randomly and out of it he computes K_E. Then, he makes K_E public to everyone he expects to receive messages from, while keeping K_D private. Now, if someone wants to send Bob a message, she has to encrypt the message with the *public key* K_E and send the ciphertext over the channel. After receiving the ciphertext, Bob uses his *private key* K_D to decrypt it back into plaintext message. A widely used public key system is the RSA encryption system.

One advantage of public key systems over private key systems is that when m parties want to communicate with each other, only m pairs of keys are needed using a public key system, whereas $O(m^2)$ are needed in a private key system. However, many private key systems are still being used. One reason is that most public key systems are more computationally intensive than private key systems. Besides, known algorithms for breaking public key systems are more efficient than those for private key systems, although they all run in exponential time. As a result, the key size of a public key system tends to be larger than that of a private key system.

3 One-Way Functions

The encryption systems in the previous section become vulnerable if $P = NP$. However, P $\neq$ NP does not guarantee good security. Practically, we want it to be case that NP $\not\subseteq$ BPP, or even NP $\not\subseteq$ P/poly. NP $\not\subseteq$ BPP rules out the possibility of having efficient randomized algorithms for breaking the above systems. NP $\not\subseteq$ P/poly states that NP does not have polynomial-size circuits. These encryption systems would become insecure if NP has polynomial-size circuits, for one can compute polynomial-size circuits for certain input sizes (possibly using a huge amount of resources), and use them to break the systems efficiently on those input sizes. Yet, these assumptions are still not enough to ensure confidentiality. In this section, we look at one-way functions, which capture the notion of hardness in computing the decryption key given the encryption key. Intuitively, a one-way function is a function which is easily computable in one direction, but hard to invert on average.

3.1 Definition

There are two issues in encryption systems that do not seem to be solved by just assuming NP $\not\subseteq$ P/poly.

- We look at the *average-case hardness* of a function instead of worst-case hardness. We want the function to be hard to invert for not just a few inputs but most.
- We require the existence of an efficient procedure to generate "solved" hard instances.

Definition 1. *Given a function $f : \{0,1\}^* \to \{0,1\}^*$, the* hardness of inverting f on input size n, *denoted $HI_f(n)$, is the largest integer s such that no circuit C of size at most s succeeds in inverting f on at least a $1/s$ fraction of the inputs of size n.*

Note that $s = HI_f(n)$ is at most exponential in n, since any Boolean function with n inputs has circuits of size $2^{O(n)}$. As s gets smaller, the circuit C becomes more computationally restrictive,

yet it is required to successfully invert a larger fraction of inputs. Thus $HI_f(n)$ serves as a measure of average-case hardness of inverting f on input size n.

We need a more precise definition for the statement "a circuit C succeeds in inverting a function f on at least a $1/s$ fraction of the inputs of size n". For a permutation (a one-to-one onto function) f, the preimage of an output is always unique. The statement can be restated as

$$\Pr_{|x|=n}[C(f(x)) = x] \geq \frac{1}{s}.$$

For a general function f, an output may have multiple preimages. We only require C to compute one of the preimages, that means

$$\Pr_{|x|=n}[f(C(f(x))) = f(x)] \geq \frac{1}{s}.$$

We now give the definition of one-way functions.

Definition 2. *A function $f : \{0,1\}^* \to \{0,1\}^*$ is a* one-way function *if f is polynomial-time computable and $HI_f(n) \geq n^{\omega(1)}$.*

In the uniform setting, circuits are replaced by any polynomial-time randomized algorithm. Intuitively, a function f is hard to invert if for any polynomial-time randomized algorithm A, A fails to invert f on a significant fraction of the inputs as the input size gets larger.

3.2 Candidate One-Way Functions

We give a few candidates for one-way functions. Note that they are candidates only — one-way functions do not exist if P = NP.

- *Multiplication of primes:*
$$f(p,q) = pq,$$
where p and q are primes. Factorization of integers is believed to be a hard problem.

- *Squaring modulo pq:*
$$f(p,q,x) = (pq, x^2 \bmod pq),$$
where p, q are primes and x is an integer. It can be shown that inverting this function is equivalent to factoring in terms of average-case complexity. As an aside, finding square roots modulo p is a very easy problem. That's why we need to use modulo pq.

- *Exponent modulo p:*
$$f(p,g,x) = (p,g,g^x \bmod p),$$
where p is a prime, g is a generator of the multiplicative group modulo p, and x is an integer between 1 and $p-1$. Computing f is easy using a divide-and-conquer approach (repeated squaring); however, inverting f is hard as it is essentially the *discrete logarithm problem* for which no efficient algorithm is known. One interesting property of this function is that it becomes a permutation if we fix p and g.

- *RSA:*

$$f(p, q, e, x) = (pq, e, x^e \bmod pq),$$

where p, q are primes, and e, x are integers satisfying $\gcd(x, \phi(pq)) = 1$ [1]. This function is also a permutation after fixing p, q and e. Inverting f becomes easy once d, the inverse of e modulo $\phi(pq)$, is given. One-way functions which are easy to invert given some special information are called *trapdoor functions*. In an RSA encryption system, e and d are used as the public key and private key respectively.

It is conjectured that for all of the above functions f,

$$HI_f(n) \geq 2^{n^\epsilon}$$

for some constant $\epsilon > 0$ which depends on f. Note that this statement is stronger than that required by a one-way function.

3.3 Existence of One-Way Functions

It is still an open question whether NP $\not\subseteq$ P/poly implies the existence of one-way functions. As mentioned previously, there are two ingredients we need to go from NP $\not\subseteq$ P/poly to the existence of one-way functions: average-case hardness of functions and efficient generation of "solved" hard instances. The second ingredient is in fact easy to obtain. For example, let f be a function defined by

$$f(\phi, x) = \begin{cases} \phi & \text{if } x \text{ satisfies } \phi \\ \rho & \text{otherwise} \end{cases}$$

where ϕ is a Boolean formula, x is an assignment for ϕ, and ρ is some trivial Boolean formula. Then, inverting f is equivalent to finding a satisfying assignment for ϕ, which is hard as long as P $\neq$ NP. Thus the key to showing the implication is to boost hardness from worst-case to high average-case. We encountered this before when we discussed how to go from worst-case hardness to high average-case hardness for functions in E by applying error-correcting codes with list decoding. However, the techniques for E fail in our case since they require exponential time to compute. Plausible directions to tackling this problem have been proposed. Some of them are discussed below.

3.3.1 Random Self-Reducibility

We use the discrete logarithm problem as an example. Recall the discrete logarithm problem:

> Given a prime p, a generator g of the multiplicative group modulo p, and an integer y $(1 \leq y < p)$, find x such that $y = g^x \bmod p$.

Consider the following algorithm, which uses an oracle that solves the discrete logarithm problem on some fraction of inputs.

1. Pick r uniformly at random.
2. Query the oracle to find s such that $g^r y = g^s \bmod p$.

[1] $\phi(n)$ is the *Euler totient function*. For different primes p and q, $\phi(pq) = (p-1)(q-1)$.

3. Output $s - r$.

Note that the distribution of $g^r y$ in step 2 is uniform provided that r is picked uniformly at random. If the oracle correctly solves a nontrivial fraction of inputs (say, $1/\operatorname{poly}(n)$) for all input size n, the above algorithm is a randomized algorithm for finding discrete logarithms. This shows that worst-case hardness of the discrete logarithm problem implies average-case hardness. Unfortunately, the discrete logarithm problem, as well as many other problems that exhibit random self-reducibility, are not complete for NP.

3.3.2 Concatenation

Given a function f, define

$$g(x_1, \ldots, x_k) = f(x_1) \circ \cdots \circ f(x_k),$$

where $\circ$ represents string concatenation. Inverting g on $(x_1, \ldots, x_k)$ requires inverting f on all of $x_1, \ldots, x_k$. If we can invert f only on a small fraction of inputs, we expect ourselves to be able to invert g on an even smaller fraction of inputs. Formally, suppose that we can only invert f on an ϵ fraction of inputs (of length n). Then, the fraction of inputs on which we can invert g is no more than ϵ^k. However, the best bound we have for worst-case hardness is $\epsilon > 1 - 2^{-n}$. Boosting hardness through concatenation thus requires k to be exponential in n, which is not feasible.

4 Next Lecture

Next time we will show that the existence of one-way functions allows us to construct cryptographic PRGs. We will also focus on authentication protocols based on zero-knowledge proof systems.

CS 810: Complexity Theory 4/13/2007

Lecture 27: Zero-Knowledge

Instructor: Dieter van Melkebeek Scribe: Matthew Anderson

Last lecture we looked at one cryptographic primitive: the one-way function. Today we will see several applications of one-way functions in the construction of cryptographic protocols. All cryptographic primitives imply the existence of one-way functions, so in order to accomplish anything interesting we must assume their existence. We show how to construct a cryptographic pseudorandom generator using one-way functions, which in turn can be used to construct a protocol for confidential information transfer. We will also discuss several interactive protocols for authentication that allow a party to prove their identity but provide the verifier with no extra information, these types of protocols are known as zero-knowledge proofs.

1 Authentication Using One-Way Functions

One-way functions are functions that are computable in polynomial time in one direction but are hard to invert in the average case. The notion of hardness of some boolean function f, HI_f, was the size, s, such that all circuits of size less than s cannot compute f correctly on more than a fraction of $\frac{1}{2} + \frac{1}{s}$ of the inputs. A cryptographic pseudorandom generator (CPRG) is a standard PRG with the additional restriction that the sequence should be computable in polynomial time from the seed (rather than exponential time that is the case of standard PRGs). CPRGs maintain the property that the output sequence is not distinguishable from uniform by any polynomial time adversary (asymptotically). In our discussion we restrict ourselves to a subclass of one-way functions, namely one-way permutations. All of the results that we present here can be reformulated in the context of the more general one-way function; however, the analysis is more complex.

1.1 A Simple Flawed Authentication Protocol

Start with some one-way permutation f. Recall that for any authentication protocol to have an advantage the parties involved must have some additional knowledge that any other party (such as an eavesdropper) does not have access to. In this case both parties, A and B, know something different: A knows it's password, x, and B stores $f(x)$. In order to authenticate A's identity A will send B $f(x)$. B can easily verify that $f(x)$ matches the value stored for A. Since B stores the password as $f(x)$ instead of the plain text x and f is a one-way permutation it is difficult to invert and recover x from $f(x)$. This prevents even whoever is maintaining the list of values at B from inverting the function to determine x. Unfortunately this protocol is seriously flawed. Any eavesdropper, E, can watch A send $f(x)$ to B; then when E wants to pretend to be A, E simply repeats the message $f(x)$ to B which B has no choice but to accept.

1.2 A Better Authentication Protocol

We can apply the previous flawed protocol iteratively so that a password is used to authenticate only one time. This protocol will allow A to authenticate k times with B. A picks a password x, then computes $x \to f(x) \to f(f(x)) \to ... \to f^k(x)$. Where f^i is f composed i times with itself.

Ahead of time B stores $f^k(x)$. During its first login A provides B with $f^{k-1}(x)$. B verifies that $f(f^{k-1}(x))$ matches the value stored. B then stores $f^{k-1}(x)$. On the next login A provides $f^{k-2}(x)$, again B verifies $f(f^{k-2}(x))$ matches the value stored and updates the stored value. During the i^{th} login A sends $f^{k-i}(x)$ and B verifies $f(f^{k-i}(x))$ matches the value stored and updates the value. If x is chosen uniformly at random and f is a one-way permutation then it should be hard to compute $f^{k-i}(x)$ from $f^{k-i+1}(x)$. If the permutation is hard, then an adversary will not be able to invert a single step with more than $\frac{1}{\text{poly}}$ probability. This means the chance to invert an entire polynomial length sequence is very small if k is some polynomial in $|x| = n$.

One negative aspect of this protocol is that the number of logins, k, needs to be known *a priori*. Zero-knowledge proofs will be a way to get around this constraint. We discuss these later in this lecture.

2 Constructing CPRGs

We can leverage our construction in the previous section to build a CPRG. Consider looking at a prefix of the interaction sequence between A and B: $(f^k(x), f^{k-1}(x), ..., f^i(x))$. If one had a procedure that given a prefix could determine with high probability the next element of the sequence, $f^{i-1}(x)$, then one could determine the entire sequence with high probability. Last lecture we showed that PR sequences are unpredictable (otherwise one could construct a small circuit that could distinguish them from uniform). The protocol from the previous section almost gives us a CPRG with x as the seed. However, if a sequence is PR it should also be symmetric (this sequence is easy to compute in one direction). The proof of unpredictability of PR sequences depended on the fact that the sequence was a bit sequence, in this situation we do not have a bit sequence. For example, using the exponentiation one-way function we saw last time, the lowest order bit is efficiently predictable (using some properties of quadratic residues), though the highest order bit is as hard to compute as the function itself. In the squaring and RSA function all bits are difficult to compute. We can formalize this idea of hard to compute bits:

Definition 1 (Hardcore bit)**.** *A hardcore bit for f is a polynomial time boolean function h such that for any $c > 0$, all circuits C of size $\leq n^c$,*

$$\Pr_{|x|=n}[C(f(x)) = h(x)] \leq \frac{1}{2} + \frac{1}{n^c}. \tag{1}$$

Once we have a hardcore bit, h, for f we can use our protocol to construct a CPRG with the sequence $(h(f^k(x)), h(f^{k-1}(x)), ..., h(f(x)), h(x))$. The input is a seed of length n and the output is a bit sequence of length k, where k is polynomial in n.

Claim 1. *This sequence is a CPRG.*

Proof. Suppose that the sequence were predictable: that given a prefix $(h(f^k(x)), ..., h(f(x)))$ one could compute $h(x)$ using some small circuit C. Then there exists another (slightly larger) circuit C' that computes $h(x)$ from $f(x)$ with high probability. We can construct C' by building the prefix, which can be done efficiently in this direction, then applying C to the result. Therefore C' is a small circuit that computes $h(x)$ from $f(x)$ which contradicts the definition of h as a hardcore bit. □

This is reasonable but do such hardcore bits h exist? We can show how to construct an appropriate h from any one-way permutation f (this also holds for one-way functions):

Lemma 1. *For all one-way permutations f, the function $g(x,y) = (f(x), y)$ with $|x| = |y|$ is a one-way permutation and $h(z = (x,y)) = \langle x, y\rangle$ is hardcore for g.*

$\langle x, y\rangle$ is the inner product between x and y modulo 2 ($\langle x, y\rangle = \sum_i x_i y_i (\mod 2)$). We can rewrite our CPRG output sequence for a uniformly chosen x and y:

$$\langle f^k(x), y\rangle, ...\langle f^2(x), y\rangle, \langle f(x), y\rangle, y. \tag{2}$$

We can add y to the end because it was selected uniformly at random.

Theorem 1. *The existence of a one-way permutation implies that for all $\epsilon > 0$ there exists a CPRG with seed length $l(n) = n^\epsilon$.*

Proof. Follows directly from Lemma 1 and Claim 1. □

Proof. (Lemma 1) As defined g is a one-way permutation. This follows because f is a one-way permutation and y is tacked on to the end.

We show that h is a hardcore bit by contradiction. Suppose there exists a circuit C, for some $\epsilon = \frac{1}{\text{poly}}$ such that:

$$\Pr_{z=(x,y)}[C(g(z)) = h(z)] \geq \frac{1}{2} + \epsilon. \tag{3}$$

Then by an averaging argument for a fraction at least ϵ of the x's,

$$\Pr_y[C(f(x), y) = \langle x, y\rangle] \geq \frac{1}{2} + \frac{\epsilon}{2}. \tag{4}$$

But $\langle x, y\rangle$ taken over all y is just the Hadamard encoding of x. This gives us a small circuit that predicts the Hadamard code. We can use our list decoding procedure, with high probability, to make a list of all possible x candidates. Since $\epsilon = \frac{1}{\text{poly}}$ the decoding algorithm runs in polynomial time. We then run f on all values in the list and pick one that works. For at least ϵ fraction of x (namely those for which (4) holds), this procedure works with high probability. This means we can invert f on a fraction at least $\frac{1}{\text{poly}}$ inputs with high probability. This contradicts the hardness of f. So h is hardcore. □

We can use this CPRG to send secret messages between two parties by first establishing a common secret seed then xoring the PR sequence with the message text as a one-time pad.

2.1 Bit Commitment

Another application of one-way permutations is bit commitment. Bit commitment protocols work in two phases:

1. Commit: A chooses a bit b and sends some encoding of b to B. B should not be able to determine the value of b at this stage.

2. Reveal: A sends additional information to B so that B can determine the bit b that A committed. A should not have been able to alter the value of the bit that B determines from what A committed in the first phase.

We can think of this as follows: A sends a locked box with a message inside, and later sends the key for B to open the box. We can use one-way permutations to develop a protocol for bit commitment which is successful when both parties are limited to being computationally efficient:

1. Commit: A picks a commit bit b, and $x \in_R U$. A sends B $(f(x), b \oplus h(x))$.
2. Reveal: A sends x to B. B computes $f(x)$ and verifies it matches the first value. B computes $h(x)$ and $(b \oplus h(x)) \oplus h(x) = b$.

Suppose B were able to efficiently compute b after the first phase, then B would also be able to compute $h(x)$ from $f(x)$, violating the hardness of h. A cannot change the bit that B computes in the second phase because only one choice of x maps to $f(x)$ since we are using one-way permutations. Thus this protocol has the appropriate properties.

We will see an application of this protocol in Section 3.1.

2.2 Pseudorandom Functions

A pseudorandom function generator (PRFG) is a function that generates a function $\{0,1\}^n \to \{0,1\}^n$ that appears to be a completely random function to a computationally efficient adversary. This is formalized in the following definition

Definition 2 (Pseudorandom Function Generator)**.** *A polynomial-time computable function $F : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ is thought of taking one input to specify (i.e. output) a function, with $F(\rho, x) \to F_\rho(x)$ where ρ specifies the function and x is the actual argument. F is a PRFG iff for all $c > 0$, infinitely many n and all oracle circuits C of size at most n^c,*

$$\left| \Pr_{\rho \in_R U}[C^{F_\rho} = 1] - \Pr_{F \in_R U}[C^F = 1] \right| \leq \frac{1}{n^c}. \tag{5}$$

We can use a CPRG to construct such a PRFG.

Theorem 2. *If there exists a CPRG then there exists a PRFG.*

Proof. (Sketch) Given a polynomial time computable CPRG, $G : \{0,1\}^n \to \{0,1\}^{2n}$. We aim to construct a PRFG F. We will describe the construction but omit its proof, which follows in another hybrid argument as we have seen before.

Think of the argument x as describing a path (of length n) in a full binary tree. Consider G as (G_0, G_1) where G_0 produces the first half of the output of G and G_1 produces the second half. Then the output of $F_\rho(x) = G_{x_n} \circ G_{x_{n-1}} \circ ... \circ G_{x_1}(x)$ where x_i is the value of the i^{th} bit of x. This is the application of n choices of G_0 and G_1. □

Applying the same construction as above will give a regular PRG where the stretch is exponentially large for some seed ρ. The bits on the bottom level of the tree will specify the PR sequence. Each bit on the bottom level is computable in polynomial time by applying F_ρ to the appropriate x. This construction can be used to prove the following.

Theorem 3. *If there exists $f \in$ P with $HI_f(n) \geq 2^{n^\epsilon}$ then there exists a PRG $G : \{0,1\}^n \to \{0,1\}^{2^{n^\delta}}$ for $\delta = \Omega(\epsilon)$ such that each bit of G is computable in polynomial time and for all circuits C of size at most 2^{n^δ},*

$$\left| \Pr_{|x|=n}[C(G(x)) = 1] - \Pr_{|y|=2^{n^\delta}}[C(y) = 1] \right| \leq \frac{1}{2^{n^\delta}}. \tag{6}$$

However, for this to work a stronger hardness assumption is essential because now an exponential sized adversary circuit is appropriate since the stretch is exponential.

This type of PRG is critical to an argument of the difficulty of showing that P $\neq$ NP. Recall, if one wants to show that P $\neq$ NP a non-relativising technique must be used. Remember that the argument for the lower bound on parity for constant depth circuit does not relativize. However, it can be shown that the argument can be viewed as a "natural proof". These proofs try to prove P $\neq$ NP by showing that NP $\not\subseteq$ P/poly. That is, we would like to demonstrate an NP language L that requires superpolynomial size circuits. A proof that L has superpolynomial size circuits is called *natural* if it shows the existence of a property Π with the following properties:

1. $\Pi(\chi_{L/n}) = 1$.
2. If $\Pi(\chi_{L'/n}) = 1$ then $C_{L'}(n) = n^{\omega(1)}$.
3. Π can be computed in time $2^{O(n)}$.
4. A non trivial fraction of characteristic sequences, χ, have $\Pr[\Pi(\chi) = 1] \geq \frac{1}{2^{n^\delta}}$ for some $\delta > 0$.

All known circuit lower bound results can be characterized by these types of proofs for certain creative choices of Π. We will not show how to frame the lower bounds that we have seen in this form.

Theorem 4. *If such CPRGs exist then there are no "natural" proofs to establish super polynomial circuit lower bounds.*

Proof idea: Let G be a PRG according to Theorem 3 and interpret $G(x)$ as a characteristic sequence of some function. Each bit of $G(x)$ is computable in polynomial time so it must be the case that $\Pi(G(x)) = 0$ for all x, meaning $\Pr_x[\Pi(G(x)) = 1] = 0$. However, the last property of natural proofs says that $\Pr_\chi[\Pi(\chi) = 1] \geq \frac{1}{2^{n^\delta}}$. As Π is computable in time $2^{O(n)}$ and therefore also by a circuit of similar size, it is a distinguishing circuit contradicting the pseudorandomness of G. □

As it is conjectured that such CPRG do exist, the conjecture is that natural proofs do not. We can "rule out" natural proofs of circuit lower bounds for a number of specific problems as well. For example, the worst-case hardness of the discrete log is known to imply its average-case hardness, meaning it could then be used as the basis for a CPRG. Then applying Theorem 4 we get that there are no natural proofs for proving discrete log is exponentially hard.

3 Zero-Knowledge Proofs

Our motivation in discussing zero-knowledge(ZK) proofs is to improve the behavior of the second authentication algorithm we presented so that knowing the number of logins ahead of time is unnecessary. The main idea for ZK proofs for authentication is that no party will learn anything more than that you are actually who you say you are.

Suppose you have some secret witness for some NP problem. You want to reveal that you know a witness but do not want to disclose any information other than that you know a witness. This idea can be formalized in the structure of an interactive proof system:

Definition 3 (Zero-knowledge)**.** *An IPS (P, V) for a language L is zero-knowledge if ($\forall V'$ in randomized polynomial time)($\exists$ a simulator $S_{V'}$ in randomized polynomial time) such that ($\forall x \in L$) the distribution representing the view of V' in the interaction of $(V' \leftrightarrow P)(x)$ and $S_{V'}(x)$ are "similar".*

Definition 4 (Similarity)**.** *Similarity is a measure of how close two distributions are. There are several possible ways to define similarity:*

1. *Identical - The distribution are equal. This definition gives us perfect zero-knowledge (PZK) proofs.*
2. *Statistically Close - This is a quantitative measure of how close the distributions are with respect to the 1-norm. This leads to statistical zero-knowledge (SZK) proofs.*
3. *Computationally Indistinguishable - A polynomial time circuit cannot distinguish between the two distributions with more than $\frac{1}{\text{poly}}$ probability. This gives us computational zero-knowledge(CZK) proofs.*

The idea behind ZK proofs is that the prover is the party trying to get the verifier to believe the prover's identity, without disclosing too much to the verifier.

One example of a ZK proof is an IPS for graph isomorphism. $L = \text{GI}$ and the input is (G_0, G_1). Since $\text{GI} \in \text{NP}$ there is a trivial IPS for GI which produces a permutation π, such that $\pi(G_0) = G_1$. However, this is clearly not ZK because the prover gives away the witness. One protocol which is ZK is as follows:

1. P: Pick $a \in_R \{0,1\}, \pi \in_R S_n$. Send $H = \pi(G_a)$ to V.
2. V: Pick $b \in_R \{0,1\}$. Send b to P.
3. P: Send to V a σ such that $H = \sigma(G_b)$.
4. V: Verify that $H = \sigma(G_b)$.

Completeness of this system is clear because if $G_0 \cong G_1$, the prover can aways prove it. If $G_0 \ncong G_1$, V catches the cheating prover with probability one half. Thus this is a IP for GI.

This IPS is also ZK since we can construct a simulator, $S_{V'}$, that produces a distribution which is indistinguishable from the actual distribution. $S_{V'}$ wishes to output (H, b, σ) with the same distribution as V' sees during the protocol with the actual prover.

- Pick a, π, H as in the original protocol.
- Run $V'(H)$ to determine b.
- If $(a = b)$ output (H, b, π) otherwise retry.

If $G_0 \cong G_1$, the probability that $a = b$ is exactly $\frac{1}{2}$, since the distribution of H is independent of a, so b must be chosen independently of a as well. This algorithm runs in expected polynomial time (exactly two runs are expected). PZK assumes expected polynomial time just like ZPP, since the simulator may fail many times to output anything. We can get a SZK proof system by stopping the simulator after a polynomial number of retries.

3.1 A ZK Protocol for 3-Coloring

If GI $\in$ P this protocol is useless since the verifier could solve the problem on its own. We would rather use an NP-complete problem than an NP-intermediate problem as the basis for the protocol since our belief of hardness is stronger for these complete problems.

In a similar fashion we can construct a ZK protocol using graph 3-coloring as the underlying problem. In this case the prover P knows some valid 3-coloring of the vertices, $\gamma : \textit{vertices} \to \{R, G, B\}$.

1. P: Pick a random permutation π for the colors $\{R, G, B\}$. For each vertex v, put in a lockable box the value $\pi(\gamma(v))$. Send all n boxes to V.
2. V: Pick an edge $e = (u, v) \in_R E$, send e to P.
3. P: If $e = (u, v) \in E$ send the keys for u and v to V, otherwise abort.
4. V: Accept if the colors in the two boxes are different.

This is an IPS because if P knows a 3-coloring, V will always verify that on any edge the endpoints are colored differently. If G is not 3-colorable, no matter what P does there will be at least one edge that is wrong giving V a $\frac{1}{\text{poly}}$ chance to catch a cheating prover.

This proof is ZK. The boxes that the verifier sees are indistinguishable without any keys. Thus the choice a V' makes for e is independent of what P put in the boxes. The distribution of the colors that V' sees after unlocking two boxes is uniform. So V' can simulate this interaction himself if G is 3-colorable. This scheme is PZK, however it is not clear that the lock boxes can be implemented computationally. We can translate these lock boxes into the computational realm by applying our bit commitment protocol. Using the bit commitment protocol gives a SZK protocol. This shows if one-way permutations exist there is a SZK protocol for all problems in NP (reduce each problem to 3-coloring).

4 Next Time

Next lecture we will discuss computational learning theory.

CS 810: Complexity Theory 4/16/2007

Lecture 28: Computational Learning Theory

Instructor: Dieter van Melkebeek Scribe: Jake Rosin

Today's lecture discusses computational learning theory. This topic is expansive enough to fill an entire course, so this lecture will provide only an overview.

In this setting the goal is to learn some process through observation, generate a model for that process, and use the model to predict the process's future behavior. An example of this is a spam filter tuned by a user, which attempts to learn the process by which a human distinguishes between spam and legitimate mail.

We give a general algorithm for learning under certain conditions, and discuss decision lists as an application of that algorithm. We also use harmonic analysis to construct a learner which functions in a more restricted setting, and give example processes which it can learn.

1 Concept Learning

Our focus will be on concept learning: learning a certain predicate. Continuing our example from above, our filter constructs a predicate for whether a message is considered spam. A concept in the Boolean domain maps inputs to a single bit of output: $c : \{0,1\}^n \longrightarrow \{0,1\}$.

We need certain ingredients to form a learning problem.

1. A *concept class* C representing a set of concepts. C is the set of possible functions to be learned, with $c \in C$. Often the class will be partitioned and indexed by n and sometimes s.

   Definition 1. *$C_{n,s}$ represents the partition of concept class C for inputs of size n with implementations size-bounded by s (for example, predicates computed by circuits with size $\leq s$).*

   One variant of concept learning is agnostic learning, where there is no underlying concept class (or equivalently where C is the set of all predicates).

2. The process by which the learning algorithm learns the model. It is often assumed that the learner is a passive observer. In this context the learner makes a series of queries to a *sample oracle*; for each query the oracle selects an example at random from the domain according to some distribution D, which may or may not be known to the learner. In the case that D is unknown this is called "distribution-free" learning; in either case a general learning algorithm should work for any D.

   The examples are given to the learner either labeled or unlabeled. Unlabeled samples provide information about the distribution.

   In some cases the learner must make *membership queries*. These queries allow the learner to choose the x to be sampled; equivalently the learner asks an oracle for the label of x.

   Equivalence queries may also be used. Once the learner forms a hypothesis about the concept being learned it forms an equivalence query; the query is answered with either "yes" or a counterexample which the hypothesis misclassifies.

3. A hypothesis class H represents the class of predicates which the learning algorithm may output. It may be partitioned and indexed as $H_{n,s}$ as with a concept class. Proper learning is a variant of concept learning where $H = C$, which is ideal, but not always the case. Another variant is prediction learning, where H is the set of all predicates. In this context whether the learned predicate matches the concept c is irrelevant if it can accurately predict c's behavior. These two variants represent opposing extremes: often $C \subsetneq H \subsetneq \{\text{all predicates}\}$.

4. We measure the success of a learning algorithm by its *error*.

   Definition 2. *The error ε of a learner is* $\Pr_{x \leftarrow D}[h(x) \neq c(x)]$.

   When we require $\varepsilon = 0$ this is called exact learning. Achieving this goal in complex settings usually requires equivalence queries. Another measure of success is the number of errors made by the learner. During each step of the learning process the learner has some hypothesis h with which it can predict the next sample from D. Minimizing the number of times h mispredicts the next sample is the goal.

5. The final ingredient is the complexity measures used. An information-theoretic measure is the number of samples which must be used before the learner's error falls below a given ε. A complexity-theoretic measure is the running time of the learner, based both on the number of samples and the efficiency of the processing of the samples.

2 Distribution-Free Learning

Definition 3. *A concept class C is PAC-learnable (probabilistically approximately correct) by some hypothesis class H if there exists an algorithm $L(n, s, \delta, \varepsilon)$ such that $(\forall n, s, \delta > 0, \varepsilon > 0)(\forall c \in C_{n,s})(\forall D$ on $\{0,1\}^n)$ L outputs with probability $\geq 1 - \delta$ a hypothesis $h \in H_{n,s}$ such that the error $\leq \varepsilon$ where L gets labeled samples from D. This process is efficient if the running time of L is poly$(n, s, \frac{1}{\varepsilon}, \frac{1}{\delta})$.*

2.1 An Example: Decision Lists

Decisions lists are a concept class which is PAC-learnable. For a given input x, a decision list queries a particular bit x_i, then either outputs 0 or 1 or queries another bit. This process may repeat no more than $|x|$ times. An example decision list is given in Figure 1.

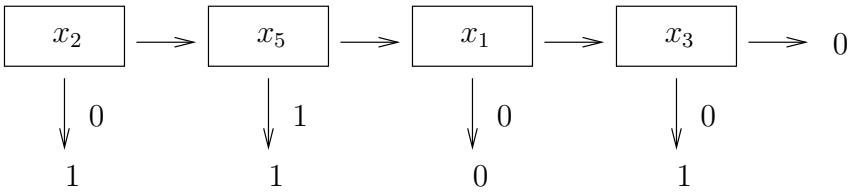


Figure 1: A decision list which queries at most 4 bits of the input.

The length of a decision list is bounded by the number of bits in the input. Each non-terminal step in the list may end the computation on either a 0 or 1, and then output either a 0 or 1. The

final step may output the value of the bit being queried, or output its complement. Bits may be queried in any order. This results in $|C_n| \leq n!4^n$.

Our proper learning algorithm L is simple. It gets m samples from the oracle, then outputs a decision list that is consistent with those samples. The problem of constructing a decision list consistent with some fixed set of samples is left as an exercise; instead we find a bound for m.

Theorem 1. *Decision lists are PAC-learnable in m sample queries, with $m = poly(n, \frac{1}{\varepsilon}, \log \frac{1}{\delta})$.*

Proof. We want to ensure that bad hypotheses are output with low probability. A bad hypothesis is an $h' \in H_n$ such that $\Pr_{x \leftarrow D}[c(x) \neq h(x)] \geq \varepsilon$. We fix h to be one such bad hypothesis.

$$\Pr[L \text{ outputs this } h] \leq \Pr[\text{all } m \text{ samples are consistent w/ } h] \leq (1-\varepsilon)^m \tag{1}$$

The final term follows from our definition of h. h differs from the concept on any sample $x \leftarrow D$ with probability at least ε; m such samples are chosen independently.

This provides an upper bound for failure (Equation 2). We want this value to be at most δ.

$$\Pr[L \text{ fails}] \leq |H_n| \cdot (1-\varepsilon)^m \leq \delta \tag{2}$$

It suffices to find $m \geq \frac{1}{\varepsilon}(\log |H_n| + \log \frac{1}{\delta})$. In this case $\log |H_n| = O(n \cdot \log n)$, so $m = \text{poly}(n, \frac{1}{\varepsilon}, \log \frac{1}{\delta})$. Since finding a consistent hypothesis can be done efficiently, this bound on m shows that L is an efficient PAC-learning algorithm.

□

We point out that the algorithm and analysis above can be applied to a number of settings. A similar algorithm and analysis works for conjunctions, disjunctions, k-CNF, k-DNF, and k-decision lists for any constant k.

2.2 VC-dimension

In this section, we define a quantity that will allow us to improve the number of samples required by the above learning algorithm. We will also be able to use the ideas of this subsection to later give a general learning algorithm.

Definition 4. *$P_{H_n}(k)$ is the maximum over all sets of unlabeled samples $(\xi_1, \ldots, \xi_k)$ of the number of different vectors of the form $(h(\xi_1), \ldots, h(\xi_k))$ with $h \in H_n$.*

In other words, $P_{H_n}(k)$ is the number of hs which form distinct characteristic vectors over a set of k samples, maximized over all such sets. Using this we define the VC-dimension [1] of hypothesis class H_n:

$$\text{VC-dim}(H_n) = \max_m (P_{H_n}(m) = 2^m) \tag{3}$$

In words, this quantity is the largest m for which all possible characteristic vectors of length m may be realized by choosing the appropriate hypotheses from H_n. This quantity may be small even if $|H_n|$ is large. As an example we consider perceptrons.

A perceptron is a linear threshold function within d-dimensional real space, outputting a 1 if some linear combination of the inputs is at least some threshold value, and 0 otherwise: $\sum_{i=1}^{d} a_i x_i \geq$

[1] VC stands for Vapnik-Chervonenkis, so the letters VC have no particular meaning.

t. The VC-dimension of this class is the maximum number of points for which all possible settings may by classified appropriately. For perceptrons VC-dim$(H_n) = d + 1$; see Figure 2 for details for the case $d = 2$. As we will see a bit later, this means that while in principle the number of patterns from k samples is bounded by 2^k, in fact this bound is a polynomial in k of degree $d + 1$.

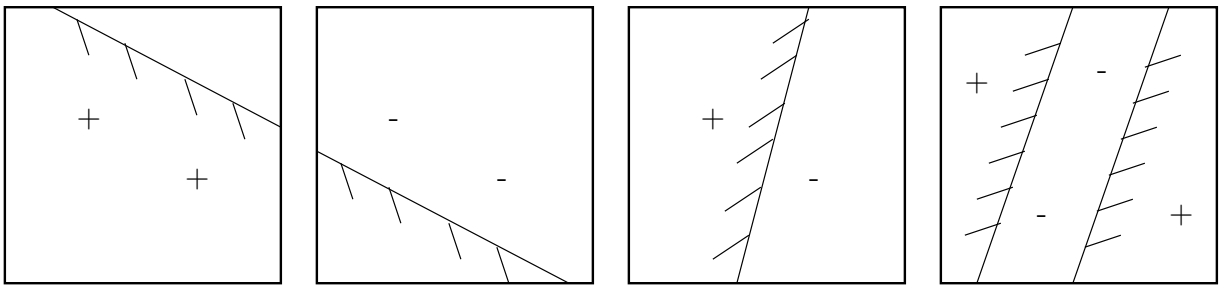


Figure 2: For $m = 2$, any possible setting may be realized by a single linear boundary. Convince yourself that this is true for $m = 3$. For $m = 4$, the setting on the right cannot be classified by a single line (we leave this as an exercise). Thus VC-dim(perceptrons in $\mathbb{R}^2$)=3.

2.3 Improving PAC-Learning for Decision Lists

We showed in subsection 2.1 that using $m = \frac{1}{\varepsilon}(\log |H_n| + \log \frac{1}{\delta})$ is sufficient for PAC-learning the concept class of decision lists. This bound comes from Equation 2, where the $|H_n|$ term is present as an upper bound on the number of distinct h that could be output using m samples. We can lower the number of samples needed by getting a tighter bound for the latter quantity. It turns out that this quantity is upper bounded by $P_{H_n}(2m)$, a fact we do not prove here. Given this unproven fact, our goal is to upper bound $P_{H_n}(2m)$, and then we can replace the $|H_n|$ term above with this value.

Obviously the maximum possible value of $P_{H_n}(2m)$ is 2^{2m}, but depending on the hypothesis class the value may be much smaller. The argument that $P_{H_n}(2m)$ bounds the number of nonequivalent hypotheses over a sample size of m is non-trivial, but it should be obvious that this number is an improvement over $|H_n|$. $P_{H_n}(2m)$ can be bounded by the VC-dimension of the hypothesis class, VC-dim(H_n) defined above. Specifically

$$P_{H_n}(k) \leq \sum_{c=0}^{d} \binom{k}{c} \leq \left(\frac{ek}{d}\right)^d \text{ where } d = \text{VC-dim}(H_n). \tag{4}$$

The log of this term is $d \log k$, which replaces $\log |H_n|$ in the value of m above.

2.4 A General PAC-Learning Algorithm

Given the analysis above, we can give a general PAC-learning algorithm. Recall that the learning algorithm for decision lists was to obtain a number of samples and output a hypothesis consistent with those samples. The same analysis given there applied to the general case shows that $m = O(\frac{1}{\varepsilon}(\text{VC-dim}(H_n) \log \frac{1}{\varepsilon} + \log \frac{1}{\delta}))$ samples suffice. The general PAC-learning algorithm, then, is as follows.

1. Query the sample oracle m times.

2. Output a hypothesis $h \in H_n$ consistent with those samples.

At the same time, a lower bound, which we do not prove here, is known which shows that $\Omega(\frac{1}{\varepsilon}(\text{VC-dim}(H_n) + \log\frac{1}{\delta})$ samples are needed for this algorithm. This shows that the general algorithm given above is essentially tight with respect to the number of samples needed.

We point out that this PAC-learning algorithm assumes that finding a consistent hypothesis can be done efficiently. This may not always be the case, as described in the next section.

2.5 Complexity of the Consistency Problem

Up to now we have argued from an information-theoretic perspective. Complexity-theoretic arguments deal with step 2 above - finding a consistent hypothesis.

If P = NP and H is polytime computable, the consistency problem is trivial (guess the hypothesis and verify that it is consistent). We can also show that for some simple examples the consistency problem can be NP-hard, depending on the choice of the hypothesis class. Finding a consistent hypothesis for $H = C = \{$DNF formulas with ≤ 3 clauses (i.e., disjunctions of at most three conjunctions, with the conjunctions being unbounded in size)$\}$ is NP-hard, for example. Using distributivity such a formula can be expanded into 3-CNF form; the relaxed version with $H = \{$3-CNF formulas$\}$ is easy. The important point here is that for some concept classes proper-learning is difficult, but learning the same concept class with a somewhat larger hypothesis class H can be much easier.

As mentioned last lecture, if one-way functions exist then we can't efficiently PAC-learn $C_{n,s} = \{$circuits of size $\leq s\}$ by any H. This follows from the fact that if one-way functions exist we can construct pseudorandom bit generators, and pseudorandom function generators. Given a pseudorandom function generator as the concept c, the learner must construct a hypothesis which predicts the output of c on the next sample. By the definition of pseudorandom generators this cannot be done by computationally limited processes.

2.6 Notes on PAC-learning

- In our definition we required our algorithm to work for any choice of (δ, ε). As we saw with randomized algorithms we can relax restrictions on error rate and boost the result through multiple applications of the algorithm. The same is true for PAC-learning; given the following settings of δ and ε we can reach any arbitrary degree of confidence and accuracy.

$$\delta = 1 - \frac{1}{poly(n)} \tag{5}$$

$$\varepsilon = \frac{1}{2} - \frac{1}{poly(n)} \tag{6}$$

  Confidence can be boosted in a similar way that was used for error reduction with randomized algorithms: by generating a large number of hypotheses h, then testing each on samples taken from the distribution, estimating the accuracy of each using the Chernoff bound, and choosing the best.

- Reducing the error of the algorithm does not follow from a simple use of the techniques for error reduction of randomized algorithms, but with more work we can also reduce the error of the learning algorithm. The key intuition is that our PAC-learning algorithm is required to perform correctly for all distributions D, not just the one being learned. We can exploit this by taking our hypothesis h and re-weighting the distribution to place equal weight on the inputs for which h performs correctly and those for which it fails. Running the PAC-algorithm on this new distribution will provide new information. After enough iterations of this procedure we output a weighted majority vote of all the hypotheses. Arguing that this boosting procedure works is non-trivial.

- In an agnostic learning setting (one with no underlying C) there is no guarantee that the concept c exists within our hypothesis class H. The best we can hope for is to get a hypothesis as close to c as possible within H, with some margin of error. We look for error at most ε plus the minimum distance between H and c.

  The generalized PAC-learning algorithm given in subsection 2.2 may be used in this setting with one modification: rather than output h consistent with all m samples, we output an h which is as consistent as possible given H. Finding this consistent hypothesis becomes much more complicated, even for simple problems. For example with $H = \{\text{conjunctions}\}$ the consistency problem is NP-hard.

- One unrealistic assumption made by our PAC algorithm is that the samples received from the oracle are completely error-free. Errors could occur on the labels or the inputs themselves (which may be inconsistent with D), and may occur due to noise, or maliciousness on the part of some attacker. The simplest error-aware model to which PAC-learning work has been extended is one which allows random classification noise. Formally we assume that for every possible input, the label is flipped with probability $\zeta < \frac{1}{2}$ independently for each input. Clearly the closer ζ is to $\frac{1}{2}$ the more difficult the learning problem becomes. The running time of an efficient PAC-learner becomes $\text{poly}(n, s, \frac{1}{\varepsilon}, \frac{1}{\delta}, \frac{1}{\frac{1}{2}-\zeta})$

  Statistical query algorithms are a class of algorithms which work in this setting. Rather than query a sample oracle, statistical query algorithms ask the teacher for an approximation for the probability that a certain predicate holds with respect to D. For example, one valid query asks for the probability over D that the label is the parity of the input bits.

  Many learning algorithms can be cast in this framework; any algorithm which can will be robust in a setting with random classification noise.

3 Learning w.r.t. the Uniform Distribution

PAC-learning algorithms function under any possible distribution D. For some problems this is unrealistic, but algorithms can be devised which function with respect to some fixed distribution. The uniform distribution is a natural choice. In this context harmonic analysis turns out to be useful for learning concepts for which the Fourier transform is concentrated on very few coefficients. Recall that by Parseval's equality, the sum of the squares of the coefficients of the Fourier transform of a Boolean function is 1. The 2^n coefficients may have uniform values, but there could be a few large ones. In the latter case there exists a learning algorithm with respect to the uniform distribution.

In particular this condition holds for decision trees, as well as for constant-depth circuits. The latter assertion corresponds to the proof from earlier in the course that constant-depth circuits can be approximated well by multivariate polynomials of low degree. The Fourier transform corresponds to expressing the function as a linear combination of characters which are essentially parity functions over certain subsets. If bit settings are in $\{\pm 1\}$ rather than $\{0,1\}$ these parity functions become products. This means that constant-depth circuits can be approximated very well by linear combinations of characters that correspond to small subsets.

We now see how we can efficiently learn with respect to the uniform distribution in such a case using membership queries.

3.1 Learning Algorithm Using List Decoding

Suppose we have a concept class C which has its power spectrum concentrated over few coefficients. We can use the list decoding algorithm for the Hadamard code as a component in a learning algorithm for C. We view the concept c as a function $f : \{\pm 1\}^n \to \{\pm 1\}$. Recall the properties of the list decoding procedure for the Hadamard code when applied to the characteristic sequence of f: in $\text{poly}(n, 1/\epsilon)$ time, we produce a list of all information words with Hadamard encoding within distance $1/2 - \epsilon$ of f. By the fact that the characteristic sequence of χ_g is equal to the Hadamard encoding of g, we have a list of characters χ_g that with high probability includes all those such that $\Pr_x[\chi_g(x) = f(x)] \geq 1/2 + \epsilon$. Plugging in the equality $\hat{f}(g) = 2\Pr[f(x) = \chi_g(x)] - 1$, we convert this inequality to a bound on the size of the Fourier coefficients, concluding that we have a list of characters that includes all those s.t. $\hat{f}(g) \geq 2\epsilon$.

We conclude that given some threshold τ, all Fourier coefficients of absolute value $\geq \tau$ can be found in time $\text{poly}(n, \frac{1}{\tau})$. We point out that using the list-decoding algorithm to determine the coefficient with large weight requires evaluating the received word at specific points of our choosing. For the learning algorithm, this means we will need membership queries. We now give the learning algorithm for f using membership queries.

1. Generate L, a list including indices of all Fourier coefficients with absolute value at least τ using the list-decoding algorithm for the Hadamard code, with the characteristic sequence of f as the "received word" for the list-decoding.

2. For each $y \in L$, estimate $\hat{f}(y) = 2\Pr_x[f(x) = \chi_y(x)] - 1$ by picking a polynomial number of points at random. Let α_y be the approximation of $\hat{f}(y)$. Ensure this approximation is within η of the true value with high probability.

3. Recall that
$$f(x) = \sum_y \hat{f}(y)\chi_y(x) \tag{7}$$
We define an approximation of f as
$$g(x) = \sum_{y \in L} \alpha_y \chi_y(x).$$
We consider g an approximation of f; we have dropped (some of) the small coefficients of f and approximated the other ones. This will be a good approximation provided most of the mass is concentrated on a few large coefficients, as we have assumed.

4. Output $h(x) = \text{sign}(g(x))$. Note that g is not necessarily a Boolean function, as it works over the set of real numbers, but h is.

What remains is finding a reasonable setting for the parameter τ, and showing that this algorithm will succeed with probability within a certain degree. We know that the complexity of this algorithm depends on τ, as τ determines the running time of the algorithm, which implicitly gives an upper bound on the length of the list L.

Suppose for some x, $h(x) \neq f(x)$. In this case, the difference between $g(x)$ and $f(x)$ is at least one. We square this to get the absolute value and apply Markov's inequality to get

$$\Pr[h(x) \neq f(x)] \leq \Pr[(g(x) - f(x))^2 \geq 1] \leq E_x[(g(x) - f(x))^2]. \tag{8}$$

Now we use Parseval's equality and get

$$\Pr[h(x) \neq f(x)] \leq E_x[(g(x) - f(x))^2] = \sum_y \widehat{(g(y) - f(y))}^2 = \sum_y (\hat{g}(y) - \hat{f}(y))^2 \tag{9}$$

where the last equality is because of linearity of the Fourier transform. We divide the sum over all ys between those in L and those not. For those in L the coefficient is α_y, a good approximation to $\hat{f}(y)$ to within η. For any $y \notin L$, $\hat{g}(y)$ is zero. This gives

$$\Pr[h(x) \neq f(x)] \leq \eta^2 |L| + \sum_{y \notin L} (\hat{f}(y))^2. \tag{10}$$

This depends only on f, and by exploiting the fact that the power spectrum is concentrated we can make that term small by setting τ appropriately.

3.2 Decision Trees

Consider decision trees as a specific example. We will need a result relating the Fourier spectrum of a function f and its representation as a decision tree.

Exercise 1. *If f is computed by a decision tree T of size s, then $\sum_y |\hat{f}(y)| \leq$ #leaves of $T \leq s$.*

Now suppose we apply the learning algorithm using list-decoding described above. We first want to bound the error term of $\sum_{y \notin L} (\hat{f}(y))^2$. Since we know that $|\hat{f}(y)| \leq \tau$ for all $y \notin L$, the exercise implies the following:

$$\sum_{y \notin L} (\hat{f}(y))^2 \leq \tau \cdot \sum_{y \notin L} |\hat{f}(y)| \leq \tau \cdot s. \tag{11}$$

To have error at most ε, we want to set τ and η so that $\tau \cdot s \leq \frac{\varepsilon}{2}$ and $\eta^2 |L| \leq \frac{\varepsilon}{2}$. We achieve the former by setting $\tau = \frac{\varepsilon}{2s}$. From this we know that $|L|$ is polynomial in n, s and $\frac{1}{\varepsilon}$. To ensure the latter, we set $\eta = \sqrt{\frac{\varepsilon}{2|L|}}$. Recall that η is the maximum error we want to allow on the approximations we calculated for the Fourier coefficients. Using a Chernoff bound, we can achieve $\eta = \sqrt{\frac{\varepsilon}{2|L|}}$ with $\text{poly}(|L|, \frac{1}{\varepsilon})$ samples.

We conclude that the algorithm, which uses membership queries to run the list-decoding algorithm, has error at most ε and runs in $\text{poly}(n, s, \frac{1}{\varepsilon})$ time.

3.3 Constant-depth Circuits

Constant-depth circuits also have the property that their Fourier spectrum is concentrated on a small number of coefficients. This will allow us to make use of the above analysis to give a learning algorithm for constant-depth circuits. The key difference is that we will not need to use the list-decoding algorithm to generate the list containing large coefficients, so the learning algorithm will only require samples and not membership queries. The intuition is that because constant-depth circuits cannot even approximate parity, the characters with large Hamming weight must have small coefficients (otherwise, the function would have high agreement with parity over the bits indexed by that character). The list of coefficients with large weight, then, will just be the list of coefficients corresponding to characters with small Hamming weight.

We now give the analysis. Let f be a Boolean function on n variables computed by a depth d unbounded fanin circuit of size s. In the lecture on constant-depth circuits, we constructed a low-degree approximation of f as a step towards proving circuit lower bounds for parity. In particular, we constructed a Boolean function g' computed by a polynomial over $GF(3)$ of degree $\Delta \leq (2t)^d$ such that

$$\Pr[f(x) \neq g'(x)] \leq \frac{s}{3^t}.$$

This implies that

$$E_x[(f(x) - g'(x))^2] \leq 4\frac{s}{3^t}, \tag{12}$$

which is in a form that is useful for applying Fourier analysis. But to use Fourier analysis, we want a polynomial over $\mathbb{R}$ rather than $GF(3)$. The polynomial g' was constructed using the approximation method. Using the switching lemma method instead, g' can be constructed with the same properties mentioned above but over $\mathbb{R}$. Let L be the set of binary y's with Hamming weight $\Delta \leq (2t)^d$. This is our list of characters with large coefficients, so we want to show that characters outside of L have small coefficients. We have the following

$$\sum_{y \notin L} (\hat{f}(y))^2 = \sum_{y \notin L} (\hat{f}(y) - \widehat{g'}(y))^2 \leq \sum_{y} (\hat{f}(y) - \widehat{g'}(y))^2, \tag{13}$$

where $\widehat{g'}(y) = 0$ for $y \notin L$ because g' can be expressed as a polynomial over $\mathbb{R}$ of degree at most $(2t)^d$. Now (13) is in a form we have seen earlier, so using (12) we get

$$\sum_{y} (\hat{f}(y) - \widehat{g'}(y))^2 = E_x[(f(x) - g'(x))] \leq \frac{4s}{3^t}.$$

We ensure this is at most $\frac{\varepsilon}{2}$ by setting $t = \Omega(\log \frac{s}{\varepsilon})$. Setting $\eta = \sqrt{\frac{\varepsilon}{2|L|}}$ implies that the RHS of (10) is at most ε.

We have given the analysis, so we recap the learning algorithm and see how efficient it is. We set $t = \Theta(\log \frac{s}{\varepsilon})$ and let L be the set of characters with Hamming weight at most $\Delta = (2t)^d = \Theta((\log \frac{s}{\varepsilon})^d)$. Once we have L, we proceed from the second step of the learning algorithm in section 3.1. The analysis there and here shows that we get an approximation with error at most ε. The running time and number of samples needed by the algorithm are $\text{poly}(|L|, n, \frac{1}{\varepsilon})$. As $|L| = \sum_{d=0}^{\Delta} \binom{n}{d} \approx n^{\Delta}$, the running time of the learning algorithm is quasi-polynomial for polynomial size constant-depth circuits. However, recall that the algorithm we have given only requires samples and not membership queries.

For the specific case of depth $d = 2$, meaning circuits of DNF or CNF form, there does exist a polynomial time algorithm for learning with respect to the uniform distribution which uses membership queries and Fourier analysis.

4 Next Lecture

In the next lecture we discuss query complexity.

CS 810: Complexity Theory 4/18/2007

Lecture 29: Query Complexity

Instructor: Dieter van Melkebeek Scribe: Jeff Kinne

The topic of today's lecture is query complexity. Suppose we have access to a database and wish to evaluate a predicate over the information contained in the database. The goal is to evaluate the predicate with as few queries to the database as possible. We formalize the problem and examine the query complexity of three models: deterministic, randomized, and quantum. We show that for fully specified problems the query complexities of all three models are within polynomial factors of each other; but for promise problems we give examples that demonstrate exponential gaps in the complexities for the three models.

1 Query Complexity Models

We first formalize the intuitive notion of a database described above. We let our database be specified by a bit string $x \in \{0,1\}^n$, and let $f : \{0,1\}^n \to \{0,1\}$ be a predicate we wish to evaluate on x with as few accesses to x as possible. Most natural functions that depend on all the input bits (majority, parity, and, or, etc.) require full query complexity on deterministic machines. However, database queries are often very simple and only depend on few input bits. Also, some interesting functions have smaller query complexity in alternate models (see below).

We will look at the query complexity on three different models.

1. **Deterministic**. As we are only interested in the queries that are generated and not the complexity of the computation, we view the evaluation of f in this model as a *decision tree*. The root of the decision tree is the first bit of x queried, say x_i. The child on its left branch is the next bit queried if $x_i = 0$, and the child on the right branch for $x_i = 1$. The leaves of the tree correspond to our outputting a value for f. Notice that the *depth* of the decision tree measures the maximum number of queries that are asked for any string $x \in \{0,1\}^n$.

   We let $D(f)$ denote the deterministic query complexity of f - the minimum depth of a decision tree deciding f.

2. **Randomized**. We use the two-sided bounded-error setting. We would like to measure the worst-case number of queries that the machine asks over all inputs and random strings. The computation of the machine can be viewed as first picking a random string and then performing a deterministic evaluation. In other words, we pick a random string, and that random string determines a decision tree to use to evaluate the function. Thus we view the computation as a probability distribution over deterministic decision trees, with the worst-case query complexity corresponding to the maximum depth of any tree in the support of the distribution of trees.

   We let $R(f)$ denote the randomized query complexity of f - the minimum depth d such that there exists a probability distribution on decision trees of depth at most d that decides f correctly on every input with probability $> 2/3$.

3. **Quantum**. In the quantum setting, recall that we in general have access to unitary operations that act on superpositions of bits. Thus we provide a unitary operation U_x for accessing the input. We define U_x as $U_x : |i, b, z\rangle \rightarrow |i, b \oplus x_i, z\rangle$, where i is the index to the bit of x we want, b is an auxiliary bit for XORing the desired bit into the output, and z contains the remaining quantum bits we are working with. For quantum machines, we measure the query complexity as the number of times we need to apply U_x. As with the randomized case, we require that the quantum process computes the correct value on each input with probability $> 2/3$. Without loss of generality, we designate an output bit, which we measure at the end of the quantum computation to determine the output.

   We let $Q(f)$ denote the quantum query complexity of f - the minimum number of queries to U_x by a bounded-error quantum process solving f.

2 Examples

2.1 OR

$D(OR) = n$: given a decision tree of depth $n-1$, we can construct an x such that the decision tree gives the wrong value.

$R(OR) = \Theta(n)$. This can be argued by considering a randomized process deciding OR as a probability distribution over decision trees. Suppose $R(OR) = k$, that is that there is a probability distribution over depth at most k decision trees that decides f on each input with probability $> 2/3$. An averaging argument shows that there is at least one $i \in \{1, 2, ..., n\}$ such that $\Pr[i^{th}$ bit of input queried$] \leq \frac{k}{n}$ when using a decision tree from the distribution. Consider the strings 0^n and $0^{i-1}10^{n-i}$. Because $OR(0^n) = 0$ and $OR(0^{i-1}i0^{n-i}) = 1$, and we have assumed the distribution of trees solves OR with probability $> 2/3$, we conclude that

$$\Pr[\text{output different on } 0^n \text{ and } 0^{i-1}10^{n-i}] \geq 1/3.$$

But because the i^{th} bit is queried with probability at most $\frac{k}{n}$, $\Pr[\text{output different}] \leq \frac{k}{n}$. We conclude that $k \geq n/3$.

Alternatively, we can argue $R(OR) = \Theta(n)$ using the *Min-Max principle* described below.

2.1.1 Min-Max principle

The *Min-Max principle* can be used to argue lower bounds on R for many functions. To determine the query complexity of f, we think of playing a game between two players:

I) Pick a decision tree A of depth $\leq k$,
II) Pick an input x.

f has query complexity $\leq k$ if there is a randomized strategy for player I that results in $A(x) = f(x)$ with probability $\geq 2/3$. To formalize this, we introduce some notation. Let $M_{A,x} = \chi[A(x) \neq f(x)]$ be a matrix indicating for each decision tree of depth $\leq k$ which inputs it is incorrect on. Player I's strategy can be modeled as a probability distribution p over decision trees of depth $\leq k$. Then player I has a winning strategy (and thus $R(f) \leq k$) iff

$$\min_p \max_x (p^T M e_x) < 1/3.$$

Some manipulations give alternate formulations of this equation. We can also allow player II to have a randomized strategy; let q be the distribution over inputs x that player II chooses. Then

$$\min_p \max_x (p^T M e_x) = \min_p \max_q (p^T M q)$$

because e_x is a particular q and $p^T M q$ is a convex combination of the $p^T M e_x$. For any fixed q',

$$\min_p \max_q (p^T M q) \geq \min_p (p^T M q') = \min_A (e_A^T M q')$$

for similar reasons.

We can thus show that $R(f) > k$ by demonstrating a distribution q with $\min_A (e_A^T M q) \geq 1/3$. That is, we need to demonstrate a distribution of inputs so that all decision trees of depth at most k fail on this distribution with high probability.

We note that in fact an application of linear programming duality shows that $\min_p \max_q (p^T M q) = \max_q \min_p (p^T M q)$, which can be used to give tight bounds on $R(f)$ by picking q appropriately.

2.1.2 OR (continued)

We use the Min-Max principle on the function $f = OR$. Consider the distribution

$$q = \begin{cases} 1/3 \text{ on } 0^n, \\ 2/(3n) \text{ on } 0^{i-1}10^{n-i} \text{ for } 1 \leq i \leq n. \end{cases}$$

If $R(OR) = k$, then by the Min-Max principle, no matter which decision tree A of depth $\leq k$ we pick, A fails on this distribution with probability less than $1/3$. Consider the all 0's path through A. Because $1/3$ of the weight is on 0^n and A is supposed to fail with probability less than $1/3$, the answer on the all 0's path cannot be 1. As the depth of A is at most k, at least $\frac{2}{3}(1 - \frac{k}{n})$ of the weight in the distribution is on strings whose true output is 1 but will go down the all 0's path of A. Then $\frac{2}{3}(1 - \frac{k}{n}) < \frac{1}{3}$, and we conclude that $k > \frac{n}{2}$.

We note that $Q(OR) = \Theta(\sqrt{n})$. This follows by Grover's search algorithm, a celebrated quantum algorithm which we do not cover in this course.

2.2 Hadamard Code Property Testing

Let f be the function that tests whether a string is either a Hadamard codeword or is far away from one. We point out that this is a promise problem. Recall that the linearity test used with PCPs for the Hadamard Code only required three queries (namely $x, y, x+y$). We conclude that $R(f) = \Theta(1)$. As a quantum process can simulate a random process, we also conclude that $Q(f) = \Theta(1)$. It can be shown that $D(f) = \Theta(n)$ by using an adversarial argument. Thus this function demonstrates a promise problem with an exponential gap between deterministic and randomized query complexity.

2.3 Simon's Problem

We demonstrate a promise problem with an exponential gap between randomized and quantum query complexity. Recall Simon's Problem: x is the characteristic sequence of $g : \{0,1\}^\ell \to \{0,1\}^\ell$ with the promise that either g is 1-1 or $\exists s \neq 0$ such that $g(y) = g(z) \Leftrightarrow y = z + s$. In the lecture on quantum effects, we gave a quantum algorithm to find s using ℓ queries. We cast the problem as a promise problem by setting $f(x) = 1$ if g_x is 1-1 and $f(x) = 0$ if there is an s as above. The quantum algorithm gives us $Q(f) = \Theta(\log n)$. It can be shown that $R(f) = \Omega(\sqrt{n})$ using the Min-Max principle and the birthday paradox rule.

3 Main Theorem

We have seen examples of promise problems where there is an exponential gap between the deterministic, randomized, and quantum query complexity. The main theorem of this lectures shows that this can only be true for promise problems. We show that for fully specified functions the query complexities D, R, Q are polynomially related.

Theorem 1. *For any fully specified f, $Q(f) \leq R(f) \leq D(f) \leq O((Q(f))^6)$.*

The first two inequalities follow from the efficient simulations of deterministic processes by randomized processes and of randomized processes by quantum processes. So we must show that $D(f) \leq O((Q(f))^6)$. We point out that the largest known gap in query complexity is the quadratic gap known for OR, so this result may not be tight.

3.1 First Step

We introduce a few related complexity measures. The first step in the proof is to relate $D(f)$ to these new complexity measures with a series of lemmas.

Definition 1. *The* certificate complexity *of f on input x is defined as*

$$W(f;x) = \min_{I \subseteq [n]} \{|I| \mid (\forall y) y|_I = x|_I \Rightarrow f(y) = f(x)\}.$$

In words, $W(f;x)$ is the minimum number of bits of x that need to be checked to uniquely determine the value of $f(x)$.

We also define $W_b(f) = \max_{x|f(x)=b} W(f;x)$ and $W(f) = \max(W_0(f), W_1(f))$.

For the OR function, we have that $W_0(OR) = n$ and $W_1(OR) = 1$. We point out that $D(f) \geq W_b(f)$ for all functions f because each path on a decision tree contributes a certificate for the inputs that end up there. This may not be the best certificate, so $W_b(f)$ can be smaller than $D(f)$.

Definition 2. *The* sensitivity *of f on input x is defined as*

$$S(f;x) = |\{i \in [n] \mid f(x + e_i) \neq f(x)\}|.$$

In words, $S(f;x)$ is the number of bits of x that if changed would change the value of f.

We also define $S_b(f) = \max_{x|f(x)=b} S(f;x)$ and $S(f) = \max(S_0(f), S_1(f))$.

For the OR function, we have that $S_0(OR) = n$ and $S_1(OR) = 1$.

Definition 3. *The* block sensitivity *of f on input x is defined as*

$$BS(f;x) = \textit{Max \# disjoint blocks } B_i \subseteq [n] \textit{ s.t. } (\forall i) f(x + \chi_{B_i}) \neq f(x).$$

In words, $BS(f;x)$ is the maximum number of disjoint subsets of bits of x that if flipped change the value of f.

We also define $BS_b(f) = \max_{x|f(x)=b} BS(f;x)$ and $BS(f) = \max(BS_0(f), BS_1(f))$.

For the OR function, BS and S are the same. There are other functions on which they differ. We point out that because S is a restriction of BS, $S_b(f) \leq BS_b(f)$ for all f and b. Further, $BS_b(f) \leq W_b(f)$ because flipping any bit in a minimal certificate flips the output of the function.

For the measures we have defined, we already have the following inequalities:

$$S_b(f) \leq BS_b(f) \leq W_b(f) \leq D(f).$$

We want to upper-bound $D(f)$ by the smaller measures, which we will do with two lemmas.

Lemma 1. *For all functions f and bits b, $W_b(f) \leq BS_b(f) \cdot S_{\overline{b}}(f)$.*

Proof. Consider any x with $f(x) = b$ and $k = BS_b(f;x)$ with corresponding blocks $B_1, .., B_k$. Without loss of generality, we assume the B_i are minimal sets - meaning that flipping values of x in a subset of any of them does not change the value of f.

We first show that for each i, $|B_i| \leq S_{\overline{b}}(f)$. Consider the input $y = x + \chi_{B_i}$. By the properties of B_i, $f(y) = \overline{b}$. Because B_i is minimal, flipping the bits in a subset of B_i still produces output b. Then $S(f;y) \geq |B_i|$ as flipping any of the bits indexed by B_i in y changes the output back to b again.

Second, we claim that $\cup_{i=1}^{k} B_i$ induces a certificate for $f(x)$. Suppose not. Then there are a set of positions outside of $\cup_{i=1}^{k} B_i$ that can be flipped to flip the output of f, and we would have that $BS_b(f) \geq k + 1$.

Then for x, we have a certificate of size $\sum_{i=1}^{k} |B_i| \leq BS(f;x) \cdot S_{\overline{b}}(f) \leq BS_b(f) \cdot S_{\overline{b}}(f)$. □

The second lemma upper bounds $D(f)$ as a function of the other measures.

Lemma 2. *For any f and b, $D(f) \leq W_b(f) \cdot BS_{\overline{b}}(f)$.*

Proof. We wish to construct a decision tree for f of depth at most $W_b(f) \cdot BS_{\overline{b}}(f)$. We use the following algorithm to determine $f(x)$:

(1) $I \leftarrow \emptyset$, I is set of indices we have queried in x.
(2) **while** $x|_I$ does not force f
(3) Pick a b-certificate consistent with $x|_I$.
(4) Add these indices to the set I of queried bit positions.
(5) Output $f(x)$.

To show that $|I| \leq W_b(f) \cdot BS_{\overline{b}}(f)$, we note that each iteration of the loop queries at most $W_b(f)$ bit positions, and then show that the number of iterations is at most $BS_{\overline{b}}(f)$. We omit this proof. □

By combining the two above lemmas with the fact that $S_b(f) \leq BS_b(f)$, we conclude that

$$D(f) \leq (BS(f))^3.$$

3.2 Second Step - Multi-variate Polynomials

The second main ingredient in the proof of the main theorem is to relate the quantum query complexity of a function f to the degree of a polynomial approximating f.

Definition 4. *$\widetilde{Deg}(f)$ = the minimum degree of a multi-linear n-variate polynomial A over $\mathbb{R}$ s.t.*

$$(\forall x \in \{0,1\}^n) A(x) \in [0,1], \text{ and } |A(x) - f(x)| \leq \frac{1}{3}.$$

Lemma 3. *For any function f, $\widetilde{Deg}(f) \leq 2Q(f)$.*

Proof. Consider a quantum algorithm computing a function f with bounded error which applies U_x $Q(f)$ times. We first show that the amplitudes after k applications of U_x can be written as multi-variate polynomials in $x_1, ..., x_n$ of total degree $\leq k$. The proof is by induction. Let $\alpha'_{|i,b,z\rangle}$ be the amplitude after applying U_x and $\alpha_{|i,b,z\rangle}$ be the amplitude before. By the definition of U_x, we have that

$$\alpha'_{|i,b,z\rangle} = x_i \alpha_{|i,\overline{b},z\rangle} + (1 - x_i)\alpha_{|i,b,z\rangle}.$$

We claim that if we are looking at the k^{th} application of U_x, then each of $\alpha_{|i,\overline{b},z\rangle}$ and $\alpha_{|i,b,z,\rangle}$ are of degree $\leq k-1$. By induction, the amplitudes immediately after applying U_x for the $(k-1)^{th}$ time were of total degree $\leq k-1$. In between applications of U_x there may be some unitary operations not involving the input. As these do not do not involve the input, the amplitudes immediately before the k^{th} application of U_x are still of degree at most k. Then $\alpha_{|i,\overline{b},z\rangle}$ and $\alpha_{|i,b,z\rangle}$ each have total degree at most $k-1$ and $\alpha'_{|i,b,z\rangle}$ has total degree at most k.

The output distribution of the quantum algorithm is the sum of squares of certain $\alpha_{|i,b,z\rangle}$'s. The polynomial approximation we give is the sum of squares of these amplitudes. As these amplitudes are expressible as polynomials of degree at most $Q(f)$ and squaring multiplies the degree by 2, we have a polynomial of degree at most $2Q(f)$. It is linear in each variable x_i because we can replace x_i^2 with x_i since we are looking at boolean variables. The polynomial approximates f because we are looking at a bounded-error quantum algorithm. And finally, the polynomial takes values in $[0,1]$ because it computes a probability. □

3.3 Wrap-up

The final lemma we need to prove the main theorem is the following.

Lemma 4. *For any function f, $BS(f) \leq 4(\widetilde{Deg}(f))^2$.*

Proof sketch. Let $k = BS(f) = BS(f;x)$ with blocks $B_1, B_2, ..., B_k$, $b = f(x)$, and let $\tilde{f}$ be an approximating polynomial for f. The result is proved by making a number of transformations to $\tilde{f}$ based off of the blocks such that: i) the degree with each transformation does not go up, and ii) the final result is a univariate polynomial whose degree can be lower bounded based off of the number of blocks.

For the first step, consider $g : \{0,1\}^k \to \mathbb{R}$ defined by $g(y) = \tilde{f}(x + \sum_{i=1}^k y_i \cdot \chi_{B_i})$. Notice that $deg(g) \leq deg(\tilde{f})$, and that

$$\begin{aligned} &\text{on weight 0 } y\text{'s}, \quad |g(y) - b| < 1/3, \\ &\text{on weight 1 } y\text{'s}, \quad |g(y) - \overline{b}| < 1/3. \end{aligned}$$

Intuitively, g should have high degree to be able to accomplish this, but it is difficult to analyze as it is multi-variate. Notice that the above property only relates to the weight of the input - an input has weight 0 or 1 under any permutation of its bits.

The second step is to convert g into a univariate polynomial h such that $h(w)$ captures the behavior of g on inputs of weight w. Formally, we define h to be the *symmetrization* of g, that is

$$h(w) = \text{Average of } g \text{ over } y\text{'s of weight } w.$$

h is defined by the formula $h(w) = \frac{\sum_{\pi \in S_k} g(\pi(w))}{k!}$. From this, it is clear that $deg(h) \leq deg(g)$. Also notice that h has the following properties: 1) $h(0)$ and $h(1)$ differ by at least $1/3$, and 2) $h(w) \in [0,1]$ for all $w \in \{0,1,...,k\}$. Informally, h must have a large derivative somewhere in the interval $[0,1]$ but at the same time h is small on a number of points in the interval $[0,k]$. Tools from analysis can be used to formalize the intuition that h should have high degree. In particular, it can be shown that $deg(h) \geq \sqrt{\frac{k}{4}}$. Combined with the fact that $deg(h) \leq deg(\tilde{f})$, we get the stated result. □

We get the result of the main theorem by combining the results we have proved:

$$D(f) \leq (BS(f))^3 \leq (4(\widetilde{Deg}(f))^2)^3 \leq 64(Q(f))^6.$$

CS 810: Complexity Theory 4/20/2007

Lecture 30: Communication Complexity

Instructor: Dieter van Melkebeek Scribe: Piramanayagam Arumuga Nainar

In our study of complexity theory in non-standard settings, last time we discussed query complexity - the number of bit positions in the input that must be looked up while evaluating a query. Today we discuss communication complexity which is motivated by distributed computing. In this setting, several parties, each with access to a private input, collaborate with each other to realize some activity. The objective function to be minimized is the number of bits exchanged between the parties during the activity. Our focus is only on the communication aspect and we do not care about the computational complexity.

First, we introduce the formal model in deterministic, randomized and quantum settings. Next, we give the lower bounds for some example tasks and discuss techniques that can be used to establish such bounds. Finally, we demonstrate an application of communication complexity by using it to derive bounds on circuit complexity.

1 Model

In the formal model, we make the usual assumption that we are solving decision problems. We also make the simplifying assumption that there are only two communicating entities: Alice and Bob. Given a function $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ and two strings $x, y \in \{0,1\}^n$, such that Alice has private access to x and Bob to y, Alice and Bob must evaluate $f(x,y)$ by exchanging as few bits as possible.

A solution to the above problem is called a *communication protocol*. It is specific to f and can be visualized as a binary-tree. The interior nodes are labeled A (for Alice) or B (for Bob) that denote the party initiating the communication at that step. The edges are labeled 0 or 1 denoting the value of the exchange. At each node, the corresponding party can decide which bit to send based on its input. The leaves of the tree are labeled 0 or 1 and denote the result on the given inputs. This result should then be conveyed to the other party. Thus, the complexity of the protocol is one more than the depth of the tree. The communication complexity of f is that of the protocol with smallest complexity.

Deterministic protocols, D(f): In a deterministic protocol, the decisions made at each node of the tree model are deterministic and the tree itself is fixed. Most interesting functions depend on all of their input and hence all n bits of one party's inputs must be sent to the other. Thus, they have a communication complexity of $n + 1$.

Randomized protocols: In the randomized setting, we allow Alice and Bob to toss coins during the computation. Also, we want the computation to be correct on a significant fraction ($\frac{2}{3}$) of the random strings. The cost of the protocol is the worst case communication complexity over all random strings. There are two variants in this setting:

1. **Public coin**: In query complexity, we discussed how a randomized decision procedure can be viewed as a probability distribution over deterministic decision procedures. Similarly, a

randomized communication protocol can be viewed as a probability distribution over deterministic protocols. Then, Alice and Bob use a public random string to sample a protocol from this distribution and use it for the rest of the communication. It can be shown that this is equivalent to Alice and Bob using a common random string for their computation. The complexity of f under public coin randomness is denoted by $R_{pub}(f)$.

2. **Private coin**: Here, Alice and Bob use coin tosses locally and the outcomes are not known to each other. Communicating local random bits also contributes to communication complexity. The complexity of f under private coin randomness is denoted by $R(f)$.

Quantum protocols: In this setting, Alice and Bob are quantum machines and exchange qubits between them. Like in the randomized case, there are two measures of complexity. The two machines may have a prior set of entangled states, in which case the complexity is denoted by $Q_{ent}(f)$. If they do not share entangled states, then the complexity is denoted by $Q(f)$.

We will prove that there is an exponential gap between D and R. We will also establish a relation between R and R_{pub}. All known bounds for fully specified functions establish only a polynomial gap between randomized and quantum settings. For functions that are not fully specified, exponential gaps are known to exist.

1.1 Examples

The following are some interesting problems in this setting:

1. Equality: Whether the two input strings given to Alice and Bob are equal? $EQ(x, y) = \chi[x = y]$
2. Disjointness: Interpreting the input strings as the characteristic sequence of subsets of a set of size n, are the two subsets disjoint? $DISJ(x, y) = \chi[x \cap y = \emptyset] = \neg \vee_{i=1}^{n} (x_i y_i)$
3. Inner product: $IP(x, y) = \Sigma_{I=1}^{n} x_i y_i$.

Table 1 shows the complexities for these problems under different models. We prove some of these results in the next subsection.

Table 1: Complexities of $EQ, DISJ, IP$ under $D, R, R_{pub}, Q, Q_{ent}$

f	$D(f)$	$R(f)$	$R_{pub}(f)$	$Q(f)$	$Q_{ent}(f)$
EQ	$\theta(n)$	$\theta(\log n)$	$\theta(1)$	$\theta(\log n)$	$\theta(1)$
$DISJ$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(\sqrt{n})$	$\theta(\sqrt{n})$
IP	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$

1.2 Protocols

There are several classes of protocols that can be used to solve communication problems. We will discuss some of them and use them to derive a few bounds listed in Table 1.

Trivial: The trivial protocol is for Alice to send all of its input x to Bob and Bob will compute and share the value of $f(x, y)$. The complexity will be $\theta(n)$ and is the best we can hope for all the examples functions under the deterministic model.

Fingerprinting: Alice can create a short fingerprint of x and send it to y. The finger print should be in such a way that Bob can guess the correct answer with a reasonable accuracy. For example, consider the following $\theta(\log n)$ protocol for EQ under R. Alice computes $x \mod p$ for a randomly chosen prime number p of $2\log n$ bits and sends the modulus as well as p to Bob. Bob concludes the result by computing $y \mod p$ and comparing it to $x \mod p$. If $x \neq y$, $x \equiv y \mod p$ for at most n choices of primes p since $x < 2^n$. Using $2\log n$ bits, we have approximately $\theta(n^2)$ choices for p and $x \neq y \mod p$ for most of them.

Hadamard code: Using a public random string r of length n, Alice sends the r^{th} bit of the Hadamard encoding of x to Bob. If $x \neq y$, the bit that Bob computes using y will be different with probability $\frac{1}{2}$. We can repeat this protocol many times to get the desired accuracy. This gives a $\theta(1)$ R_{pub} protocol for EQ.

Quantum query: If f can be mapped to another function g whose *query complexity* is $p(n)$ such that each bit of the mapping can be computed using minimal communication, say $q(n)$, then the communication complexity of f is $p(n)q(n)$. For example, in last lecture, we mentioned that the quantum query complexity of the OR function is $\theta(\sqrt{n})$. The disjointness function $DISJ$ can be mapped to the OR function, as we saw earlier. Thus, we have a $\theta(\sqrt{n}\log n)$ protocol for $DISJ$ as follows: Alice runs the decision tree corresponding to OR and every time the i^{th} bit of the input is queried, Alice sends the configuration $|i, b, \xi\rangle$ to Bob after encoding x_i in ξ. Bob extracts x_i, computes $z_i = x_i y_i$ and sends back the configuration $|i, b \oplus z_i, \xi\rangle$. Alice returns this configuration as the answer to the query. Finally, Alice sends the negation of the decision tree's output to Bob. The length of ξ for this exchange turns out to be constant and hence the index i, which takes $\log n$ bits, is the dominating term. Since the query complexity of OR is $\sqrt{n}$, this protocol communicates $\theta(\sqrt{n}\log n)$ bits. Note: This protocol does not have the tight bound specified in Table 1.

2 Lower Bounds

In this part of the lecture, we establish the lower bounds for the problems discussed earlier. In order to do that, we need to introduce some new terminology.

Definition 1. *The* characteristic matrix M_f *of* f *is a* $2^n \times 2^n$ *matrix such that* $M_f(i,j) = f(b_i, b_j)$ *where* b_i, b_j *are the binary representations of* i *and* j *respectively.*

Definition 2. *A* combinatorial rectangle *in a matrix* M *is a* $p \times q$ *matrix induced by a subset of* p *rows and* q *columns of* M*.*

Definition 3. *A combinatorial rectangle of the characteristic matrix* M_f *is* f-monochromatic *if all the entries in the rectangle are equal.*

In the binary tree visualization of a communication protocol, there is a combinatorial rectangle A, B attached with each node of the tree that represent the subset of inputs for which the protocols reach that node. A node corresponding to Alice splits A into two disjoint sets, one for each of its successors. A node corresponding to Bob splits B into two disjoint sets. It is easy to see that the combinatorial rectangles associated with the leaves of the binary tree are *f-monochromatic*. Since f is fully specified, each cell of M_f is present in the combinatorial rectangle of at least one of the 2^{depth} leaves of the tree. Thus, each protocol for f involving an exchange of b bits induces a tiling of M_f into at most 2^b f-monochromatic combinatorial rectangles.

Definition 4. *T(f) is the minimum number of f-monochromatic rectangles induced in M_f by any communication protocol for f.*

Fact: $D(f) \geq \log T(f)$

2.1 Fooling set method

Claim 1. *If $S \subseteq \{0,1\}^n \times \{0,1\}^n$ such that:*

1. $(\exists b)(\forall (x,y) \in S) f(x,y) = b$
2. $\forall (x_1,y_1) \neq (x_2,y_2) \in S : f(x_1,y_2) \neq b \vee f(x_2,y_1) \neq b$

then, $|S| \leq T(f)$.

Proof. Let S' be the combinatorial rectangle induced by $A = \{x|(x,y) \in S\}$ and $B = \{y|(x,y) \in S\}$. The two conditions on S ensure that every f-monochromatic combinatorial rectangle in the characteristic matrix of S' under f is of size 1. Thus, no tiling of the characteristic matrix of S', and consequently M_f, can have fewer than $|S'|$ rectangles. Moreover, $|S'| = |A \times B| \leq |S|$. □

Examples: The set $S = \{(x,x)|x \in \{0,1\}^n\}$ satisfies the conditions in claim 1 for EQ. Thus, $D(EQ) \geq \log |S| = \log 2^n = n$. This is also easily seen by the fact that M_{EQ} is the identity matrix and requires 2^n tiles to cover each of the 2^n non-zero entries in it.

As another example, $S = \{(x,\bar{x})|x \in \{0,1\}^n\}$ satisfies condition 2 in claim 1 for $DISJ$ because for any two pairs $(x,\bar{x})$ and $(y,\bar{y})$ if $x \neq y$ then y has at least one element in common with $\bar{x}$. Thus, $D(DISJ) \geq \log |S| = n$.

2.2 Bounding the size of maximum monochromatic combinatorial rectangle

To illustrate this method, consider the inner product function IP. We will show shortly that the size of any 0-chromatic combinatorial rectangle in M_{IP} is at most 2^n. At least half of the 2^{2n} entries in M_{IP} are zeroes. Thus, $T(f) \geq \frac{2^{2n-1}}{2^n} = 2^{n-1}$. And, $D(IP) \geq \lceil \log T(f) \rceil = n$.

To prove the upper bound on the size of a 0-chromatic combinatorial rectangle, consider a combinatorial rectangle $A \times B$ where $A, B \subseteq \{0,1\}^n$. Then, $f(A \times B)$ is equal to 0 and so is $f(A' \times B')$ where $A' = span(A)$ and $B' = span(B)$. Here, the *span* of a set A of vectors is the set of all linear combination of A over the finite field of length 2. We can show that for IP, $dim(A') + dim(B') \leq n$ and hence $|A \times B| \leq |A' \times B'| \leq 2^n$.

2.3 Using rank of M_f

Claim 2. $Rank(M_f) \leq T(f)$

Proof. For a tile t of any tiling of M_f, consider the matrix C_t obtained by setting all entries outside of t in M_f to zero. C_t will be the all zero matrix for all zero-tiles. Thus, if T_1 is the set of all 1-tiles of the tiling,

$$\begin{aligned} M_f &= \sum_{t \in T_1} C_t \\ Rank(M_f) &\leq \sum_{t \in T_1} Rank(C_t) \\ Rank(M_f) &\leq |T_1| \leq T(f) \end{aligned}$$

The second equation is obtained using the fact that the *Rank* function is subadditive. The last line is derived using the fact that the rank of C_t for any t in T_1 has rank 1. □

As a corollary to the above claim, $D(f) \geq \log Rank(M_f)$. As examples for this technique, since the matrix M_{EQ} is the identity matrix, $Rank(M_{EQ}) = 2^n$. Thus, $D(EQ) \geq n$.

Exercise 1. *Derive $D(IP)$ using this technique.*

Conjecture 1 (*Log rank Conjecture*)**.** *There exists a constant c such that the complexity of deterministic communication protocols for f is bounded by $(\log Rank(M_f))^c$ where M_f is the characteristic matrix of f.*

3 Randomized Lower Bounds:

For R_{pub}, we can view the communication protocol as a two player game and use the Min-Max strategy similar to the one we used for randomized query complexity. In this section, we discuss other techniques and results for R and R_{pub}.

Claim 3. $R(f) = \Omega(\log D(f))$

Proof. In a randomized protocol, each node in the binary tree visualization of a communication protocol decides the bit to send based on its input as well as coin flips. Now, we associate a probability distribution, instead of just a subset, of inputs with each node of the tree. Now, for a given input, for each leaf ℓ, Alice can compute $\Pr_r[\text{on input } x, \text{ Alice ends up on leaf } \ell]$. Since this probability is computed by performing $R(f)$ operations on the initial probability of 1 for the root node, it can be expressed using $O(R(f))$ bits. Alice will send $2^{R(f)}O(R(f))$ bits. Since the two random coin flips of Bob are independent of those of Alice, Bob can compute the probability of ending up in a leaf ℓ for input (x, y) by computing a value similar to the one computed by Alice and multiplying them together. Based on the label of each leaf, Bob can compute the probability of acceptance. This gives a $2^{R(f)}O(R(f))$ deterministic protocol. Thus, $R(f) \geq \log D(f)$. □

This shows that the randomized protocol for EQ using fingerprinting is optimal.

Claim 4. $R(f) = O(R_{pub}(f) + \log n)$

To show this result, we can amplify the success probability of the public coin randomized protocol and then use an averaging argument to prove the existence of polynomially many public random sequences with sufficiently high success probability on all inputs. So, Alice and Bob select one sequence using $\log n$ random bits and then follow the public coin protocol.

4 Applications in other areas

An important reason for studying communication complexity is that problems in other domains can be reduced to communication problems. Then, lower bounds for communication problems establish lower bounds for the corresponding problem. Examples include Area-Time lower bounds for VLSI chip design and query complexity on data structures. This lecture, we reduce the bounded fan-in circuit depth of a function to a communication problem.

Theorem 1. *Let $f : \{0,1\}^n \to \{0,1\}$. Alice and Bob have an input $x \in f^{-1}(0)$ and $y \in f^{-1}(1)$ respectively. The minimum communication complexity of finding an index i such that $x_i \neq y_i$ is equal to the minimum depth of a bounded fan-in circuit computing f.*

Note: The communication problem in this theorem is not a decision problem.

Proof. $\leq$: Consider a fanin 2 circuit C of depth d that computes f. WLOG, we can assume that all but the bottom-most level has AND and OR gates. Each gate g in the circuit computes a function f_g of the inputs. Suppose the top most gate is an AND gate. Since the output of the gate is 0 on input x at least one of the gate's inputs will be zero. Alice will choose the gate that produced the 0 input and convey it to Bob (using a one bit). On the other hand, if the gate is an OR gate, Bob will choose an input to the OR gate that is 1 and conveys the gate that produced that output to Alice. This process continues for d iterations, while maintaining the invariant that x evaluates to 0 and y evaluates to 1 on the gate under consideration. After d iterations, the process terminates at an input gate, which has different values on x and y. Note that we just have to convey the d bit indices to choose the gates at each depth. The circuit can be hardwired into both the parties and the result need not be communicated because both of them know the output at the end of the protocol.

$\geq$: For this direction, we prove the following stronger result.

Claim 5. *For any A, B such that $A \subseteq f^{-1}(0)$ and $B \subseteq f^{-1}(1)$, if there exists a protocol that works for A, B and is of communication complexity d, then there exists a circuit of depth $\leq d$ that outputs 0 on A and 1 on B.*

Proof by induction. For the base case, when $d = 0$, Alice and Bob know the index i in the input that are different. The corresponding depth 0 circuit is either x_i or $\bar{x_i}$ depending on the i^{th} bit in x and y. For the induction step, suppose there exists a communication protocol of complexity d that works for A, B. Suppose Alice initiates the proceedings in this protocol. Alice sends a 0 if its input x is in a subset A_0 of A and a 1 if its input is in a subset A_1 of A (A_0, A_1 are a partition of A). Now, there exist communication protocols of depth $\leq d - 1$ for $\{A_0, B\}$ and $\{A_1, B\}$. From the induction hypothesis, there exist circuits C_0 and C_1 such that:

1. C_0 evaluates to 0 on an input from A_0 and 1 on an input from B.
2. C_1 evaluates to 0 on an input from A_1 and 1 on an input from B.

We must construct a circuit from C_0 and C_1 that produces 0 on an input from A and 1 on an input from B. Combining the output of C_0 and C_1 using an AND gate does the job.

On the other hand, if Bob starts the communication, the dual argument holds and we combine the two circuits from the induction hypothesis using an OR gate. □

Corollary 1. *Monotone NC^1 circuits cannot decide if a graph has a perfect matching.*