Lecture 1

Complexity Classes

Lecturer: Jaikumar Radhakrishnan **Date:** 7 January 1998

Our purpose in this course is to study the problems that can be solved efficiently on a computer. In any such formal study, we must first understand what a problem is, how a computer is expected to solve it, and when a proposed solution is to be considered efficient. In the following sections, we present the treatment of these issues as is now accepted in Complexity Theory.

1.1 Alphabet, Strings, Languages

In our study, data will be represented as a string of symbols from some finite set. The finite set used will be called the *alphabet*. For example, the sets $\{a, b, \ldots, z\}$, $\{0, 1, 2, \ldots, 9\}$ and $\{0, 1\}$ can serve as our alphabet. A *string* over an alphabet is a a finite sequence of symbols from the alphabet. For example, "*complexity*" is a string over the alphabet $\{a, b, \ldots, z\}$, and "007" is a string over the alphabet $\{0, 1, \ldots, 9\}$. A string with no symbols at all is called the *empty string* and denoted by ϵ. For an alphabet Σ, we shall denote the set of all strings over Σ (including the empty string ϵ) by Σ^*.

The *length* of a string is the number of symbols in it. For example, the length of the string "007" is 3. We shall denote the length of the string x by $|x|$; thus $|\epsilon| = 0$. The set of all strings of length k over the alphabet Σ will be denoted by Σ^k; observe that $|\Sigma^k| = |\Sigma|^k$ (is $0^0 = 1$?).

A *language* is a set of strings over an alphabet; that is, a language over the alphabet is a subset of Σ^*. A language may be finite; for example, $\{00, 01, 10001\}$ is a finite language over the alphabet $\{0, 1\}$. Or it could be infinite; for example,

$$\{w \in \{0,1\}^* \,:\, w \text{ has an equal number of 0's and 1's}\}.$$

Similarly, if w^R denotes the reversal of the string w, then the language of *palindromes*,

$$\{w \in \Sigma^* \,:\, w = w^R\},$$

is in an infinite language for every nonempty alphabet Σ.

With every language L over the alphabet Σ one may associate a computational task as follows.

Input: A string $x \in \Sigma^*$.
Output: 'Yes', if $x \in L$;
'No', if $x \notin L$.

This task will be called the *recognition problem* for the language L. With the language L is associated the predicate $P_L : \Sigma^* \to \{0, 1\}$ defined by

$$P_L(x) = 1 \Longleftrightarrow x \in L.$$

Clearly, the language recognition problem for the language L is equivalent to computing P_L.

At first sight, a language recognition problem may appear to be rather abstract, far removed from the tasks one normally expects a computer to perform. To motivate the study of language recognition problems, we consider the following natural task in connection with graphs.

Input: An undirected graph G.
Output: 'Yes', if G is connected;
'No', if the G is not connected.

To phrase this as a language recognition problem, we must somehow translate graphs into strings. This is easily done using the adjacency matrix of the graph. For example, the undirected graph (V, E) with

$$\begin{aligned} V &= \{1,2,3,4\}; \\ E &= \{\{1,2\},\{2,3\},\{3,4\},\{4,1\}\}, \end{aligned}$$

has adjacency matrix

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}.$$

Then the string corresponding to this graph is obtained by concatenating the rows; in this case, the string is "0101101001011010". With this correspondence in mind, we may say that

$$\{w \in \{0,1\}^* : w \text{ corresponds to a connected graph}\}$$

is the language corresponding to the task described above. Often the method of coding graphs using strings is not controversial, or is quite irrelevant; it is, therefore, preferable to omit it and write the language as

$$\{G : G \text{ is a connected graph}\},$$

where we assume tacitly that some reasonable coding over some alphabet is being employed to represent graphs as strings.

1.2 Machines and Algorithms

Having modelled a typical computational task faced by a computer in terms of a language recognition problem, we now turn to modelling the device (the computer itself) on which such a task is to be performed. Historically, the Turing machine has been used to model the process of computation, and, perhaps, every complete and precise treatment must use a similar device. However, in our study we will rarely need this level of precision. It will suffice for our purposes to develop some intuitive understanding of how a typical computation proceeds. This we propose to do using examples.[1]

1. The language of Palindromes is recognized by the following procedure.

   Input: $w \in \{0,1\}^*$.

[1]For this course, it can be safely assumed that a valid solution for a computational problem is program in some programming language, say PASCAL. It is not hard to show that in certain broad terms the powers of this language is equivalent to the power of the Turing machine.

Step 1: Compute $n = |w|$.

Step 2: For $i = 1, 2, \ldots, n$,
if $w_i \neq w_{n-i+1}$, then output 'No'; STOP.

Step 3: Output 'yes'; STOP.

Before we accept this procedure, we must ensure that the steps performed are feasible for a computer. For instance, is it true that a computer, given a string, can compute some representation of its length? or, when given a position, determine whether that position is a 0 or a 1? These tasks the computer can perform easily, and we admit their use in the procedure. We shall refer to such a sequence of steps as an *algorithm*.

2. The language of connected graphs is recognized by the following procedure.

   Input: An undirected graph G.

   Step 1: Set $n = |V(G)|$.

   Step 2: Pick the first vertex in $V(G)$ and mark it.

   Step 3: While there is an edge $\{i, j\} \in E(G)$ with vertex i marked and j unmarked, mark vertex j.

   Step 4: If all the vertices of the graph are marked then output 'Yes'; otherwise output 'No'.

   Once again, before accepting this procedure, we must convince ourselves that all the steps are feasible. How does one mark a vertex? On possibility is to maintain in the memory an array of locations $A[1], A[2], \ldots, A[n]$. Initially, each $A[i]$ contains a 0 and whenever we need to mark a vertex i we set $A[i]$ to 1. Note that checking if vertex i is marked is now easy – we just check if $A[i] = 1$. Also, when implementing Step 3, we must go through the adjacency matrix systematically and check if any edge meets the condition stated there. It is not hard to convince oneself that all this can indeed be done.

Implementation issues such as those encountered above will arise often in our study. What is feasible and easy to implement will rarely be controversial, and we will ignore these technical details. In principle, we must use a formal programming language or a Turing machine to justify these algorithms. But the benefit one derives from such an exercise is not worth the time spent on it. Moreover, by now most have seen some programming language and developed some intuition about the process of computing. The informal language of our algorithms will be much easier for them to appreciate than the cold formal code of a Turing machine, where even simple steps need to be carried out in a circuitous way. Yet, if due to the informal nature of description, there arises some doubt about the feasibility of some step, we shall fall back to the formal models where things can be treated precisely.

1.3 Complexity Classes

There are many problems that can, in principle, be solved by a computer. However, the solutions proposed are not always realistic, for they take unreasonably long to finish, even on inputs of moderate size. Any theory that purports to describe what is efficiently computable, must take into account the resources used in the solutions. One of the goals of complexity theory is to classify problems based on the amount of resources they need. The two resources most frequently considered are *time* and *space*.

Let us consider time complexity first. Suppose we have a language L that is recognized by an algorithm A. How efficient is A? Now, A might take different amounts of time on different inputs; it is quite reasonable that as the input string becomes longer the algorithm should need more time to decide. For evaluating the efficiency of such an algorithm one proceeds as follows. The performance of A is studied as a function of input length. That is, for each natural number n we compute the maximum amount of time that A might take on a string of length n. The running time of A, denoted by $t_A(n)$, is given by

$$t_A(n) = \max \text{time taken by } A \text{ on input of length at most } n.$$

Before we describe the classification of computational problems, we need to introduce some notation pertaining to the growth of functions. Let f and g be functions from $\mathbf{N}$ to $\mathbf{N}$. We write $f = O(g)$ if there is a constant c and and integer $n_0 \in \mathbf{N}$ such that

$$f(n) \leq cg(n) \text{ for all } n \geq n_0.$$

We write $f = \Omega(g)$ if there is a constant $c > 0$ and and integer $n_0 \in \mathbf{N}$ such that

$$cg(n) \leq f(n) \text{ for all } n \geq n_0.$$

We write $f = o(g)$ if for all $c > 0$ there exists an $n_0 \in \mathbf{N}$ such that

$$f(n) < cg(n) \text{ for all } n \geq n_0.$$

We write $f = \omega(g)$ if for all $c > 0$ there exists an $n_0 \in \mathbf{N}$ such that

$$cg(n) < f(n) \text{ for all } n \geq n_0.$$

For $t : \mathbf{N} \to \mathbf{N}$, $\mathrm{DTime}(t(n))$ is defined by

$$\mathrm{DTime}(t(n)) = \{L \subseteq \Sigma^* : \text{there is an algorithm } A \text{ recognizing } L \text{ such that } t_A(n) = O(t(n))\};$$

that is, $\mathrm{DTime}(t(n))$ consists of exactly those languages that can be recognized by some algorithm with whose running time is *bounded* by $t(n)$. The letter 'D' in DTime stands for *deterministic*, emphasizing that the algorithms we are considering are deterministic; later we shall consider nondeterministic computation and use the notation $\mathrm{NTime}(t(n))$ to denote the complexity class arising there.

Similarly, for space complexity, we have the class $\mathrm{DSpace}(s(n))$ defined by

$$\mathrm{DSpace}(s(n)) = \{L \subseteq \Sigma^* : \text{there is an algorithm } A \text{ recognizing } L \text{ such that } s_A(n) = O(s(n)).\}$$

Here $s_A(n)$ denotes the space used by algorithm A, in the worst case, on an input of length n.

1.4 Interpreter programs

It is natural to think of computer programs as being themselves coded as strings over some alphabet. Consider another program that takes as input the code of a program and its input and *executes* it. That is the program takes as input the pair $\langle \lceil \mathsf{A} \rceil, x \rangle$, where $\lceil \mathsf{A} \rceil$ is the code of the program A and $x \in \Sigma^*$, and produces the output $\mathsf{A}(x)$. The existence of such a program is crucial to many of the results in classical computability theory. In the usual treatment of this subject, the role of such a program is played by the universal Turing machine. Since we have been dealing with programs, we will call it the *Interpreter Program*, and denote it by $\mathcal{I}$. In some applications, we will find the following convention useful: the string w and $w10^n$ represent the code of the same algorithm (for $n \geq 1$).

We shall make certain additional assumptions about the interpreter program. We shall assume that the simulation performed is *step by step*. That is, the interpreter takes a step in the code of A and executes it, goes to the next step, executes it, and so on. It is not allowed to combine the operations of many steps of A. It might happen that one step of program A requires many steps to interpret, but we will assume that the simulation is efficient in the following sense: *the running time of the computation* $\mathcal{I}(\langle\lceil\mathsf{A}\rceil, x\rangle)$ *is not much more than the original running time of the computation* $\mathsf{A}(x)$; if $t_{\mathsf{A}}(x)$ is the running time of A on input x, then the running time of $\mathcal{I}$ on $\langle\lceil\mathsf{A}\rceil, x\rangle$ is at most $c \cdot t_{\mathsf{A}}(x) \cdot \log t_{\mathsf{A}}(x)$, where c is constant that depends on A but not x. Similarly, we assume that the space required by the interpreter is not much more than the space required for $\mathsf{A}(x)$; if $s_{\mathsf{A}}(x)$ is the space used by A on input x, then the $\mathcal{I}$ on $\langle\lceil\mathsf{A}\rceil, x\rangle$ uses at most $d \cdot s_{\mathsf{A}}(x)$ space, where d is a constant depending on A but not x.

1.4.1 Space and Time Hierarchy

It appears reasonable to expect that if we allow our computations to use more space or time, then we should be able to recognize more languages. To prove this formally, we need to impose certain restrictions on the functions that measure the space and time used by an algorithm.

Definition 1.1 (Space constructible function) *We say that* $s : \mathbf{N} \to \mathbf{N}$ *is* space constructible *if*

1. $s(n) \geq \log n$.
2. *The map* $1^n \mapsto 1^{s(n)}$ *is computable in space* $O(s(n))$*. That is, there is an algorithm which given* 1^n *as input, produces* $1^{s(n)}$ *as its output, while using at most* $s(n)$ *units of work-space.*

Theorem 1.2 (Space hierarchy) *Suppose* $s_1, s_2 : \mathbf{N} \to \mathbf{N}$ *such that* $s_1 = o(s_2)$ *and* s_2 *is space constructible. Then DSpace*(s_1) *is a proper subset of DSpace*(s_2)*.*

Proof. (Sketch.) It can be easily verified that any language in DSpace(s_1) is a subset of DSpace(s_2). We will now show that the following language is in DSpace(s_2) but not in DSpace(s_1).

$$L = \{w : \mathcal{I}(w, w) \text{ halts and rejects using space at most } s_2(|w|) \text{ and time at most } 2^{s_2(|w|)}\}.$$

We have two claims.

Claim 1.3 $L \in$ *DSpace*(s_2)*.*

Since s_2 is space constructible, we can mark off space $s_2(|w|)$ in the work-space, and restrict all later computation to stay within this space. Furthermore, a counter which starts at 0 and counts up to $2^{s_2(|w|)}$ can be implemented using $s_2(|w|)$ space. Thus, the interpreter program can recognize L, if it is suitably modified to run in $s_2(|w|)$ space and $2^{s_2(|w|)}$ time.

Claim 1.4 $L \notin$ *DSpace*(s_1)*.*

We need to show that no algorithm that uses s_1 space can recognize L. Fix one such algorithm A. We first observe that the space used by $\mathcal{I}(\lceil\mathsf{A}\rceil, w)$ is at most $d \cdot s_1(|w|)$. Since $s_1 = o(s_2)$, for $|w|$ large enough, $d \cdot s_1(|w|) \leq s_2(|w|)$; thus $\mathcal{I}$ can simulate the entire computation in the available. Furthermore, since A always halts, it can use at most $2^{O(s_1(|w|))}$ time; thus, for $|w|$ large enough the interpreter program can complete the simulation of A in time $2^{s_2(|w|)}$. Assume, that both the above properties hold whenever $|w| \geq n_0$.

Consider the computation $\mathcal{I}(\langle w, w\rangle)$ for $w = \lceil \mathsf{A} \rceil 10^{n_0}$. Since $|w| \geq n_0$ and $\mathsf{A}_w = \mathsf{A}$, the simulation $\mathcal{I}(\langle w, w\rangle)$ terminates using at most $s_2(|w|)$ space and $2^{s_2(|w|)}$ time. But then, we have

$$w \in L \text{ iff } \mathsf{A}_w \text{ rejects } w \text{ iff } \mathsf{A} \text{ rejects } w.$$

Thus, A does not recognize L (because it produces an incorrect answer on input w). ∎

This theorem tells us that additional space does lead to an increase in the power of the algorithm. What about time?

Definition 1.5 (Time constructible) *We say that $t : \mathbf{N} \to \mathbf{N}$ is* time constructible *if*

1. $t(n) \geq n \log n$.
2. *There is algorithm that on input 1^n runs for time $O(t(n))$ and produces as output the binary representation of $t(n)$.*

Theorem 1.6 (Time hierarchy) *Suppose $t_1, t_2 : \mathbf{N} \to \mathbf{N}$ such that $t_1(n) = o(t_2(n)/\log t_2(n))$ and t_2 is time constructible. Then, $DTime(t_1)$ is a proper subset of $DTime(t_2)$.*

Proof. Similar to the space hierarchy theorem. Omitted. ∎

Tractable Computation: In complexity theory, the following class plays a central role.

$$\mathcal{P} = \bigcup_{k \geq 0} \mathrm{DTime}(n^k).$$

The notion of polynomial time has several nice closure properties that make it an attractive mathematical approximation for what one would intuitively consider feasible. Moreover, many problems of practical interest are not even know to be solvable in polynomial time. It is, therefore, natural to try to obtain polynomial time algorithms for them, before proceeding to propose practical solutions. Though $\mathcal{P}$ has come to be accepted as the class of languages that can be recognized efficiently, the argument in its favour is by no means compelling. For example, an algorithm whose running time in n^{100} is far from being practical; yet, the definition above considers it reasonable. In any case, it is certainly safe to say that a problem not solvable in polynomial time is intractable.

Similar to the class $\mathcal{P}$ we have, for space complexity, the class

$$\mathrm{PSPACE} = \bigcup_{k} \mathrm{DSpace}(n^k).$$

1.4.2 Nondeterminism

Let us accept, then, that $\mathcal{P}$ is the set of problems that have efficient solutions. As mentioned earlier, there are many natural problems that are not known to be in $\mathcal{P}$. As examples consider the following three problems on graphs.

1. *Hamilton Graphs.* Input: a graph G. Task: Determine if G has a Hamilton cycle.
2. *Graph Colouring.* Input: a graph G. Task: Determine if the graph has a proper colouring using at most three colours.
3. *Independent Set.* Input: A graph G and a number k. Task: Determine if the graph has an independent set of size k.

For these three problems, no practical algorithm is known; perhaps none exists, even under the somewhat generous notion of what is practical that is implicit in the definition of the class $\mathcal{P}$. Can we somehow enhance the power of our computational model a little so that these problems can be solved? What if our computers had the ability to guess or make nondeterministic choices? It turns out that many problems, including the ones above, can be easily solved if the computer is allowed nondeterminism. What is *nondeterminism*?

In nondeterministic computation, we permit the algorithm to make guesses (imagine that our PASCAL program can invoke a function that could return either a 0 or a 1; the program can use these 'magic' digits and compute). To clarify this concept let us consider an efficient nondeterministic algorithm for the language of Hamilton graphs.

Input: An undirected graph G.

Step 1: Choose a subset S of edges of G.

Step 2: Check if the the edges in S form a Hamilton cycle.

How do we choose a subset S of the edges? For each edge of G we invoke the guessing device, and if the outcome is 1 then we include the edge and if it is 0 we omit it. In other words, we guess the characteristic vector of the set S. How many guesses did it take? One for each edge of G. Once the set has been obtained, how hard is it to check if it describes a Hamilton cycle? Not hard – just verify that the graph described by them is connected and that every vertex of G has exactly two edges from S incident on it. This can certainly be done in polynomial time. So the algorithm is efficient. Note that the second step is entirely deterministic because we make no guesses.

The algorithm is not of the usual kind; it is not clear right in the beginning how the computation will proceed. Much depends on which subset S is chosen in the first step, and for different subsets the final conclusion may be different. It is possible for the algorithm, however, to produce the answer 'No' even when the input contains a Hamilton cycle. However, if the graph is not Hamiltonian, the algorithm will never produce a 'Yes', and, if the graph is Hamiltonian, there is *at least one* sequence of choices (lucky guesses) that would cause the algorithm to produce a 'Yes' (Why?). This last sentence contains the essence of nondeterministic computation.

A nondeterministic algorithm A recognizes a language L if

- If $x \in L$ then there exists a sequence of guesses made by A that would cause it to produce the answer 'Yes'.
- If $x \notin L$ then A always produces a 'No'.

The first condition appears overly generous – of the possibly millions of choice sequences, there might be just one that causes A to produce the answer 'Yes', yet this is sufficient. It is, no doubt, unrealistic; the reader might try to imagine that the following law is in operation: if anything can go right (cause you to accept), it will! So, for inputs that must be accepted we trust our luck, but for inputs not in the language we must make sure that nothing will cause us to accept.

As another example, let us consider the following algorithm for determining if a graph G has an independent set of size k.

Input: An undirected graph G and an integer k.

Step 1: Choose a subset S of $V(G)$.

Step 2: Output 'Yes' if $|S| \geq k$ and there are no edges of G with both ends in S; otherwise, output 'No'.

The reader should convince herself that this nondeterministic algorithm constitutes a valid solution under the definition stated above.

The operation of a nondeterministic program can be visualized as a rooted binary tree, where each node corresponds to a guess, and the two outgoing edges of the node correspond to the two possible outcomes of the guess. After making the sequence of guesses the algorithm moves to its deterministic phase and, based on the input and the guesses, either accepts or rejects the input. Thus, the input is in the language if at least one leaf of the tree corresponds to the answer 'Yes', and if all the leaves correspond to 'No', the input is not in the language.

As usual, we measure the running time of the nondeterministic algorithm as a function of the input size. For a nondeterministic algorithm A, its running time $t_A(n)$ is given by

$$t_A(n) = \max_{w:|w|=n} \max_{\text{guess sequence } y} \text{time taken by } A \text{ on input } w \text{ with guess } y.$$

Then NTime$(t(n)$ denotes the set of languages that can be recognized by nondeterministic algorithms whose running time is $O(t(n))$. The class $\mathcal{NP}$, for *nondeterministic polynomial time*, is defined by

$$\mathcal{NP} = \bigcup_k \text{NTime}(n^k).$$

Similarly, $s_A(n)$ denotes the maximum number of cells algorithm A uses on inputs of length n, and NSpace$(s(n))$ denotes the set of languages that can be recognized by non-deterministic algorithms that use space $O(s(n))$.

Another view of $\mathcal{NP}$: In our description of nondeterministic algorithms there were two phases: in the first phase, a string was guessed; in the second, a deterministic computations was performed using the input and the guess. If the entire computation is to be performed in polynomial time then the length of the guess and the time taken for the second phase must both be bounded by some polynomial. Suppose we denote the computation performed in the second phase by the predicate $P(x, y)$, (that is, $P(x, y)$ is true iff on input x and guess y the algorithm produces the output 'Yes'), then we have, for a constant k and all $x \in \Sigma^*$

$$x \in L \Longleftrightarrow \exists y(|y| \leq |x|^k)\ P(x, y).$$

We may thus formulate an alternative definition of $\mathcal{NP}$ as follows. A language L is in $\mathcal{NP}$ iff there exists a (deterministic) polynomial time computable predicate $P(x, y)$ and a constant k such that for all x

$$x \in L \Longleftrightarrow \exists y(|y| \leq |x|^k)\ P(x, y). \tag{1.1}$$

That all languages in $\mathcal{NP}$ can be expressed in this form follows from the discussion above. On the other hand, that every language defined in the form (1.1) is in $\mathcal{NP}$ can be shown by explicitly constructing a nondeterministic (guess and verify) algorithm (How?).

We shall often drop the condition that $|y| \leq |x|^k$ (in $\exists y(|y| \leq |x|^k)$) and instead write $\exists^P y$, where the superscript P will remind us that we are allowed to choose as guesses only those y's whose length is bounded by a fixed polynomial in the length of x. Then we rewrite (1.1) as

$$x \in L \Longleftrightarrow \exists^P y\ P(x, y). \tag{1.2}$$

The characterization (1.2) has an interesting interpretation. Suppose we are in possession of an algorithm A_P that computes the predicate P in polynomial time. Now if someone claims that a string x is in L, we will ask for the guess y. Once we have both x and y, we run A_P on input (x, y) and verify the claim.[2] This *protocol* is feasible because we know that there is a reasonable sized guess y and that A_P runs in polynomial time.

Thus, we may interpret y as a *witness* or *proof* of the fact that $x \in L$ and the computation of $P(x, y)$ using A_P as the proof verification process. We are therefore justified in saying that $\mathcal{NP}$ is the class of languages that have easy to verify membership proofs.

Easy inclusions: First observe that our definitions immediately give $\mathcal{P} \subseteq \mathcal{NP}$ and $\mathcal{P} \subseteq$ PSPACE (How?).

A somewhat more interesting inclusion is

$$\mathcal{NP} \subseteq \text{PSPACE}.$$

To see this consider any language $L \in \mathcal{NP}$. We know that there exists a polynomial time computable predicate $P(x, y)$ such that

$$x \in L \iff \exists^P y\ P(x, y).$$

Let A be an algorithm that computes P in polynomial time. We now show how we can recognize L using only polynomial amount of space. On input x, the algorithm tries all potential strings y and each time computes $P(x, y)$ using the algorithm A. It is not hard to see that one can generate all strings y systematically one after the other using just polynomial amount of space. Since A is a polynomial time algorithm, it uses only polynomial amount of space. The algorithm accepts the input x if any of the witnesses y cause A to accept; otherwise it rejects x.

$\mathcal{P}$ versus $\mathcal{NP}$: The condition for a language L to be in $\mathcal{P}$ can be stated in a manner similar to (1.2):

$$x \in L \iff P(x),$$

where P is some polynomial time computable predicate. That is, for languages in $\mathcal{P}$ memberships claims can be checked efficiently (without any additional proof). It is natural, then, to compare the two notions $\mathcal{P}$ and $\mathcal{NP}$. Is it true, perhaps, that for each polynomial time computable predicate $P(x, y)$ and constant k, there exists a polynomial time computable predicate $Q(x)$ such that for all x

$$\exists y(|y| \leq |x|^k)\ P(x, y) \iff Q(x)?$$

If true this would imply that whenever a language has easy to verify membership proofs, the language itself can be recognized efficiently (without proof); that Hamilton graphs can be recognized efficiently; that it is easy to determine if a graph is 3-colourable; ...

> The $\mathcal{P}$ versus $\mathcal{NP}$ question grew out of developments in mathematical logic and electronic technology during the middle of the twentieth century. ... This question has attracted considerable attention. Its intuitive statement is simple and accessible to non-specialists, even those outside science. By embracing issues in the foundations of mathematics as well as in the practice of computing, it gains something in character beyond that of a mere puzzle, but with apparently deeper significance and broader consequences.

– **Mike Sipser, 1992**

[2] Observe that we do not immediately have a way of verifying the claim that $x \notin L$.

The class co-$\mathcal{NP}$: The class $\mathcal{P}$ has an interesting property. Suppose a language L is in $\mathcal{P}$, and consider its complement $\overline{L}$ (consisting of all the strings not in L). Is $\overline{L}$ in $\mathcal{P}$? Yes, because one can easily obtain a program for recognizing $\overline{L}$ from the one for L by simply reversing the roles of 'Yes' and 'No'.

Now, what about the class $\mathcal{NP}$? Is it true, for example, that the language of non-Hamiltonian graphs[3] is in $\mathcal{NP}$? This is not immediately clear. Recall that $\mathcal{NP}$ is the class of languages having easy to verify membership proofs. What easy to verify proof can there be to convince one that a graph has no Hamilton cycle? It is not known whether or not such a proof always exists. It is, therefore, conceivable that the class consisting of the complement of the language in $\mathcal{NP}$ is different from $\mathcal{NP}$. This class is denoted by co-$\mathcal{NP}$. That is,

$$\text{co-}\mathcal{NP} = \{L \subseteq \Sigma^* : \overline{L} \in \mathcal{NP}\}.$$

In general, for a class $\mathcal{C}$, the class co-$\mathcal{C}$ is defined by

$$\text{co-}\mathcal{C} = \{L \subseteq \Sigma^* : \overline{L} \in \mathcal{C}\}.$$

As an example, consider the language L of 2-colourable (bipartite) graphs. The language is clearly in $\mathcal{NP}$ (guess the colouring and verify it). Is it in co-$\mathcal{NP}$? What do we need to show to put L in co-$\mathcal{NP}$? We need to show that $\overline{L}$, the language of non-bipartite graphs, is in $\mathcal{NP}$. It might, at first, seem hopeless to be able to obtain an easy to verify proof for non-bipartite graphs. Recall the following fact from graph theory.

Fact: A graph is not 2-colourable iff it has a cycle with an odd number of vertices.

So, to show that G is not bipartite, we just guess a sequence of vertices, and verify that it indeed forms an odd cycle in the graph. This is not hard to do. Thus, $\overline{L} \in \mathcal{NP}$ and, using what we already know about L, we may write $L \in \mathcal{NP} \cap \text{co-}\mathcal{NP}$.

Remarks

For most of the material covered in this lecture, the text book of Lewis and Papadimitriou, *Elements of the Theory of Computation* [LP81], constitutes a more than adequate reference. Our discussion of alphabets, strings and languages and the classes $\mathcal{P}$ and $\mathcal{NP}$ are based on the treatment in that book. A formal description of the Turing Machine model, which we chose to omit, can also be found there. The definitions of $O(f(n))$, $\Omega(f(n))$, $o(f(n))$ and $\omega(f(n))$ are taken from the book of Leiserson, Cormen and Rivest, *Introduction to Algorithms* [CLR90, p. 23]. The standard text treating complexity classes arising from time and space bounded computations is Hopcroft and Ullman's *Introduction to Automata Theory, Languages and Computation* [HU79]. The passage attributed to Mike Sipser above is taken from the paper *The History and Status of the $\mathcal{P}$ versus $\mathcal{NP}$ Question* [Sip92]. The following two books give an excellent modern treatment of the subject: Sipser's *Introduction to the Theory of Computation* by [Sip97] and Papdimitriou's *Computational Complexity* [Pap94]. Our description of the space and time hierarchy theorems is taken from Sipser's book.

[3] When considering a language L of graphs having a certain property, it will be convenient to take $\overline{L}$ to be the language of those graphs that do not have the property.

Lecture 2
Space Complexity

Lecturer: Jaikumar Radhakrishnan **Date:** 14 January 1998

Recall that for an algorithm A, $s_{\mathsf{A}}(n)$ denotes the maximum number of cells algorithm A uses on inputs of length n. In this lecture, we will consider computations that use $O(\log n)$ space on inputs of length n. Although this restriction on space appears sever, several interesting problems can still be solve. For example, the language of *palindromes* can be solved by a deterministic algorithm using $O(\log n)$ space. (To count up to n one does not require more than $\log n$ bits.)

$$\begin{aligned} \text{Log} &= \text{DSpace}(\log n); \\ \text{NLog} &= \text{NSpace}(\log n). \end{aligned}$$

It is not known if there is any language in $\mathcal{NP}$ that is not in L! What is the relationship between the classes Log, NLog and $\mathcal{P}$? Our first result in this direction is based on a remark we made in the last lecture: an algorithm running with limited space cannot run for very long.

Theorem 2.1 *Log* $\subseteq \mathcal{P}$.

Proof. Let $L \in$ Log; then, there is an algorithm A that recognizes L and uses space $O(\log n)$. Once an input (of length n) is fixed, the number of distinct configurations of this algorithm is bounded by n^c for some constant c (where c is independent of n). Thus, if A is to avoid going into an infinite loop it must run in polynomial time. ∎

We next study the relationship between NLog and $\mathcal{P}$.

Theorem 2.2 *NLog* $\subseteq P$.

Proof. Again, our argument is based on the observation that a space bounded algorithm cannot have too many configurations. Suppose $L \in$ NLog. Let A be a non-deterministic algorithm that recognizes L. We will now show how a deterministic polynomial time algorithm can determine whether or not a given string x ($|x| = n$) is in L. Consider the following graph G.

$$\begin{aligned} V(G) &= \text{set of all possible configurations of } \mathsf{A} \text{ on input } x; \\ E(G) &= \{\langle C, C' \rangle : \text{if } \mathsf{A} \text{ goes from } C \text{ to } C' \text{ in one step}\}. \end{aligned}$$

Let C_s be the starting configuration and C_t be the accepting configuration (we assume that it is unique). Then, $x \in L$ iff there is a path in G from C_s to C_t. Thus, determining if $x \in L$ reduces to solving the directed (s,t)-connectivity problem on the graph G. Since A runs in logarithmic space, the size of G is bounded by a polynomial in n. The claim follows by noting that the directed (s,t)-connectivity problem is in $\mathcal{P}$. ∎

We thus have the following containments:

$$\text{Log} \subseteq \text{NLog} \subseteq \mathcal{P} \subseteq \mathcal{NP} \subseteq \text{PSPACE}.$$

The space hierarchy theorem tells us that Log is a proper subset of PSPACE. In the next section, we will see that even NLog is a proper subset of PSPACE.

2.1 DSpace versus NSpace

Theorem 2.3 (Savitch's theorem) *(a) NLog $\subseteq$ DSpace$((\log n)^2)$.*

(b) For all space constructible functions s, NSpace$(s) \subseteq$ DSpace(s^2).

Proof. We will only prove the first part; the second part is similar.

Fix $L \in$ NLog, and a nondeterministic algorithm A that recognizes L using logarithmic space. Given input x ($|x| = n$), we wish to determine if $x \in L$ using a deterministic algorithm that uses space $O((\log n)^2)$. As observed before, this is equivalent to determining if the accepting configuration is reachable from the starting configuration. Unfortunately, we cannot in space $O((\log n)^2)$ expect to write down the entire description of the graph explicitly (because the graph has polynomially many vertices). Nevertheless, given two configurations one can determine in logarithmic space if A goes from the first to the second in one step. We will now present an algorithm to test reachability using this observation.

Procedure Reachable(C_i, C_f, t)
This procedure returns 1 iff C_f is reachable from C_i in at most t steps.

Step 1. If $C_i = C_f$ then return 1.

Step 2. If C_f is reachable from C_i in one step, then return 1.

Step 3. For all configuration C_{mid} do
If Reachable$(C_i, C_{\mathrm{mid}}, \lceil t/2 \rceil)$ and Reachable$(C_{\mathrm{mid}}, C_f, \lfloor t/2 \rfloor)$
then return 1.

Step 4. return 0.

The algorithm is correct because, if C_f is reachable from C_i in at most t steps, then there must be a configuration C' such that C' is reachable from C_i in at most $\lceil t/2 \rceil$ steps and C_f is reachable from C' in at most $\lfloor t/2 \rfloor$ steps. The *for loop* in Step 3 tries all possible values for C_{mid}. Clearly, when C_{mid} assumes the value C', both recursive calls will return 1 and the final result returned will be 1. It is also easy to see that if the procedure returns 1, then, in fact, C_f is reachable from C_i.

Since A uses $O(\log n)$ space, if C_i is reachable from C_f, then there must be a path of length at most n^c. Thus, to determine if $x \in L$, we invoke the procedure Reachable with parameters (C_s, C_t, n^c).

Let us now estimate the amount of space used by the algorithm. In any invocation of the procedure, there are five local variables, each needing $O(\log n)$ bits. Furthermore, since t is roughly halved in each step, the depth of recursion is at most $\log(n^c) = O(\log n)$. Thus, the total space required at any stage is at most $O(\log n) \times O(\log n) = O((\log n)^2)$. ∎

Corollary 2.4 *NLog is a proper subset of PSPACE.*

2.2 NSpace versus co-NSpace

In our discussion above the problem of direct (s,t)-connectivity plays an important role in the study of space bounded computation. Formally, this problem corresponds to the following language.

$$\text{DSTConn} = \{\langle G, s, t\rangle : \text{there is a path from } s \text{ to } t \text{ in } G\}.$$

Here G is a *directed* graph, and the path sought is a directed path.

Theorem 2.5 *DSTConn* $\in$ *NLog.*

Proof. Fix $\langle G, s, t\rangle$. A non-deterministic log-space algorithm can start at s and successively guesses the next vertex n the path, verifying each time that the new vertex is reachable from the old by traversing one edge. If at any stage it encounters vertex t, it accepts and stops. On the other hand, if after n (here $n = |V(G)|$) steps, vertex t has not been encountered, then it rejects and stops. Since at any time, the names of at most two vertices are recorded in the algorithm's work-space, we can implement this using $O(\log n)$ space. ∎

Theorem 2.6 *DSTConn* $\in$ *co-NLog.*

Proof. We need to show that the following language is in NLog.

$$L = \{\langle G, s, t\rangle : t \text{ is not reachable from } s\}.$$

We will show this in two steps. Suppose we have access to a function, count(i), that gives the number of vertices reachable from s in at most i steps. Our first, goal is to show that L can be recognized by a non-deterministic log-space algorithms using this function. Later, we will show how we can compute count(i) in non-deterministic log-space.

```
Procedure CantReach-t
{i ← 0;
 For v ∈ V(G) do
 {guess and verify that there is a path from s to v of length at most n
  if successful then {i ← i + 1;
                      if v = t then reject and stop
                      }
  }
 if i = count(n) then accept and stop
  else reject and stop
}
```

If t is not reachable from s, then for some sequence of guesses all vertices reachable from s will be discovered in the *for* loop. In that case, i will become equal to count(n) and the algorithm will accept immediately after coming out of the algorithms. On the other hand, if the algorithm accepts, then $i =$ count(n) at the end of the for loop. That is, all vertices reachable from s were, in fact, discovered in the loop. Since the algorithm did not reject inside the loop, none of these vertices was t. Hence, t is not reachable from s. We have thus shown that the algorithm accepts iff t is not reachable from s. But we still need to show how we will compute count(i).

Now, we consider the function count(i). In fact, we will not be able to give a method for computing count(i) in the usual sense. What we will provide is a non-deterministic procedure, pseudocount(i), with the following property: pseudocount(i) either rejects and stops or returns a value; if it does return a value, it will be correct (that is, equal to count(i)). We have the follow observations about count(i). First, count$(0) = 1$. Next, if count$(i-1)$ is known, then count(i) can be computed (in the sense outlined above) in non-deterministic log-space. Assume that the constant C' has been set to count$(i-1)$.

```
C = 0
for v ∈ V(G) do
{ j ← 0
```

```
  for w ∈ V(G) do
  { guess path of lenght at most i − 1 from s to w
    If successful then {j ← j + 1
                        if v is reachable from w in one step then
                        { C ← C + 1; next v }
                        }
  }
  if j ≠ C' then reject
}
return C
```

Thus, to compute count(i) in general, the procedure pseudocount(i) starts with count$(0) = 1$, computes count(1) from it, then count(2) from that and so on until count(i) has been computed. Note that while computing count(j) the procedure needs to know only count($j-1$), and none of the previous values. Thus the entire procedure can be implemented in log-space. ∎

Theorem 2.7 (Immerman-Szelepcsényi) *NLog = co-NLog.*

Proof. Similar to the proof of Theorem 2.6. Omitted. ∎

Lecture 3

Reductions, Completeness, Relativization

Lecturer: Jaikumar Radhakrishnan **Date:** 21 January 1998

We observed an interesting property of the class $\mathcal{P}$. If a language is in $\mathcal{P}$, then its complement $\overline{L}$ is also in $\mathcal{P}$. Its proof rested on the following reasoning: to decide if an input x is in $\overline{L}$, we may invoke the algorithm for deciding membership in L and invert the answer it returns. In other words, we showed how one can obtain an efficient algorithm for the language $\overline{L}$ using the efficient algorithm for the language L. This is, indeed, a special case of a general method.

3.1 Reductions

Imagine we have a program A that uses a subroutine B. Further, suppose that A runs in polynomial time (calls to the subroutine B take one unit of time and the answers appear immediately), and if the subroutine B recognizes the language L, then A itself recognizes the language L'. That is, we have an efficient method for recognizing the language L' provided we are allowed to ask membership questions for the language L (which are answered without delay). Now suppose $L \in \mathcal{P}$. Can we then conclude that $L' \in \mathcal{P}$? Yes, for we can run the efficient algorithm B_L for the language L whenever the program A calls the program B and use the answer returned by it. This program clearly constitutes a correct method for recognizing L'. But is it efficient? Surely, A still performs the same computations as before. However, it does not enjoy the luxury of obtaining answers instantaneously; instead, it must wait for B_L to give the correct answer. True, B_L runs in polynomial time, but this is measured in terms of the length of input supplied to it. We must, therefore, verify that the inputs supplied to B_L are not unreasonably long. Let the original running time of A (assuming that the call to B are serviced immediately) be $p(n)$ and the running time of B_L be $q(n)$. Then, A cannot invoke B_L with an input longer than $p(n)$, and on this B cannot take more that $q(p(n))$ units of time. Since the total number of calls to B in the original program is at most $p(n)$, the running time of the new algorithm for recognizing the language L' can be at most $p(n)q(p(n)) + p(n)$, which is easily seen to be bounded by a polynomial.

In complexity theory the programs, such as the program A above, that operate with the assumption that a certain subroutine B is available, are referred to as *Oracle Programs.* The subroutine B is called the *oracle* and the questions that A asks of B are referred to as *queries.* For concreteness, let us fix some of the modalities for invoking the oracle. We shall assume that every oracle program has two special variables Query and Answer. When the oracle is invoked, the query should have been assigned to the variable Query, and as soon as the oracle is invoked the answer ('Yes' or 'No') will appear in the variable Answer.

Oracle programs that enable us to propose solutions for one problem assuming that one exists for another are among the most frequently studied objects in complexity theory. When the oracle program A is supplied the language L via oracle, we denote the resulting algorithm by $\mathsf{A}(L)$. Thus, in our example above, we can say that $\mathsf{A}(L)$ recognizes the language L'. We say that the oracle program A runs in polynomial time if there exists a polynomial $p(n)$ such that for all languages L, the running time of $\mathsf{A}(L)$ is bounded by $p(n)$. Such oracle programs are called *$\mathcal{P}$-oracle programs.*

We may similarly consider nondeterministic and space bounded oracle programs. If A is a nondeterministic oracle program and L is a language then $\mathsf{A}(L)$ is a nondeterministic algorithm. Then the usual criterion for acceptance used for nondeterministic programs applies to A: we say that $\mathsf{A}(L)$ recognizes the language L' if, for all $x \in L'$, there is at least one sequence of nondeterministic choices that causes $\mathsf{A}(L)$ to accept, and, for $x \notin L'$, no sequence of choices can cause $\mathsf{A}(L)$ to accept. If the nondeterministic oracle program A has the property that there is a fixed polynomial that bounds the running time of $\mathsf{A}(L)$ for every language L, then we refer to it as a nondeterministic polynomial time oracle program or *$\mathcal{NP}$-oracle program.*

It will be useful to introduce another form of polynomial time computation, called co-$\mathcal{NP}$-computation. A co-$\mathcal{NP}$ computation proceeds by making guesses just like an $\mathcal{NP}$ computation; the difference lies in the criterion for acceptance. For an $\mathcal{NP}$ computation to accept an input, it suffices that there exist some sequence of guesses leading to acceptance. In contrast, a co-$\mathcal{NP}$ computation accepts the input precisely when all possible computations, induced by the different sequences of guesses, lead to acceptance. If the above acceptance criterion is enforced on an oracle program that makes guesses, then we call it a *co-$\mathcal{NP}$-oracle program.*

Let us consider the space complexity of oracle programs. We say that $\mathsf{A}(L)$ has space complexity $s(n)$ if the maximum number of bits of memory accessed by A on any input of length n is $s(n)$ for all languages L. Note that we count the number of bits used, so programs cannot hide the space used by using large word size. Also, the variable Query is treated as any other variable and the number of bits used to maintain it must be taken into account while considering the space used by the program. We say that A is a polynomial space oracle program, or PSPACE-oracle program if its space complexity is bounded by some polynomial.

Definition 3.1 (Reductions)

- *We say that a language L' is polynomial time (Turing) reducible to the language L if there exists a $\mathcal{P}$-oracle program A such that $\mathsf{A}(L)$ recognizes L'; we denote this by $L' \leq_T^{\mathrm{P}} L$.*
- *We say that a language L' is nondeterministic polynomial time (Turning) reducible to the language L if there exists a $\mathcal{NP}$-oracle program A such that $\mathsf{A}(L)$ recognizes L'; we denote this by $L' \leq_T^{\mathrm{NP}} L$.*
- *We say that a language L' is co-nondeterministic polynomial time (Turing) reducible to the language L if there exists a co-$\mathcal{NP}$-oracle program A such that $\mathsf{A}(L)$ recognizes L'; we denote this by $L' \leq_T^{co\text{-}NP} L$.*
- *We say that a language L' is polynomial space (Turing) reducible to the language L if there exists a PSPACE-oracle program A such that $\mathsf{A}(L)$ recognizes L'; we write this as $L' \leq_T^{\mathrm{PSPACE}} L$.*

When discussing the relationship between language recognition problems, one often considers various restricted kinds of reductions. In *many-one* reductions, the oracle program is allowed to query the oracle exactly once; moreover, the answer returned by the oracle is immediately output (as it is) by the program. Thus, in this case, the entire computation of the program is directed towards generating the one appropriate query to the oracle.

Definition 3.2 *A language L' is polynomial time many-one reducible to the language L (written as $L' \leq_m^{\mathrm{P}} L$) if there exists a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ such that*

$$x \in L' \iff f(x) \in L.$$

Other restricted kinds of reductions are truth-table reductions. They are intermediate in power between Turing reductions and many-one reductions. In a truth-table reduction all the queries have to be presented to the oracle simultaneously, that is, the oracle program cannot decide what queries to ask based on the answers to the previous queries. However, after receiving all the answers to the queries, it can resume its computation and produce the final 'Yes' or 'No'. We write $L' \leq_{\mathrm{tt}}^{\mathrm{P}} L$ to denote that the language L' is polynomial time truth-table reducible to the language L. A truth-table reduction where the number of queries asked is bounded by a constant (independent of the input) is called a bounded truth table reduction and denoted by $\leq_{\mathrm{btt}}^{\mathrm{P}}$.

Proposition 3.3 *1.* $L_1 \leq_T^{\mathrm{P}} L_2 \iff \overline{L}_1 \leq_T^{\mathrm{P}} L_2 \iff L_1 \leq_T^{\mathrm{P}} \overline{L}_2 \iff \overline{L}_1 \leq_T^{\mathrm{P}} \overline{L}_2$.

2. If $L_1 \in \mathcal{P}$ then, for every language L_2, $L_1 \leq_T^{\mathrm{P}} L_2$.

3. If $L_1 \leq_T^{\mathrm{P}} L_2$ and $L_2 \leq_T^{\mathrm{P}} L_3$ then $L_1 \leq_T^{\mathrm{P}} L_3$ (transitivity).

4. If $L_1 \leq_m^{\mathrm{P}} L_2$ and $L_2 \leq_m^{\mathrm{P}} L_3$ then $L_1 \leq_m^{\mathrm{P}} L_3$.

Properties 1 and 2 are easy to verify and left to the reader. To see 3, assume that $\mathsf{A}_1(L)$ recognizes L_1 and $\mathsf{A}_2(L_3)$ recognizes L_2. Consider the oracle program A^* which functions as A_1, and when it needs to invoke the oracle, it instead invokes A_2 on the query. A_2, however, functions as before, and is allowed to consult its oracle. Clearly $\mathsf{A}^*(L_3)$ recognizes L_1. It remains to verify that the running time of A^* is bounded by a polynomial. Since A_1 runs in polynomial time, any input supplied to A_2 will have length bounded by a polynomial in the length of A_1's input. As we saw before, this implies that the running time of the combined algorithm is also bounded by a polynomial. Property 4 can be proved using the same reasoning.

Relativized complexity classes: So far we have considered oracle programs recognizing languages and have developed various notions of reductions based on the power of the oracle programs. Now fix a language L and denote the class of languages recognized by $\mathcal{P}$-oracle programs with access to an oracle for L, by $\mathcal{P}(L)$. That is

$$\mathcal{P}(L) = \{L' \subseteq \Sigma^* : L' \leq_T^{\mathrm{P}} L\}.$$

Similarly,

$$\begin{aligned} \mathcal{NP}(L) &= \{L' \subseteq \Sigma^* : L' \leq_T^{\mathrm{NP}} L\}; \\ \text{co-}\mathcal{NP}(L) &= \{L' \subseteq \Sigma^* : L' \leq_T^{\text{co-NP}} L\}; \\ \mathrm{PSPACE}(L) &= \{L' \subseteq \Sigma^* : L' \leq_T^{\mathrm{PSPACE}} L\}. \end{aligned}$$

The classes obtained by considering computation relative to an oracle are called relativized complexity classes. We may extend this notation by permitting the language L to vary within some class $\mathcal{C}$. This leads to the complexity class

$$\mathcal{P}(\mathcal{C}) = \bigcup_{L \in \mathcal{C}} \mathcal{P}(L),$$

and, similarly,

$$\begin{aligned} \mathcal{NP}(\mathcal{C}) &= \bigcup_{L \in \mathcal{C}} \mathcal{NP}(L); \\ \text{co-}\mathcal{NP}(\mathcal{C}) &= \bigcup_{L \in \mathcal{C}} \text{co-}\mathcal{NP}(L); \\ \mathrm{PSPACE}(\mathcal{C}) &= \bigcup_{L \in \mathcal{C}} \mathrm{PSPACE}(L). \end{aligned}$$

As we have observed earlier a $\mathcal{P}$-oracle program with access to an oracle for a language in $\mathcal{P}$ recognizes a language in $\mathcal{P}$. Thus, using the notation introduced above, we may write $\mathcal{P}(\mathcal{P}) \subseteq \mathcal{P}$, and since $\mathcal{P} \subseteq \mathcal{P}(\mathcal{P})$ (Why?), we may write $\mathcal{P}(\mathcal{P}) = \mathcal{P}$. Many such relationships between the relativized complexity classes can be deduced; we collect some of them below for later reference.

Proposition 3.4 *1.* $\mathcal{P}(\mathcal{P}) = \mathcal{P}$.

2. *PSPACE(PSPACE) = PSPACE.*
3. $\mathcal{P}(A) = \mathcal{P}(\overline{A})$ *and* $\mathcal{NP}(A) = \mathcal{NP}(\overline{A})$.
4. *For all languages A,* $\mathcal{P}(A) \subseteq \mathcal{NP}(A)$.
5. *For all languages A,* $\mathcal{NP}(A) \subseteq PSPACE(A)$.

Part 1 was discussed above; the next two can be proved using similar reasoning. Parts 4 and 5 are instances of *relativizing* statements. We have seen in the last class that $\mathcal{P} \subseteq \mathcal{NP}$ and $\mathcal{NP} \subseteq$ PSPACE. Now, 4 and 5 tell us that these inclusions hold relative to every oracle. To see 4, consult the definition of $\mathcal{P}(A)$ and $\mathcal{NP}(A)$ and observe that for any language L, $L \in \mathcal{P}(A) \Longrightarrow L \in \mathcal{NP}(A)$, because a $\mathcal{P}$-oracle program is also a $\mathcal{NP}$-oracle program (just that it makes no guesses).

To see 5, we first characterize the languages in $\mathcal{NP}(A)$ in a manner similar to the characterization (1.2) of last week. It can be shown using the same reasoning that for every language $L \in \mathcal{NP}(A)$ there exists a predicate $P(x, y)$ that can be computed in polynomial time *with access to an oracle for A* such that for all $x \in \Sigma^*$

$$x \in L \iff \exists^P y \; P(x, y).$$

Alternatively, we may say that for every language $L \in \mathcal{NP}(A)$ there exists a language $L' \in \mathcal{P}(A)$ such that for all $x \in \Sigma^*$

$$x \in L \iff \exists^P y \; (x, y) \in L'.$$

[By the way, if $L \in$ co-$\mathcal{NP}(A)$, then there exists a language $L' \in \mathcal{P}(A)$ such that for all $x \in \Sigma^*$

$$x \in L \iff \forall^P y \; (x, y) \in L'.$$

]

Now, a PSPACE-oracle program with access to A can systematically go through all possible strings y and each time check of $(x, y) \in L'$ (this is feasible because $L' \in \mathcal{P}(A)$ and the program has access to A).

We just saw some statements about complexity classes where the introduction of an oracle did not present any difficulty in adapting the old proofs to the new setting. Whenever a proof has the property that the reasoning used can be applied in the presence of any oracle, we say, informally, that the *proof relativizes.* Most of the proofs in classical computability theory, which are based on simulations or diagonalization, relativize. It is only recently that non-relativizing proofs have been discovered; we will see some of these later in this course.

3.2 Completeness

In this section we shall use many-one reductions only. Reductions permit us to compare the complexity of language recognition problems. For example, if $L_1 \leq_m^{\mathrm{P}} L_2$. then we may say that the language L_2 is at least as hard as the language L_1. It often happens that there are problems to which languages from an entire class reduce. Such languages are called *hard* languages and defined formally as follows.

Definition 3.5 (Hard, Complete) *For a class $\mathcal{C}$, a language L is $\mathcal{C}$-hard if for every language $L' \in \mathcal{C}$, $L' \leq_m^{\mathrm{P}} L$. If, in addition, the language L itself is in $\mathcal{C}$ then we say that L is $\mathcal{C}$-complete.*

Proposition 3.6

1. *If L_1 is $\mathcal{C}$-hard and $L_1 \leq_m^{\mathrm{P}} L_2$ then L_2 is $\mathcal{C}$-hard.*
2. *If L_1 is $\mathcal{C}$-complete, $L_2 \in \mathcal{C}$ and $L_1 \leq_m^{\mathrm{P}} L_2$, then L_2 is $\mathcal{C}$-complete.*
3. *If L is $\mathcal{C}$-hard then $\overline{L}$ is co-$\mathcal{C}$-hard.*
4. *If $\exists L in \mathcal{NP}$-complete such that $L \in \mathcal{P}$ then $\mathcal{NP} \subseteq \mathcal{P}$.*

We will see in our study that complete languages exist for many complexity classes. For example, the language SAT, consisting of satisfiable Boolean expressions, was shown to be $\mathcal{NP}$-complete by S. A. Cook. A logical (or Boolean) variable is a variable that may be assigned the value *true*(or 1) or *false* (or 0). If v is a logical variable, then $\overline{v}$, the negation of v, has the value *true* if and only if v has the value *false*. A *literal* is a logical variable or the negation of a logical variable. A *clause* is a sequence of literals separated by the Boolean **or** operator ($\vee$). A logical expression is said to be *satisfiable* if it evaluates to *true*, under some assignment of values to its variables.

A logical expression in *conjunctive normal form (CNF)* is a sequence of clauses separated by the Boolean **and** operator ($\wedge$). An example of a logical expression in CNF is

$$(p \vee q \vee s) \wedge (\overline{q} \vee r) \wedge (\overline{p} \vee r) \wedge (\overline{r} \vee s) \wedge (\overline{p} \vee \overline{s} \vee \overline{q})$$

where p, q, r and s are logical variables. The language of satisfiable CNF expressions is denoted by CNF-SAT. A CNF expression in which there are exactly three literals per clause is called a 3-CNF expression, and the language of satisfiable 3-CNF expressions is called 3SAT.

Theorem 3.7 (Cook's Theorem) *SAT is $\mathcal{NP}$-complete.*

We shall not prove this theorem in this course. However, in the next class we will discuss certain connections between polynomial time computations and circuits; these will, perhaps, help in making this result less mysterious. It can also be shown that the two restrictions of SAT described above are $\mathcal{NP}$-complete.

Theorem 3.8 *CNF-SAT and 3SAT are $\mathcal{NP}$-complete.*

Historically, SAT was the first language that was shown to be $\mathcal{NP}$-complete; it still occupies the pride of place among the hundreds of problems that have since then been show to be $\mathcal{NP}$-complete. A substantial part of the work done around the $\mathcal{P} =? \mathcal{NP}$ question consists of showing that certain problems are $\mathcal{NP}$-complete and there are entire books devoted to just cataloging $\mathcal{NP}$-complete problems. Many of the $\mathcal{NP}$-completeness proofs require ingenious combinatorial constructions. We will not discuss these methods; however, to get some feeling for what this involves, we shall present an example.

Theorem 3.9 *The independent set problem is $\mathcal{NP}$-complete.*

Proof. Recall that the independent set problem corresponds to the following language.

$$L = \{\langle G, k\rangle \, : \, G \text{ has an independent set of size } k\}.$$

We have already seen in the last class that $L \in \mathcal{NP}$. We shall show that SAT $\leq_m^{\mathrm{P}} L$. Then by appealing to Proposition 3.6 (2), we can conclude that L is $\mathcal{NP}$-complete.

Consider a CNF expression φ and let its clauses be $C_1, C_2, \ldots, C_m$ and let the variables be $x_1, x_2, \ldots, x_n$. For the expression φ, we shall produce the undirected graph G_φ such that $\langle G_\varphi, m\rangle \in L$ iff φ is satisfiable. The graph G_φ is defined as follows.

$$\begin{aligned} V(G_\varphi) &= \{\langle i, l\rangle : l \text{ is a literal in } C_i\}; \\ E(G_\varphi) &= \{(\langle i, l\rangle, \langle i, l'\rangle) : l \text{ and } l' \text{ are distinct literals of clause } C_i\} \\ &\quad \cup \{\{\langle i, l\rangle, \langle j, l'\rangle\} : i \neq j \text{ and } l \text{ and } l' \text{ are complementary literals}\}. \end{aligned}$$

Observe that one can easily produce the representation of the graph G_φ when given the expression φ. It now remains to show two things.

1. $\underline{\varphi \text{ is satisfiable} \Longrightarrow \langle G_\varphi, m\rangle \in L}$: Consider a satisfying assignment of φ. Then in each clause C_i there is a literal l_i that is satisfied by this assignment. But then the set $\{\langle i, l_i\rangle\}_{i=1}^m$ is an independent set of the graph G.

2. $\underline{\langle G_\varphi, m\rangle \in L \Longrightarrow \varphi \text{ is satisfiable}}$: Note that the vertices with the same value for the first component are completely connected. Hence any independent set of G_φ can include at most one vertex for each clause. Since $\langle G_\varphi, m\rangle \in L$, G_φ has an independent set of size at least m; therefore, this independent set must consist of m vertices of the form $\langle i, l_i\rangle$, one for each i. It is easy to see that there is an assignment that satisfies all the l_i (because no two l_i can be complementary). This assignment satisfies φ.

For example, consider the CNF expression $\varphi = C_1 \wedge C_2 \wedge C_3$, where $C_1 = (x_1 \vee \overline{x}_2 \vee \overline{x}_3)$, $C_2 = (\overline{x}_1 \vee x_2 \vee x_3)$ and $C_3 = (x_1 \vee x_2 \vee x_3)$. The graph G_φ is shown in Figure 3.1. The marked nodes of the graph form an independent set, and represent the satisfying assignment $x_1 = 0$, $x_2 = 0$, $x_3 = 1$.

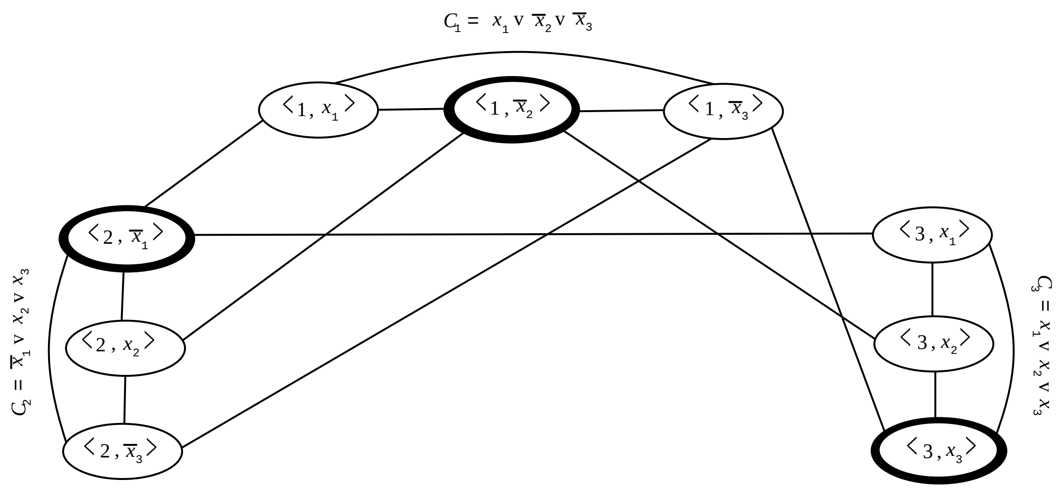

Figure 3.1: The graph G_φ for $\varphi = (x_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$.

∎

3.3 The polynomial time hierarchy

We have observed earlier that $\mathcal{P}(\mathcal{P}) = \mathcal{P}$. What about $\mathcal{NP}(\mathcal{NP})$? Is it equal to $\mathcal{NP}$? While it is easy to see $\mathcal{NP} \subseteq \mathcal{NP}(\mathcal{NP})$, the reverse containment is not clear. Recall that in our proof of $\mathcal{P}(\mathcal{P}) \subseteq \mathcal{P}$ we used the following reasoning: the calls to the oracle of a $\mathcal{P}$-oracle program can be replaced by a direct polynomial time computation because the oracle supplied is also in $\mathcal{P}$.

Let us, then, try and mimic this idea in the case of $\mathcal{NP}(\mathcal{NP})$. Suppose the $\mathcal{NP}$-oracle program A has made some nondeterministic choices, and, based on them, has constructed the query x that it wants to present to the oracle for the language $L \in \mathcal{NP}$. If $x \in L$, then there is an easy to verify proof of this fact. The program A can guess this proof and verify it quickly. No problem! But what if $x \notin L$? The oracle in normal course would have returned a 'No'. Can our $\mathcal{NP}$-program guess and verify this answer on its own? This brings us back to the question of whether co-$\mathcal{NP} \subseteq NP$, which we do not know how to answer. Indeed, it is easily seen that co-$\mathcal{NP} \subseteq \mathcal{NP}(\mathcal{NP})$; thus as long as we cannot prove co-$\mathcal{NP} \subseteq \mathcal{NP}$ it is unlikely that we will succeed in showing that $\mathcal{NP}(\mathcal{NP}) \subseteq \mathcal{NP}$.

We must, therefore, concede that the class $\mathcal{NP}(\mathcal{NP})$ is possibly a much larger class than $\mathcal{NP}$. This class is usually denoted by Σ_2^{P} (one uses Σ_0^{P} for $\mathcal{P}$ and Σ_1^{P} for $\mathcal{NP}$).

A similar analysis can be carried out for co-$\mathcal{NP}$ oracle programs with an $\mathcal{NP}$ oracle. The resulting class is called Π_2^{P}, that is, $\Pi_2^{\mathrm{P}} = \text{co-}\mathcal{NP}(\mathcal{NP})$. If we iterate this procedure, we obtain an entire hierarchy of complexity classes. This is formally defined as follows.

Definition 3.10 (Polynomial time hierarchy) *The polynomial time hierarchy is the structure formed by the classes Σ_k^{P} and Π_k^{P}, for $k \geq 0$, where*

1. $\Sigma_0^{\mathrm{P}} = \Pi_0^{\mathrm{P}} = \mathcal{P}$;
2. $\Sigma_{k+1}^{\mathrm{P}} = \mathcal{NP}(\Sigma_k^{\mathrm{P}})$, *for* $k \geq 0$;
3. $\Pi_{k+1}^{\mathrm{P}} = \text{co-}\mathcal{NP}(\Sigma_k^{\mathrm{P}})$, *for* $k \geq 0$.

The class PH is defined by $PH = \bigcup_{k \geq 0} \Sigma_k^{\mathrm{P}} = \bigcup_{k \geq 0} \Pi_k^{\mathrm{P}}$.

We list below some of the properties of these classes.

Proposition 3.11 *(a)* $\Sigma_k^{\mathrm{P}} \subseteq \Sigma_{k+1}^{\mathrm{P}}$.

(b) $\Pi_k^{\mathrm{P}} \subseteq \Sigma_{k+1}^{\mathrm{P}}$.

(c) $\Pi_k^{\mathrm{P}} = \text{co-}\Sigma_k^{\mathrm{P}}$, *for* $k \geq 0$.

(d) $\Sigma_{k+1}^{\mathrm{P}} = \mathcal{NP}(\Pi_k)$ *for* $k \geq 0$.

(e) $\Pi_{k+1}^{\mathrm{P}} = \text{co-}\mathcal{NP}(\Pi_k^{\mathrm{P}})$ *for* $k \geq 0$.

(f) The classes Σ_k^{P} and Π_k^{P} are closed under polynomial time many-one reductions.

Theorem 3.12 *PH $\subseteq$ PSPACE.*

Proof. It is enough to show that $\Sigma_k^{\mathrm{P}} \subseteq$ PSPACE for all $k \geq 0$. We proceed by induction.

Basis: For $k = 0$, $\Sigma_k^{\mathrm{P}} = \mathcal{P}$ and $\mathcal{P} \subseteq$ PSPACE.

Step: Assume that $r \geq 0$ and $\Sigma_r^{\mathrm{P}} \subseteq$ PSPACE; we shall show that $\Sigma_{r+1}^{\mathrm{P}} \subseteq$ PSPACE. Now,

$$\Sigma_{r+1}^{\mathrm{P}} \subseteq \mathcal{NP}(\Sigma_r^{\mathrm{P}}) \subseteq \mathcal{NP}(\mathrm{PSPACE}) \subseteq \mathrm{PSPACE}(\mathrm{PSPACE}) \subseteq \mathrm{PSPACE},$$

Here the first inclusion claimed is justified using the definition, the second using the induction hypothesis, and the remaining using Proposition 3.4.

∎

We have stated above that all classes in the polynomial time hierarchy are closed under polynomial time many-one reductions. In fact, the class $\mathcal{NP}$ is even closed under nondeterministic polynomial time many-one reductions. That is,

$$L_1 \leq_m^{\mathrm{NP}} L_2 \text{ and } L_2 \in \mathcal{NP} \Longrightarrow L_1 \in \mathcal{NP}.$$

We leave the proof to the reader. Contrast this with nondeterministic polynomial time Turing reductions.

What happens if $L_1 \leq_m^{\mathrm{NP}} L_2$ and $L_2 \in$ co-$\mathcal{NP}$? Since, $L_1 \leq_m^{\mathrm{NP}} L_2$, we have $L_1 \leq_T^{\mathrm{NP}} L_2$; thus, $L_1 \in \mathcal{NP}(\text{co-}\mathcal{NP}) = \Sigma_2^{\mathrm{P}}$. It turns out that for every language L_1 in Σ_2^{P}, there is a language $L_2 \in$ co-$\mathcal{NP}$ such that $L_1 \leq_m^{\mathrm{NP}} L_2$. Although this fact is not important on its own, the proof is instructive. The main idea is as follows. Let $L_1 \in \Sigma_2^{\mathrm{P}}$. Then $L_1 \in \mathcal{NP}(L_2)$, where $L_2 \in \mathcal{NP}$. That is, there is an oracle machine A such that L_1 is recognized by $\mathsf{A}(L_2)$. Let us examine the computation of $\mathsf{A}(L_2)$ more closely. A makes some nondeterministic choices and then based on these presents its first query to the oracle, then based on the answer presents the second query, and so on. We will not allow A to ask questions again and again. So, whenever it needs to ask a question, it will just guess the answer and proceed. This way it will, in the end, have collected a set of queries and the possible answers. If the original program would have rejected with this pattern of answers, then the new program rejects and stops. On the other hand, if the original program would have accepted under these conditions, then we proceed to verify that the answers we guessed were indeed correct. If we had guessed the answer 'Yes' for some query, then the verification is straightforward – we just invoke the nondeterministic algorithm for L_2 (remember $L_2 \in \mathcal{NP}$). What about the queries for whom we guessed the answer 'No'? We will collect all these queries and present them at once to an oracle L^* which will say 'Yes' precisely when all the queries are not in L_2. Now, to complete the proof, we must show why such a language L^* exists in the class co-$\mathcal{NP}$.

$$L^* = \{\langle x_1, x_2, \ldots, x_r \rangle : \text{no } x_i \text{ is in } L_2\}.$$

The reader can convince herself that $L^* \in$ co-$\mathcal{NP}$ (hint: we need to check that each x_i is in $\overline{L}_2$, and $\overline{L}_2 \in$ co-$\mathcal{NP}$).

3.4 Interpreter programs and canonical complete sets

Earlier we considered interpreter programs in separating deterministic classes. These interpreter programs simulated deterministic algorithms on the given input. One can also consider a nondeterministic interpreter program $\mathcal{I}_N$, that takes inputs of the form $\langle \lceil \mathsf{A} \rceil, x \rangle$, where A is a nondeterministic program and $x \in \Sigma^*$. The interpreter $\mathcal{I}_N$ accepts this input iff the A accepts x. Note that $\mathcal{I}_N$ is a nondeterministic program, so when we say it accepts an input, we mean that there exists a sequence of guesses that leads to acceptance. As before, we will assume that the simulation performed by $\mathcal{I}_N$ is efficient.

The existence of efficient interpreter programs enables us to write complete problems for many classes right away. For example,

$$L = \{\langle \lceil \mathsf{A} \rceil, x, 1^t \rangle \ : \ \mathsf{A} \text{ accepts } x \text{ in at most } t \text{ steps}\}$$

is $\mathcal{NP}$-complete. Here 1^t is the string consisting of t 1's. To see that $L \in \mathcal{NP}$, we use the interpreter program to simulate A on x for t steps and accept iff the corresponding computation $\mathsf{A}(x)$ halts in t steps and accepts. Since the simulation is efficient, the running time of the interpreter is bounded by a polynomial in t, and hence a polynomial in the length of the input. Next we see why L is $\mathcal{NP}$-hard. Consider any language in $L' \in \mathcal{NP}$. We wish to show that $L' \leq^{\mathrm{P}}_{m} L$. Since $L' \in \mathcal{NP}$, there exists a nondeterministic program $\mathsf{A}_{L'}$ and a polynomial $p(n)$ such that $\mathsf{A}_{L'}$ recognizes L' in time bounded by $p(n)$. Then, the function $f : \Sigma^* \to \Sigma^*$ defined by

$$f(x) = \langle \lceil \mathsf{A}_{L'} \rceil, x, 1^{p(|x|)} \rangle$$

reduces L' to$L$. It is easy to see that f can be computed in polynomial time.

Similar complete languages can be constructed for other languages. For example, the language

$$\{\langle \lceil \mathsf{A} \rceil, x, 1^s \rangle \ : \mathsf{A} \text{ accepts } x \text{ using at most space } s\}$$

is complete for PSPACE. Such complete sets that exploit the existence of the interpreter program are called *canonical complete sets.*

3.5 $\mathcal{P} =? \mathcal{NP}$ and relativization

Questions about classes of languages have been studied for a long time in Computability Theory. Formidable techniques have been developed there to show inclusion or separation results between these classes. It was, therefore, natural to expect that some adaptation of these methods would help us answer the $\mathcal{P} =? \mathcal{NP}$ question. It was observed that most of the methods developed in computability theory give proofs that work even in the presence of an oracle. We are about to show that by relativization different answers are possible for the $\mathcal{P} =? \mathcal{NP}$ question. Thus the methods that give the same answer for all oracles are not going to help us decide this question.

Theorem 3.13 *There exist languages A and B such that $\mathcal{P}(A) = \mathcal{NP}(A)$ but $\mathcal{P}(B) \neq \mathcal{NP}(B)$.*

Proof. Let A be he canonical PSPACE-complete set defined earlier. Then

$$\text{PSPACE} \subseteq \mathcal{P}(A) \subseteq \mathcal{NP}(A) \subseteq \text{PSPACE}\,(A) = \text{PSPACE}.$$

Here the first inclusions follows from the definition of PSPACE-completeness, the second holds because $\mathcal{P}$-oracle programs are a special case of $\mathcal{NP}$-oracle programs, and the next two from Proposition 3.4. Thus, $\mathcal{P}(A) = \mathcal{NP}(A) = \text{PSPACE}$.

Now we shall show the existence of the language B. Consider an enumeration of deterministic polynomial time-bounded oracle Turing machines that witness the reducibility $\leq^{\mathrm{P}}_{T}$, say $\mathsf{P}_1, \mathsf{P}_2, \ldots$. We lose no generality by assuming that for each i, machine P_i operates in time $p_i(n) = i + n^i$.

For any set $A \subseteq \{0,1\}^*$, let $L(A) = \{x \in \{0,1\}^* :$ there exists $y \in A$ such that $|y| = |x|\}$. It is clear that for every A, $L(A) \in \mathcal{NP}(A)$: on input x a nondeterministic machine need only guess a string y such that $|y| = |x|$, and then query the oracle with y; if the oracle answers 'Yes', then the machine accepts x; otherwise, it rejects.

A set B will be constructed such that $L(B)$ is not in $\mathcal{P}(B)$; hence, $\mathcal{P}(B) \neq \mathcal{NP}(B)$. The construction of B will proceed in stages.

Stage 0. Let $B(0) \leftarrow \emptyset$ and $n_0 \leftarrow 0$.

Stage i. Choose n sufficiently large that $n > n_{i-1}$ and $p_i(n) < 2^n$. Consider the computation of the deterministic oracle machine P_i relative to $B(i-1)$ on input $x_i = 0^n$. If P_i accepts x_i relative to $B(i-1)$, then let $B(i) \leftarrow B(i-1)$ and let $n_i \leftarrow 2^n$. If P_i does not accept x_i relative to $B(i-1)$, then let z be the first string of length n in the sequence $\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots$ such that P_i does not query the oracle about z during this computation. Since there are 2^n strings of length n in $\{0,1\}^*$ and P_i's computation on x_i has at most $p_i(n)$ steps, the choice of n

such that $p_i(n) < 2^n$ guarantees the existence of such a z. In this case, let $B(i) \leftarrow B(i-1) \cup \{z\}$ and let $n_i \leftarrow 2^n$.

Let $B \leftarrow \cup_{i>0} B(i)$. At stage $i > 0$, the choice of n such that $n > n_{i-1}$ and $n_i = 2^n$ guarantees that B has at most one string of length n; more generally, B has at most one string of each length. The choice of $p_i(n) < 2^n$ and $n_i = 2^n$ guarantees that in P_i's computation on x_i relative to $B(i-1)$, any string for which the oracle is queried in this computation will never be added or deleted from B at a later stage of construction. Thus, on input x_i, the computation of P_i is the same relative to B as it is relative to $B(i-1)$; any string added at a later stage is not considered at this stage (because it would be too long).

For each i, P_i rejects x_i relative to B if and only if P_i rejects x_i relative to $B(i-1)$ if and only if there is some string of length $|x_i|$ in B if and only if $x_i \in L(B)$. Thus, for every i, $\mathsf{P}_i(B)$ does not recognize $L(B)$ so that $L(B)$ is not in $\mathcal{P}(B)$. ∎

Remarks

The area of Structural Complexity Theory is devoted to the study of relationships between complexity classes. Everything we have discussed about Structural Complexity Theory – containments, oracles, relativization, hierarchies – is contained in the book of Balcázar, Díaz and Gabarró, *Structural Complexity Theory I.*

Several books on algorithms contain material on $\mathcal{NP}$-completeness. We have taken the description of SAT and Cook's theorem from the book *Computer Algorithms* by Sara Baase. The standard source on $\mathcal{NP}$-completeness is the book *Computers and Intractability: A guide to the Theory of $\mathcal{NP}$-completeness* by Garey and Johnson (see also David Johnson's $\mathcal{NP}$-completeness column in the *Journal of Algorithms*). The proof of Theorem 3.9 was taken from [CLR90] and the proof of Theorem 3.13 from a survey article of Book [Boo89].

Our definition of the hierarchy used oracle programs; unfortunately, this did not give us any means of picturing any form of computation that naturally leads to these classes. It turns out that the classes of the polynomial hierarchy can be obtained by a generalization of the notion of nondeterministic computation, called *alternating computation.*

3.6 Alternation and the hierarchy

We had defined $\mathcal{NP}$ to be the class of those languages that are recognized by polynomial time nondeterministic (guessing) programs. We then characterized $\mathcal{NP}$ as follows: $L \in \mathcal{NP}$ iff there exists a polynomial time computable predicate $R_L(x,y)$ such that, for all $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists^P y \; R_L(x,y). \tag{3.1}$$

We may, using this, write down a program for recognizing L.

1. Input x (let $|x| = n$).
2. Existentially check for $y \in \{0,1\}^{p(n)}$ if $R_L(x,y)$.

Here the Existentially check ... step can described by a computation tree of depth $p(n)$, at each of whose leaves we compute $R_L(x,y)$. The tree accepts iff the computation corresponding to at least one of the leaves accepts. Earlier we had used a sequence of guesses to implement this step; we now use this shorthand to make our notation concise.

A characterization in the same spirit as (3.1) can be obtained for the class co-$\mathcal{NP}$: $L \in$ co-$\mathcal{NP}$ iff there exists a polynomial time computable predicate $Q_L(x,y)$ such that, for all $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \forall^P y \; Q(x,y). \tag{3.2}$$

This leads to the following program for L

1. Input x (let $|x| = n$).
2. Universally check for $y \in \{0,1\}^{p(n)}$ if $Q_L(x,y)$.

The Universally check ... step can be pictured using a computation tree in the same way as we did for the existential check; however, there is a twist in the acceptance criterion—the tree accepts iff *all* the computations at the leaves accept.

We have thus interpreted the classes $\mathcal{NP}$ and co-$\mathcal{NP}$ using programs that are allowed to make existential and universal checks respectively. What if we combine both kinds of checks in the same program? For example, consider the following program, where $R(x,y_1,y_2)$ is a polynomial time computable predicate and $p_1(n)$ and $p_2(n)$ are polynomials.

1. Input x (let $|x| = n$).
2. Existentially check for $y_1 \in \{0,1\}^{p_1(n)}$ if
   Universally check for $y_2 \in \{0,1\}^{p_2(n)}$ if
   $R(x,y_1,y_2)$

Let the language recognized by this program be L. How complex is L? It is not clear anymore that L is contained in either in $\mathcal{NP}$ or co-$\mathcal{NP}$, because we are using both kinds of checks simultaneously. What we know about L is this: for all $x \in \{0,1\}^*$,

$$x \in L \iff \exists^P y_1 \, \forall^P y_2 \; R(x,y_1,y_2).$$

But this exactly matches the requirements for the class Σ_2^{P}, derived in Homework 1, Problem 3. In fact, that problem tells us that every language in Σ_2^{P} can be recognized using a polynomial time program which starts with an existential check, follows it with a universal check, and then computes a polynomial time predicate.

Similarly, one can write programs with more existential and universal checks. Such programs are called *alternating programs* because the computation switches between existential and universal modes. In an $\mathcal{NP}$-program (or a Σ_1^{P}-program) there are only existential checks; in a co-$\mathcal{NP}$-program (or a Π_1^{P}-program) there are only universal checks. In general, a Σ_k^{P} program has at most k checks, starting with an existential check (that is, at most $k-1$ switches of modes). Similarly Π_k^{P} program has k checks starting with a universal check (again at most $k-1$ switches of modes). In particular, Σ_0^{P}-programs and Π_0^{P} programs have no checks at all, and, therefore, are $\mathcal{P}$-programs. Of course, all these definitions assume that the checks are bounded by a polynomial and the predicate used at the end is also polynomial time computable.

Theorem 3.14 *1. $L \in \Sigma_k^{\mathrm{P}}$ iff L is recognized by a Σ_k^{P}-program.*

2. $L \in \Pi_k^{\mathrm{P}}$ iff L is recognized by a Π_k^{P}-program.

Proof. Use Homework 1, Problem 4. ∎

3.6.1 Alternation and PSPACE

In the alternating programs considered above, we had fixed the number of alternations in the beginning; that is, the number of mode changes, or alternations, did not depend on the input. It is natural to consider also those programs that need more and more alternations as the input becomes longer and longer. We refer to these programs as *alternating polynomial time programs.* We have the following remarkable theorem, whose proof we shall not discuss now.

Theorem 3.15 *$L \in PSPACE$ iff L is recognized by an alternating polynomial time program.*

Lecture 4

Circuit Complexity

Lecturer: Jaikumar Radhakrishnan **Date:** 3 February 1998

4.1 The circuit model

A Boolean circuit is a directed acyclic graph. The nodes of indegree 0, called *input nodes*, are labeled with a variable, a negated variable or a constant (1 or 0). The internal nodes called *gates* have indegree two and are labeled with an AND or an OR. One of the nodes of the circuit is designated the *output* node. The *size* of a circuit is the number of gates in it.

A Boolean circuit computes a Boolean function in a natural way. For a function $f : \{0,1\}^n \to \{0,1\}$, the *circuit complexity* of f is the size of the smallest circuit computing f. For $g : \{0,1\}^* \to \{0,1\}$ and $h : \mathbf{N} \to \mathbf{N}$, we say that *$g$ has circuit complexity h* if for all n, $C(g_n) = h(n)$, where g_n is g restricted to $\{0,1\}^n$. The circuit complexity of a language is the circuit complexity of its characteristic function.

In a *monotone* circuit no negated variables are allowed to appear as labels of the input nodes.

4.2 Circuits and $\mathcal{P}$

Theorem 4.1 *If $L \in \mathcal{P}$, then L has polynomial size circuits.*[1]

What about the converse of this theorem? Is it true that all languages that have polynomial size circuits are in $\mathcal{P}$? It is known that there are subsets of $\{1\}^*$ that are not recognized by any program at all. Clearly, these languages cannot be in $\mathcal{P}$. On the other hand, every subset of $\{1\}^*$ has linear size circuits (Why?). Hence, the converse of the above theorem does not hold.

Observe, however, that in the above counterexample, we showed only that a family of small circuits exists; we made no claims about how the circuits are themselves to be obtained. In fact, the family of circuits promised by Theorem 4.1 is quite regular; that is, there exists a polynomial time program that, when presented the input n in unary, will produce the description of the nth circuit in the family. We shall show below that if this condition is imposed on the circuits, then the converse of Theorem 4.1 does hold.

A family of circuits $\{C_1, C_2, \ldots\}$ is said to be $\mathcal{P}$-uniform if the exists a polynomial time program to generate the description of C_n given n in unary.

[1] A formal proof of this theorem will require some technical notions associated with Turing machines. We will, therefore, not describe it here; however, we will describe the main idea of the proof informally. Let P be the $\mathcal{P}$-program recognizing the language L. Now as the program runs it passes through various configurations. Our task then is to check if it is possible for the program to reach the final accepting configuration from the initial configuration determined by the input. We encode the configurations of the program as strings over $\{0,1\}$. Each string has polynomial length and the number of strings is also bounded by a polynomial (because the program runs in polynomial time). It can be shown that each bit of a configuration can be computed from the bits of the previous configuration using a circuit of constant size. Finally, we can put together these small circuits and compute the answer based on the input.

Theorem 4.2 *If L has $\mathcal{P}$-uniform (and hence, polynomial size) circuits then $L \in \mathcal{P}$.*

Proof. Let the $\mathcal{P}$-uniform family of circuits recognizing L be $\{C_1, C_2, \ldots\}$. Assume that the subroutine A generates the description of C_i in polynomial time given input 1^i. Then the following polynomial time program recognizes L.

1. Input x (let $|x| = n$).
2. $C \leftarrow \mathsf{A}(1^n)$.
3. Output $C(x)$.

∎

In light of the above theorems, the class of languages that have polynomial size circuits assumes importance. It has received considerable attention in complexity theory and various characterizations in terms of programs have been discovered for this class. We discuss one of them below; some others are discussed in the homework.

Definition 4.3 *We say that a language L is in $\mathcal{P}$/poly if there exists a sequence of advice $\{a_1, a_2, \ldots\} \subseteq \{0,1\}^*$, a polynomial $p(n)$, and a language $L' \in \mathcal{P}$, such that*

- $\forall n \in \mathbf{N}$, $|a_n| \leq p(n)$.
- $\forall x \in \{0,1\}^*$ $x \in L \iff \langle x, a_{|x|} \rangle \in L'$.

Theorem 4.4 *L is in $\mathcal{P}$/poly iff L has polynomial size circuits.*

Proof. ($\Rightarrow$): Suppose $L \in \mathcal{P}$/poly. Then there exists a sequence of advice $\{a_1, a_2, \ldots\}$ and a language $L' \in \mathcal{P}$ satisfying Definition 4.3. By Theorem 4.1, L' has polynomial size circuits, say $\{C_1, C_2, \ldots\}$. The circuits $\{C_1', C_2', \ldots\}$ for the language L are obtained as follows: C_i' is obtained from $C_{i+|a_i|}$ by presetting the advice a_i.

($\Leftarrow$): We encode the polynomial size circuits for L as advice. Since a polynomial size circuit can be evaluated on any input efficiently, this constitutes a valid advice sequence. ∎

4.3 Circuits and $\mathcal{NP}$

We now turn to the class $\mathcal{NP}$, and derive the following form of Cook's theorem from Theorem 4.1.

Theorem 4.5 (Cook) *If $L \in \mathcal{NP}$, then for each $n \in \mathbf{N}$, there exists a 3-CNF formula $\varphi_n(x,y)$, obtainable in polynomial time given n in unary, such that*

$$x \in L \iff \exists y \ \varphi_{|x|}(x,y).$$

Proof. Since L is in $\mathcal{NP}$, there exists a language $L' \in \mathcal{P}$ and a polynomial $p(n)$ such that

$$x \in L \iff \exists y \in \{0,1\}^{p(n)} \langle x, y \rangle \in L'.$$

Now, since $L' \in \mathcal{P}$, it has $\mathcal{P}$-uniform circuits, say $\{C_1, C_2, \ldots\}$. Thus, for all n, and all $x \in \{0,1\}^n$,

$$x \in L \iff \exists y \in \{0,1\}^{p(n)} \ C_{n+p(n)}(\langle x, y \rangle).$$

We are almost done. The only problem is that $C_{n+p(n)}$ is a circuit and not a 3-CNF expression. We next show how this circuit can be turned into a 3-CNF expression $\varphi_n(x, y, y')$ (where y' are some new variables) such that, for all $x \in \{0,1\}^n$ and $y \in \{0,1\}^{p(n)}$,

$$C_{n+p(n)}(\langle x, y\rangle) \iff \exists y' \in \{0,1\}^{p'(n)} \ \varphi_n(x, y, y'). \tag{4.1}$$

Observe that the theorem will follow from this by combining y and y' to form a single string taking values in $\{0,1\}^{p(n)+p'(n)}$.

Now it remains only to show (4.1). For each node g of the circuit $C_{n+p(n)}$ we introduce a new variable y'_g. Further with each node g of the circuit we associate a Boolean expression φ_g as follows. If g is an AND or an OR gate then the expression φ_g ensures that the variables corresponding to the two gates g_1 and g_2, immediately preceding g, and the variable corresponding to the gate g are mutually consistent. Note that this condition depends on just the three variables y'_g, y'_{g_1} and y'_{g_2}. Hence, we may write φ_g as a 3-CNF Boolean expression involving these three variables. For example, if g is an AND gate, then

$$\varphi_g \equiv (y'_g \to (y'_{g_1} \wedge y'_{g_2})) \wedge ((y'_{g_1} \wedge y'_{g_2}) \to y'_g) \equiv (\overline{y'_g} \vee y'_{g_1}) \wedge (\overline{y'_g} \vee y'_{g_2}) \wedge (\overline{y'_{g_1}} \vee \overline{y'_{g_2}} \vee y'_g).$$

If g is an input node with literal l as label, then the condition φ_g ensures that $y'_g = l$. That is

$$\varphi_g \equiv (\overline{y'_g} \vee l) \wedge (\neg l \vee y'_g).$$

Finally, let

$$\varphi_n(x, y, y') = y'_{g_0} \wedge \bigwedge_g \varphi_g,$$

where g_0 is the output node of the circuit. Clearly, φ_n is in CNF with at most three variables per clause; we can easily ensure that each clause has exactly three variables (How?). What is more important, $\exists y' \ \varphi_n(x, y, y')$ iff $C_{n+p(n)}(\langle x, y\rangle)$. In fact, if $C_{n+p(n)}(\langle x, y\rangle)$ then there exists a unique y' such that $\varphi_n(x, y, y')$ (Why?). ∎

4.4 Circuits and $\mathcal{P} =? \mathcal{NP}$

We saw in Theorem 4.1 that every language in $\mathcal{P}$ has polynomial size circuits. Thus, by showing that some language in $\mathcal{NP}$ does not have polynomial size circuits, we would succeed in showing that $\mathcal{P} \neq \mathcal{NP}$. The circuit model does appear to be more static and easy to reason about than programs. It is, therefore, hoped that some argument based on circuits will show that $P \neq \mathcal{NP}$. However, as things stand today, this goal is far from being realized. No function in $\mathcal{NP}$ has been shown to require superlinear circuit size!

On the other hand, it is conceivable that $\mathcal{NP}$ has polynomial size circuits, even though $\mathcal{P} \neq \mathcal{NP}$. Such circuits cannot, of course, be $\mathcal{P}$-uniform. Theorem 4.7 below shows that if $\mathcal{NP}$ has polynomial size circuits then the polynomial hierarchy collapses. It is, therefore, believed that $\mathcal{NP}$ does not have polynomial size (uniform or non-uniform) circuits.

We need the following important lemma.

Lemma 4.6 (Self reducibility) *There exists a polynomial time program CircuitChecker that takes a 3-CNF formula φ, $|\varphi| = n$, and a circuit C with n inputs, such that*

- *If C recognizes 3SAT for inputs of length n and φ is satisfiable, then CircuitChecker accepts.*
- *If φ is not satisfiable then CircuitChecker rejects.*

Proof. The main idea is to repeatedly use the circuit and extract a satisfying assignment for φ. If the circuit is good then we will succeed in extracting the assignment. If we fail to get the satisfying assignment we will reject.

Program CircuitChecker

1. Input φ, C. Let the variables of φ be $x_1, x_2, \ldots, x_m$.
2. If $C(\varphi) = 0$ then *reject*.
3. $\psi \leftarrow \varphi$.
4. For $i = 1$ to m do
   { if $C(\psi|_{x_i=0}) = 1$ then $a_i \leftarrow 0$ else $a_i \leftarrow 1$; $\psi \leftarrow \psi[x_i \leftarrow a_i]$ }
5. If $\varphi[x_1 \leftarrow a_1; x_2 \leftarrow a_2; \ldots; x_m \leftarrow a_m] = 1$ then *accept* else *reject*.

∎

Theorem 4.7 *If $\mathcal{NP}$ has polynomial size circuits then $PH = \Sigma_2^{\mathrm{P}}$.*

Proof. We shall use the following consequence of Homework 1, Problem 6(b): if $\Sigma_2^{\mathrm{P}} = \Sigma_3^{\mathrm{P}}$, then $\Sigma_2^{\mathrm{P}} = \mathrm{PH}$. Thus it is sufficient to show that $\Sigma_3^{\mathrm{P}} \subseteq \Sigma_2^{\mathrm{P}}$. Let L be a language in Σ_3^{P}. Then, as discussed earlier, L is recognized by a Σ_3^{P}-program, say

Program P1

1. Input x ($|x| = n$).
2. Existentially check for $y_1 \in \{0,1\}^{p_1(n)}$ if
   Universally check for $y_2 \in \{0,1\}^{p_2(n)}$ if
   Existentially check for $y_3 \in \{0,1\}^{p_3(n)}$ if
   $R(x, y_1, y_2, y_3)$.

Let us isolate from this program the last check.

Program Q

1. Input x, y_1, y_2 ($|x| = n$, $|y_1| = p_1(n)$, $|y_2| = p_2(n)$).
2. Existentially check for $y_3 \in \{0,1\}^{p_3(n)}$ if
   $R(x, y_1, y_2, y_3)$.

This is an $\mathcal{NP}$-program; hence (by Theorem 3.14) the language it recognizes is in $\mathcal{NP}$. In particular, by Cook's theorem, it is many-one reducible to 3SAT. That is, every input to this program of the form $\langle x, y_1, y_2 \rangle$ can be transformed efficiently to a 3-CNF expression φ_{x,y_1,y_2} whose membership in 3SAT determines if the program Q accepts $\langle x, y_1, y_2 \rangle$. Thus we may rewrite program P1 as

Program P2

1. Input x ($|x| = n$).
2. Existentially check for $y_1 \in \{0,1\}^{p_1(n)}$ if
   Universally check for $y_2 \in \{0,1\}^{p_2(n)}$ if
   Accept iff $\varphi_{x,y_1,y_2} \in 3SAT$.

Now since 3SAT is in $\mathcal{NP}$, using the assumption of the theorem, we conclude that it has polynomial size circuits. Thus there exists circuit C of polynomial size that determines if $\varphi_{x,y_1,y_2} \in SAT$. (We may assume that $|\varphi_{x,y_1,y_2}|$ depends only on the lengths of x, y_1 and y_2, and not on their actual values. (Why?)) Thus to check the last step in the above program we can use the circuit C. But we do not know C. Let us guess C in the beginning. But our guess could be wrong, causing us to accept formulas that are not satisfiable. Here we will use the Lemma 4.6 and invoke the CircuitChecker program. It will ensure that we never accept expressions that are not satisfiable. Thus we have the following program for recognizing the language L.

Program P3

1. Input x ($|x| = n|$).
2. Existentially check for circuits C if
   Existentially check for $y_1 \in \{0,1\}^{p_1(n)}$ if
   Universally check for $y_2 \in \{0,1\}^{p_2(n)}$ if
   CircuitChecker($C, \langle x, y_1, y_2\rangle$).

Note that we existentially check (guess) circuits only of some polynomial size in the first step; the size is determined by the length of φ_{x,y_1,y_2} and the bound on the circuit complexity of 3SAT given by our assumption. It is easy to verify that this programs recognizes L, and that each of the steps used is efficient. We omit the details.

But this is a Σ_2^{P}-program. Hence $L \in \Sigma_2^{\mathrm{P}}$. ∎

4.5 Razborov lower bound for monotone circuit size

As remarked earlier, in spite of numerous attempts, no non-trivial lower bounds are known for any problem in $\mathcal{NP}$. Hence, it is natural to try and show good lower bounds when the circuits are restricted in a certain way. By considering the restricted model of monotone circuits, the Russian scientist A. A. Razborov showed that the Clique function cannot be solved using polynomial size monotone circuits. The clique function, $\mathrm{CLIQUE}_{k,n}$ has $\binom{n}{2}$ variables, one for each potential edge in a graph on n vertices, and outputs 1 precisely when the graph has a clique (complete subgraph) on k vertices. The language associated with this function is the following.

$$L_{\text{clique}} = \{\langle G, k\rangle : \ G \text{ has a clique of size } k \ \}.$$

It is easy to see that L is in $\mathcal{NP}$. Razborov's result shows that L does not have polynomial size monotone circuits.

Remarks

Alternating computation was introduced by Chandra, Kozen and Stockmeyer [CKS81]; a detailed treatment is also available in [BDG87b]. The relation between standard complexity classes and circuit complexity is discussed in [BDG87a, Chap. 5].

Before Razborov's theorem, no superlinear lower bounds were known for the monotone circuit complexity of some explicit monotone function in $\mathcal{NP}$. Razborov [Raz85a] showed a superpolynomial ($n^{\Omega(\log n)}$) lower bound for the clique function. This was strengthened by Alon and Boppana [AB87] to $\exp(\Omega((n/\log n)^{1/3}))$. When these result first appeared, it was considered possible to approach the $\mathcal{P} =? \mathcal{NP}$ question using circuit complexity. *If it could be shown that all monotone problems in $\mathcal{P}$ had polynomial (or subexponential) sized monotone circuits, then* *one could conclude from the above that $\mathcal{P} \neq \mathcal{NP}$.* It turned out that this hope was misplaced; Razborov [Raz85b] showed a lower bound for the monotone circuit complexity of perfect matching problem on bipartite graphs. Tardos [Tar88] showed an exponential lower bound for the monotone complexity for another monotone problem in $\mathcal{P}$.

Homework 1

Date: 3 February 1998 **Due date:** 3 March 1998

Note.

(a) The phrase $\mathcal{NP}$-machine (similarly $\mathcal{P}$-machine) will denote a nondeterministic program (respectively a deterministic program) whose running time is bounded by a polynomial.

(b) The word 'set' will denote a language i.e. a subset of $\{0,1\}^*$.

(c) For $x \in \{0,1\}^*$, $w(x)$ denotes the number of 1's in x.

Problems.

1. Suppose SAT $\in \mathcal{P}$. Show that there is a polynomial time algorithm that when given a satisfiable expression ϕ produces a satisfying assignment of ϕ.

2. ULog is a subclass of NLog defined as follows. A language L is in ULog iff there is a nondeterministic log-space program A that recognizes L, and furthermore, for $x \in L$, there is a *unique* accepting computation of A. A directed graph G with a distinguished vertex s is singular if for every vertex v reachable from s, there is a unique minimum length (directed) path from s. Let Singular denote the set of singular graphs. Show that Singular $\in$ ULog.

   Hint: Use the method of the Immerman-Szelepcsényi theorem.

3. For a list S of integers, let mode(S) be the element that appears the most number of times in S. Show that there is a log-space algorithm that on receiving S prints 'no' if mode(S) appears at most $\lfloor n/2 \rfloor$ times, and prints 'yes' and mode(S) if mode(S) appears at least $\lfloor n/2 \rfloor + 1$ times. Show that this can be done *simultaneously* in log-space and linear time!

4. (a) Recall from Lecture 1 that $L \in \mathcal{NP}$ iff there is a polynomial time computable 2-input predicate R and a polynomial $p(n)$ such that for all $x \in \{0,1\}^*$,

   $$x \in L \Leftrightarrow \exists y(|y| = p(|x|) \text{ and } R(x,y)).$$

   Show that $L \in$ co-$\mathcal{NP}$ iff there is a polynomial time computable 2-input predicate R and a polynomial $p(n)$ such that for all $x \in \{0,1\}^*$,

   $$x \in L \Leftrightarrow \forall y \in \{0,1\}^{p(|x|)}\, R(x,y).$$

   (b) For any positive integer k, show that $L \in \Sigma_k^{\mathrm{P}}$ iff there is a polynomial time computable $(k+1)$-input predicate R such that, for all $x \in \{0,1\}^*$,

   $$x \in L \Leftrightarrow \exists^P y_1 \forall^P y_2 \exists^P y_3 \ldots \mathrm{Q}^P y_k\; R(x, y_1, ...y_k).$$

   (c) Show that $L \in \Pi_k^{\mathrm{P}}$ iff there is a polynomial time computable $(k+1)$-input predicate R such that, for all $x \in \{0,1\}^*$,

   $$x \in L \Leftrightarrow \forall^P y_1 \exists^P y_2 \forall^P y_3 \ldots \mathrm{Q}^P y_k\; R(x, y_1, ...y_k).$$

   Hint: Show (b) and (c) together by induction on k.

5. (a) For any set T and positive integer k, let T^k denote the set $\{\langle x_1, x_2, ..., x_k\rangle : \forall i\ x_i \in T\}$. We call a set T a *tally set* if $T \subseteq \{0\}^*$. Show that there is a tally set T' such that $T^k \leq_m^{\mathrm{P}} T'$.

   (b) Suppose $A \leq_{\mathrm{btt}}^{\mathrm{P}} T$ for a tally set T. Show that then $A \leq_m^{\mathrm{P}} T'$ for some tally set T'.

   (c) Show that if $A \leq_T^{\mathrm{P}} T$ for some tally set T, then $A \leq_{\mathrm{tt}}^{\mathrm{P}} T'$ for some tally set T'.

6. (a) A class $\mathcal{C}$ is said to be $\mathcal{NP}$-closed if it contains $\mathcal{NP}$ and is closed under nondeterministic polynomial time reductions. Let $\mathcal{C}_*$ be the intersection of all $\mathcal{NP}$-closed classes. Show that $\mathcal{C}_*$ is NP-closed and $\mathcal{C}_* = \mathrm{PH}$.

   (b) Show that if for any positive integer k, $\Sigma_k^{\mathrm{P}} \subseteq \Pi_k^{\mathrm{P}}$, then $\mathrm{PH} = \Sigma_k^{\mathrm{P}} = \Pi_k^{\mathrm{P}}$, i.e. the polynomial hierarchy collapses to its k-th level.

7. Let C be a circuit consisting of AND, OR and NOT gates. Show that there is an equivalent circuit C' (that is, C and C' compute the same function) such that $\mathrm{size}(C') \leq 2\mathrm{size}(C)$, $\mathrm{depth}(C) \leq \mathrm{depth}(C')$ and in C' all NOT gates appear at the bottom.

8. (a) A Boolean function $f : \{0,1\}^* \to \{0,1\}$ is called a *slice* function whenever there is a function $k(n)$ with the following property : if $w(x) > k(|x|)$, then $f(x) = 1$; if $w(x) < k(|x|)$, then $f(x) = 0$; otherwise $f(x)$ is either 0 or 1. Show that a slice function has polynomial size circuits iff it has polynomial size *monotone* circuits.

   Hint : Show and use the fact that sorting of binary bits can be done using polynomial size monotone circuits.

   (b) Show that there is a slice function whose characteristic language is $\mathcal{NP}$-complete. (The characteristic language of f means the set of all the x for which $f(x) = 1$.)

   [Together these two problems show that the question of whether every $\mathcal{NP}$ language has polynomial-size circuits can be reduced to a similar question about monotone circuits.]

9. Show that the class of languages recognized by *games against nature* is exactly the class PSPACE.

10. Show that (1) a set has polynomial size circuits iff (2) it is polynomial time Turing reducible to a sparse set iff (3) it is polynomial time truth table reducible to a tally set.

    Hint : Show that (1) $\Longrightarrow$ (3) $\Longrightarrow$ (2) $\Longrightarrow$ (1). Use the fact that circuits can be efficiently encoded as strings in $\{0,1\}^*$.

Lecture 5
Primality and Randomization

Lecturer: Jaikumar Radhakrishnan **Date:** 10 February 1994

5.1 Primality

Prime numbers have been studied for centuries; today they play an important role in coding theory and cryptography. Given a number in binary, how hard is it to determine if it is a prime? That is, we want to know how hard it is to recognize the following language.

$$\text{PRIMES} = \{n \in \mathbf{N} : \ n \text{ is a prime}\}.$$

One method of determining if a number n is a prime, suggested by the definition of prime, is to use trial division. That is, for $k = 1, 2, \ldots, \sqrt{n}$, check whether $k \mid n$. Unfortunately, this cannot be called an efficient method, because the size of the input is only $\lceil \log n \rceil$ and, therefore, $\sqrt{n}$ is exponential in the size lenght of the input. In fact, it is not known if PRIMES $\in \mathcal{P}$, so we do not know if polynomial time deterministic algorithms exist for this problem. What about nondeterministic programs?

Consider the following nondeterministic program.

> Input: n coded in binary.
> Existentially check for all $n' \in \{2, 3, \ldots, n-1\}$ if $n' \mid n$.

If n is *not* a prime then the above algorithm accepts (on some computation path); if n is a prime then the algorithm rejects (on all computation paths). Now it is easily seen that the numbers $2, 3, \ldots, n-1$ can each be coded using strings of length no more than the length of n. Moreover, division is a polynomial time operation. Thus the above algorithm shows that the language of composite numbers is in $\mathcal{NP}$. Thus compositeness has easy to verify proofs. Hence, we have the following theorem.

Theorem 5.1 *PRIMES* $\in$ *co-*$\mathcal{NP}$*.*

It is not so clear, however, that easy to verify proofs exist for primality also. Yet, the following remarkable theorem shows that 'every prime has succinct certificate'.

Theorem 5.2 *PRIMES* $\in \mathcal{NP}$*.*

Before we prove this, we will need to develop some background in elementary number theory. Fix $n \in \mathbf{N}$, and consider arithmetic modulo n. That is, our numbers will be $Z_n^* = \{0, 1, \ldots, n-1\}$, and addition and multiplication will be performed modulo n. Z_n^* will denote the set $\{1, 2, \ldots, n-1\}$.

Now let $a \in Z_n^*$, and compute its successive powers $a, a^2, \ldots, a^{n-1}$ (modulo n, of course). If $\{a, a^2, \ldots, a^{n-1}\} = Z_n^*$, that is, if all the elements of Z_n^* can be generated in this manner, then we refer to a as a *generator* of Z_n^* and say that Z_n^* is *cyclic*. For example, let $n = 11$. Then for $a = 2$, the sequence of powers is $(2, 4, 8, 5, 10, 9, 7, 3, 6, 1)$; hence 2 is a generator modulo n. However, for $a = 4$, the sequence of powers is $(4, 5, 9, 3, 1)$; hence 4 is not a generator. Note that if a is generator then $a^{n-1} \equiv 1 \pmod{n}$. We shall need the following fact from number theory, a proof of which can be found in any standard text on number theory.

Lemma 5.3 *Z_n^* is cyclic iff n is prime.*

Imagine that someone wants to convince us that the number n is prime. To be sure we ask for a generator a for Z_n^* and check that $a^{n-1} \equiv 1 \pmod{n}$. Note that we cannot compute a^{n-1} by repeated multiplication (remember $|n| = \lceil \log n \rceil$). So we need to be more clever. We will use the method of squaring described in the following algorithm.

Powering(a,b,n)
  $c \leftarrow 0$
  $d \leftarrow 1$
  let $\langle b_k, b_{k-1}, \ldots, b_0 \rangle$ be the binary representation of b
  for $i \leftarrow k$ downto 0
    do $d \leftarrow (d \cdot d) \pmod{n}$
      if $b_i = 1$
        then $d \leftarrow (d \cdot a) \pmod{n}$
  return d

The reader can convince herself of the correctness of this procedure. Note that the number of multiplications performed is at most $|n|$ and the numbers are never allowed to become large because we reduce them modulo n immediately after each multiplication. Thus, we easily verify that $a^{n-1} \equiv 1 \pmod{n}$. But is this enough? No. Consider $n = 15$, $a = 4$; check that $a^{14} \equiv 1 \pmod{15}$. Nevertheless, 15 is not a prime, and 4 is not a generator (because $4^2 \equiv 1 \pmod{15}$). In this counterexample, a started cycling much earlier than 14 steps. So we need to watch out for this. Let $\mathrm{ord}(a)$ be the smallest number (> 0) for which $a^s \equiv 1 \pmod{n}$. Note that if $a^{n-1} \equiv 1 \pmod{n}$, then $\mathrm{ord}(a) \mid n - 1$. Hence all we need to check is that $a^s \not\equiv 1$ for all s that are proper divisors of $n - 1$.

We thus have the following lemma (Why?).

Lemma 5.4 *Let $p_1^{e_1} p_2^{e_2} \ldots p_r^{e_r}$ be the prime decomposition of $n - 1$. Let $n_i = (n-1)/p_i$, for $i = 1, 2, \ldots, r$. Suppose $a^{n-1} \equiv 1 \pmod{n}$ and for $i = 1, 2, \ldots, r$, $a^{n_i} \not\equiv 1 \pmod{n}$. Then a is a generator of Z_n^*.*

This will take us further. We will ask for the prime decomposition of $p_1^{e_1} p_2^{e_2} \ldots p_r^{e_r}$ of $n - 1$, and for each i we check that $a^{n_i} \not\equiv 1 \pmod{n}$, where $n_i = (n-1)/p_i$. Are we done now? No. What guarantee do we have that the prime decomposition provided to us is correct? True, we can easily check that $p_1^{e_1} p_2^{e_2} \ldots p_r^{e_r} = n - 1$, but how do we ensure that the p_i's are primes. We use the same algorithm; that is, we recurse! Stated formally, we have the following nondeterministic algorithm for PRIMES.

Input n.
if $n = 2$ then *accept*
if $n = 1$ or n is even (greater than 2), then *reject*
if n is odd and greater than 2
  then guess $a \in \{2, 3, \ldots, n-1\}$ and verify that
    $a^{n-1} \equiv 1 \pmod{n}$

guess a prime factorization $p_1^{e_1}p_2^{e_2}\dots\ p_r^{e_r}$ for $n-1$
for each i recursively check if p_i is a prime
check that $n-1\ =\ p_1^{e_1}p_2^{e_2}\dots\ p_r^{e_r}$
for each i check that $a^{n_i}\ \not\equiv\ 1 \pmod{n}$
if all these conditions hold then accept.

We leave it to the reader to verify that this algorithm runs in polynomial time.

5.2 Randomized computation

When we introduced nondeterminism, we assumed that our program has the ability to make guesses. However, we assumed nothing about the pattern of guesses or their likelihood – each was equally valid. Nor did we demand that for the input to be accepted the number of guess sequences leading to acceptance must be large - even if just one among the many guesses led to acceptance, we were satisfied.

Now we change our viewpoint. Suppose the bits guessed by the program are random. That is, each guess is 0 or 1 with probability 1/2, independent of other guesses. Thus each sequence of guesses has a probability associated with it. Such algorithms are called *randomized algorithms*. For randomized algorithms, we may talk of the probability that the program accepts the input, that is, the sum of the probabilities of the guess sequences leading to acceptance. Thus it follows from Theorem 5.1 that there is a polynomial time randomized algorithm A such that

- if $x \in$ PRIMES then $\Pr[A \text{ accepts } x] = 1$;
- if $x \notin$ PRIMES then $\Pr[A \text{ accepts } x] < 1$,

and from Theorem 5.2 that there exists a randomized algorithm B such that

- if $x \in$ PRIMES then $\Pr[B \text{ accepts } x] > 0$;
- if $x \notin$ PRIMES then $\Pr[B \text{ accepts } x] = 0$,

Consider algorithm A. Is it a good algorithm? It differentiates between primes and composites. But can we really rely on its verdict? It is possible, for example, that the program accepts composite numbers of length n with probability $1 - 1/2^n$. Now suppose the algorithm accepts an input of length n. Can we be reasonably certain the number is prime? No, because the algorithm accepts composites also with very high probability. For a randomized algorithm to be useful, the acceptance probabilities for the inputs in the language and the inputs not in the language must be well separated. In fact, there do exist algorithms that accept primes and composites with vastly differing probabilities. Before we state this precisely, let us formally define the randomized complexity classes.

Definition 5.5 *A language L is in $\mathcal{BPP}$ (bounded error probabilistic polynomial time) iff there exists a polynomial time randomized algorithm* A *such that*

- *For all $x \in L$,* $\Pr[\mathsf{A} \textit{ accepts } x] \geq 3/4$;
- *For all $x \notin L$,* $\Pr[\mathsf{A} \textit{ rejects } x] \geq 3/4$;

A language L is in $\mathcal{RP}$ (randomized polynomial time) iff there exists a polynomial time randomized algorithm A *such that*

- *For all* $x \in L$, $\Pr[\mathsf{A}$ *accepts* $x] \geq 3/4$;
- *For all* $x \notin L$, $\Pr[\mathsf{A}$ *rejects* $x] = 1$;

A language L *is in* $\mathcal{PP}$ *iff there exists a polynomial time randomized algorithm such that*

- *For all* $x \in L$, $\Pr[\mathsf{A}$ *accepts* $x] > 1/2$;
- *For all* $x \notin L$, $\Pr[\mathsf{A}$ *accepts* $x] \leq 1/2$;

Thus if a language L is in co-$\mathcal{RP}$ then there exists an algorithm A such that

- For all $x \in L$, $\Pr[\mathsf{A}$ accepts $x] = 1$;
- For all $x \notin L$, $\Pr[\mathsf{A}$ rejects $x] \geq 3/4$.

The following is a direct consequence of our definitions.

Proposition 5.6 $\mathcal{P} \subseteq \mathcal{RP} \subseteq \mathcal{NP} \cap \mathcal{BPP}$.

In the next class we will study the properties of these randomized complexity classes and their relationship with the classes of the polynomial time hierarchy. We will also see that the class $\mathcal{BPP}$ has polynomial size circuits. Let us now return to the possibility of randomized solutions to the problem of recognizing primes. The following beautiful theorem is often considered to be the first result dealing with randomized complexity.

Theorem 5.7 (Rabin, Solovay-Strassen, 1976) *PRIMES* $\in$ *co-*$\mathcal{RP}$.

More recently, using some formidable machinery, it has been shown that PRIMES $\in \mathcal{RP}$.

Theorem 5.8 (Adleman-Huang, 1986) *PRIMES* $\in \mathcal{RP}$.

Reducing the error: In our definition of the class $\mathcal{BPP}$, we admitted algorithms with probability of error as high as $1/4$. In many of our applications later, it will be necessary to assume that this probability of error is very small. We show below how this error probability can be reduced significantly, while still maintaining that the running time of the algorithm is bounded by a polynomial in the input length.

Theorem 5.9 *(a)* $L \in \mathcal{BPP}$ *iff for every polynomial* $p : \mathbf{N} \to \mathbf{N}$, *there exists a polynomial time algorithm recognizing* L *with error probability at most* $(\frac{1}{2})^{p(|x|)}$ *on input* x.

(b) $L \in \mathcal{RP}$ *iff for every polynomial* $p : \mathbf{N} \to \mathbf{N}$, *there exists a polynomial time algorithm* A *recognizing* L *such that for*

- *If* $x \in L$ *then* $\Pr[A$ *accepts* $x] \geq 1 - (\frac{1}{2})^{p(|x|)}$
- *If* $x \notin L$ *then* $\Pr[A$ *rejects* $x] = 1$.

Proof.

(a) The reverse implication is trivial and so we prove only the if direction. Since $L \in \mathcal{BPP}$ there is a polynomial time randomized algorithm A recognizing L with error probability at most $\frac{1}{4}$. The idea is to run the algorithm many times, independently, and use the answer it returns most frequently.

Consider the following algorithm where $q(n) = 3p(n)$.

> Input x, ($|x| = n$).
> Run the algorithm A $2q(n) + 1$ times, independently.
> If the number of executions that accept is at least $q(n) + 1$
>   then accept
>   else reject.

Clearly the algorithm runs in polynomial time. We claim that the algorithm has error probability at most $(\frac{1}{2})^{p(|x|)}$. Indeed the error probability of the algorithm is at most

$$\begin{aligned}
\sum_{j=0}^{q(n)} \binom{2q(n)+1}{j} \left(\frac{3}{4}\right)^j \left(\frac{1}{4}\right)^{2q(n)+1-j} &\leq \sum_{j=0}^{q(n)} \binom{2q(n)+1}{j} \left(\frac{3}{16}\right)^{(2q(n)+1)/2} \\
&\leq \left(\frac{3}{16}\right)^{q(n)} 2^{2q(n)} \\
&\leq \left(\frac{3}{4}\right)^{q(n)} \\
&\leq \left(\frac{1}{2}\right)^{p(n)}.
\end{aligned}$$

(b) This is even easier. We just run the algorithm many times and accept if even one execution accepts. We omit the calculations.

∎

We started by viewing $\mathcal{NP}$-programs as randomized programs. These randomized programs never accepted inputs not in the language, and for inputs in the language, they accepted with some positive (but perhaps extremely small) probability. Although such a minor separation in the behaviour of the program for inputs in and not in the language is unacceptable for the classes $\mathcal{BPP}$ and $\mathcal{RP}$, it is similar to the conditions for the class $\mathcal{PP}$. Indeed, any $\mathcal{NP}$-program can be transformed to a randomized algorithm so that the error probability satisfies the requirements for the class $\mathcal{PP}$.

Theorem 5.10 *$\mathcal{NP} \subseteq \mathcal{PP}$.*

Proof. Let $L \in \mathcal{NP}$ and let $P_L(x, y)$ be a predicate and $p(n)$ a polynomial such that

$$x \in L \iff \exists y \in \{0,1\}^{p(|x|)}\, P_L(x, y).$$

Consider the following randomized algorithm.

> Input: x ($|x| = n$).
> Randomly check for $i \in \{0,1\}$ if
> Randomly check for $y \in \{0,1\}^{p(n)}$ if
> $[i = 0$ or $P_L(x, y)]$.

Now, it can be easily verified that if $x \in L$, then the probability that this algorithm accepts is more that $1/2$, and if $x \notin L$, then the probability that the algorithm accepts is exactly $1/2$. Hence, $L \in \mathcal{PP}$. ∎

5.2.1 The class $\mathcal{BPP}$

That $\mathcal{RP} \subseteq \mathcal{NP}$ is a direct consequence of our definitions. What is the relationship between $\mathcal{NP}$ and $\mathcal{BPP}$? It is not known if either is contained in the other. We had seen that if $\mathcal{NP}$ has polynomial size circuits, then the polynomial hierarchy collapses. It is, therefore, considered unlikely that $\mathcal{NP}$ has polynomial size circuits. Does $\mathcal{BPP}$ have polynomial size circuits? Before answering this question we need to obtain a characterization of $\mathcal{BPP}$ similar to the one for $\mathcal{NP}$ used above.

A language L is in $\mathcal{BPP}$ if and only if there exists a polynomial time computable predicate $P(x,y)$ and a polynomial $p(n)$, such that for all $x \in \{0,1\}^*$

$$x \in L \Rightarrow \Pr_{y\in\{0,1\}^{p(|x|)}}[P(x,y)] \geq \frac{3}{4}; \tag{5.1}$$

$$x \notin L \Rightarrow \Pr_{y\in\{0,1\}^{p(|x|)}}[P(x,y)] \leq \frac{1}{4}; \tag{5.2}$$

By reducing the error probability (Theorem 5.9), we may replace the bounds $(3/4, 1/4)$ by $(1-2^{-q(n)}, 2^{-q(n)})$ for any polynomial $q(n)$.

Theorem 5.11 *$\mathcal{BPP}$ has polynomial size circuits.*

Proof. Let $L \in \mathcal{BPP}$ and let $Q_L : \{0,1\}^* \to \{0,1\}$ be its characteristic vector. Now consider the characterization (5.1)-(5.2) . By reducing the error probability we may assume that

$$\Pr_{y\in\{0,1\}^{p(n)}}[Q_L(x) \neq P(x,y)] < 2^{-n}. \tag{5.3}$$

For $x \in \{0,1\}^n$, let $\mathcal{E}(x) = \{y \in \{0,1\}^{p(n)} : Q_L(x) \neq P(x,y)\}$. By (5.3), we have that $|\mathcal{E}(x)| < 2^{p(n)} \cdot 2^{-n}$, and therefore

$$|\bigcup_{x\in\{0,1\}^n} \mathcal{E}(x)| \leq \bigcup_{x\in\{0,1\}^n} |\mathcal{E}(x)| < 2^{p(n)}.$$

Since the number of strings y of length $p(n)$ is $2^{p(n)}$, we conclude that, for each $n \in \mathbf{N}$, there exists $y \in \{0,1\}^{p(n)}$ such that, for all $x \in \{0,1\}^{p(n)}$, $y \notin \mathcal{E}(x)$. Let $\hat{y}_n$ denote such a y for input lenght n; thus, for all $x \in \{0,1\}^n$, $Q_L(x) = P(x,\hat{y}_n)$. We shall use these $\hat{y}_n$ to obtain the polynomial size circuits for L.

Now, the predicate $P(x,y)$ is computable in polynomial time. Hence, by Theorem 4.1, there exist polynomial size circuits, say $C_1, C_2, C_3 \ldots$, where C_i computes $P(x,y)$ correctly on inputs (x,y) of length i. To obtain the circuits the language L we hard-code the strings $\hat{y}_n$ obtained above. That is, we obtain circuits $\hat{C}_1, \hat{C}_2, \ldots$, defined by $\hat{C}_n(x) = C_{n+p(n)}(x,\hat{y}_n)$ recognizing the language L. ∎

Although, we cannot decide whether or not $\mathcal{BPP} \subseteq \mathcal{NP}$, the following theorem locates $\mathcal{BPP}$ in the polynomial time hierarchy.

Theorem 5.12 $\mathcal{BPP} \subseteq \Sigma_2^{\mathrm{P}} \cap \Pi_2^{\mathrm{P}}$

Proof. It follows from the definition of $\mathcal{BPP}$ that $\mathcal{BPP} =$ co-$\mathcal{BPP}$. Hence, it suffices to show that $\mathcal{BPP} \subseteq \Pi_2^{\mathrm{P}}$. Let $L \in \mathcal{BPP}$. We again use the characterization (5.1)-(5.2) with error reduced to 2^{-n}. We shall show that there is Π_2^{P} program that recognizes L for all but finitely many input lengths (why is this enough?).

Consider the following Π_2^{P} program. For two 0-1 vectors v and w of the same lenght $v \oplus w$ denotes the 0-1 vector of the same length whose ith component is the exclusive (sum modulo 2) of of the ith components of v and w.

Input x ($|x| = n$).
$m \leftarrow p(n)$.
Universally check for $z_1, z_2, \ldots, z_m \in \{0,1\}^{p(n)}$ if
Existentially check for $\hat{z} \in \{0,1\}^{p(n)}$ if
$\bigwedge_{i=1}^{m} P(x, z_i \oplus \hat{z})$.

We claim that the above program recognizes L. We need to show two things.

1. If $x \in L$ then for all $z_1, z_2, \ldots, z_m \in \{0,1\}^{p(n)}$, there exists a $\hat{z}$ such that $\bigwedge_{i=1}^{m} P(x, z_i \oplus \hat{z})$.

   Indeed, we shall show that for each choice of the z_i's, a randomly chosen $\hat{z}$ will have this property with high probability. Fix $z_1, z_2, \ldots, z_m \in \{0,1\}^{p(n)}$ and notice that if $\hat{z}$ is chosen randomly from $\{0,1\}^{p(n)}$, then $z_i \oplus \hat{z}$ is also a random vector in $\{0,1\}^{p(n)}$ (with uniform distribution, of course!). Therefore, for $i = 1, 2, \ldots, m$,

$$\Pr_{\hat{z} \in \{0,1\}^{p(n)}}[\neg P(x, z_i \oplus \hat{z})] \leq 2^{-n}.$$

   Thus

$$\Pr_{\hat{z} \in \{0,1\}^{p(n)}}[\exists i \, \neg P(x, z_i \oplus \hat{z})] \leq m \cdot 2^{-n} < 1$$

   (for n large enough). Hence there exists a $\hat{z} \in \{0,1\}^{p(n)}$ such that $\bigwedge_{i=1}^{m} P(x, z_i \oplus \hat{z})$. Thus inputs in the language are accepted by the program.

2. If $x \notin L$, then there exists a choice of z_i's from $\{0,1\}^{p(n)}$ such that, for all $\hat{z} \in \{0,1\}^{p(n)}$, there exists an $i \in \{1, 2, \ldots, m\}$ such that $P(x, z_i \oplus \hat{z}) = \textit{false}$.

   This time we will show that if the z_i are chosen randomly and independently, then they will have the required property with high probability. Indeed, we have for any fixed $\hat{z}$ that

$$\Pr[\forall i P(x, \hat{z} \oplus z_i)] \leq (2^{-n})^m.$$

   Thus

$$\Pr[\exists \hat{z} \in \{0,1\}^{p(n)} \forall i \, P(x, \hat{z} \oplus z_i)] \leq 2^{p(n)} (2^{-n})^{p(n)} < 1.$$

   Thus, there exists a choice of z_i's with the required properties. Hence, the program rejects all inputs not in the language.

∎

Remarks

The study of primality testing and related number theoretic problems is the subject of the survey article of Lenstra and Lenstra [LL90]. Theorem 5.2 is due to Pratt [Pra75]. The procedure Powering is taken from [CLR90, p. 829]; the efficient the nondeterministic algorithm for showing that PRIMES $\in \mathcal{NP}$ is taken from [BDG87a, p. 59]. Randomized algorithms are described in [BDG87a, Chap. 6]; Theorem 5.9 is taken from [BDG87a, p. 139]. Theorem 5.11 is due to Bennett and Gill [BJ81] (earlier Adleman [Adl78] had shown that $\mathcal{RP}$ has polynomial size circuits). Sipser and Gács showed that $\mathcal{BPP} \subseteq$ PH; the proof presented in the lecture is due to Lauteman [Lau83].

Lecture 6

Toda's Theorems

Lecturer: Jaikumar Radhakrishnan **Date:** 18 February, 1998

6.1 Counting classes

A nondeterministic polynomial time machine N is ranked if there is a polynomial $p(n)$ such that the computation tree of N on inputs x, is a complete binary tree of depth $p(|x|)$. For a ranked nondeterministic polynomial time machine (an $\mathcal{NP}$-machine) N, let $\#\mathsf{N}$, $\#\overline{\mathsf{N}}$ denote the functions which count the number of accepting and rejecting computations of N respectively; that is, on input x, $\#N(x)$ gives the number of accepting computations of N, and $\#\overline{N}(x)$ gives the number of rejecting computations of N. Note that the classes $\mathcal{NP}$, $\mathcal{PP}$ and $\mathcal{BPP}$ can be defined using the number of accepting and rejecting computations of ranked $\mathcal{NP}$-machines.

Proposition 6.1 *(a) $L \in \mathcal{NP}$ iff there exists a ranked $\mathcal{NP}$-machine N such that, for all x,*

$$x \in L \iff \#\mathsf{N}(x) > 0.$$

(b) $L \in \mathcal{PP}$ iff there exists a ranked $\mathcal{NP}$-machine N such that, for all x,

$$x \in L \iff \#\mathsf{N}(x) > \#\overline{\mathsf{N}}(x).$$

Equivalently, $L \in \mathcal{PP}$ iff there exists a ranked $\mathcal{NP}$-machine N and a polynomial $p(n)$ giving the depth of computation tree of N, such that, for all x,

$$x \in L \iff \#\mathsf{N}(x) > 2^{p(n)-1}.$$

(c) $L \in \mathcal{BPP}$ iff there exists a ranked $\mathcal{NP}$-machine N and a polynomial $p(n)$ giving the depth of computation tree of N such that, for all x,

- $x \in L \Rightarrow \#\mathsf{N}(x) \geq \frac{3}{4}2^{p(|x|)}$*;*
- $x \notin L \Rightarrow \#\mathsf{N}(x) \leq \frac{1}{4}2^{p(|x|)}$*.*

Notice how each of the classes above is defined using the count of accepting and rejecting computations of $\mathcal{NP}$-machines. Such classes are broadly referred to as *counting classes* in complexity theory. Two other important counting classes are $\oplus\mathcal{P}$ and $\#\mathcal{P}$.

Definition 6.2 *(a) $L \in \oplus\mathcal{P}$ iff there is a ranked $\mathcal{NP}$-machine N such that, for all x,*

$$x \in L \iff \#\mathsf{N}(x) \textit{is odd}.$$

(b) $\#\mathcal{P}$ is a class of functions $f : \mathbf{N} \to \mathbf{N}$; $f \in \#\mathcal{P}$ iff there exists a ranked $\mathcal{NP}$-machine N, such that, for all x, $f(x) = \#\mathsf{N}(x)$.

These classes are interesting and important as they contain several natural computational problems.

Proposition 6.3 *(a) The language*

$$\oplus SAT = \{\varphi : \varphi \text{ is CNF formula with odd number of satisfying assignments}\}.$$

is in $\oplus\mathcal{P}$.

(b) The function which, given an undirected graph G*, returns the number of matchings in* G*, is in* $\#\mathcal{P}$.

Our main objective in this lecture is to relate the complexity of languages in polynomial hierarchy with the counting classes $\mathcal{PP}$, $\oplus\mathcal{P}$, $\#\mathcal{P}$. To begin with, we note a relationship between the complexity of languages in $\mathcal{PP}$ and functions in $\#\mathcal{P}$. We also consider some *counting operators* which go into the definition of counting classes and derive some properties about them; these will be useful in proving the main results.

Lemma 6.4 *(a) Any function in* $\#\mathcal{P}$ *can be computed in deterministic polynomial time using a language in* $\mathcal{PP}$ *as an oracle; that is,* $\#\mathcal{P} \subseteq PF(\mathcal{PP})$.

(b) Any language in $\mathcal{PP}$ *can be recognized in deterministic polynomial time using a function in* $\#\mathcal{P}$ *as an oracle; that is,* $\mathcal{PP} \subseteq \mathcal{P}(\#\mathcal{P})$.

(c) $\mathcal{P}(\mathcal{PP}) = \mathcal{P}(\#\mathcal{P})$

Proof. Omitted. ∎

6.2 Counting operators

Given a complexity class $\mathcal{C}$, we can define new classes $\oplus\cdot\mathcal{C}$, $P\cdot\mathcal{C}$, $BP\cdot\mathcal{C}$ as follows.

Definition 6.5 *(a)* $L \in \oplus\cdot\mathcal{C}$ *iff there exists a set* $A \in \mathcal{C}$ *and a polynomial* $p(n)$ *such that*

$$L = \{x : \text{the number of } w\text{'s in } \{0,1\}^{p(|x|)} \text{ such that } \langle x, w\rangle \in A \text{ is odd}\}.$$

(b) $L \in P\cdot\mathcal{C}$ *iff there exists a set* $A \in \mathcal{C}$ *and a polynomial* $p(n)$ *such that*

$$L = \{x : \text{the number of } w\text{'s in } \{0,1\}^{p(|x|)} \text{ such that } \langle x, w\rangle \in A \text{ is} > 2^{p(|x|)-1}\}.$$

(c) $L \in BP\cdot\mathcal{C}$ *if and only if there exists a set* $A \in \mathcal{C}$ *and a polynomial* $p(n)$ *such that for all* x*, the number of* w*'s in* $\{0,1\}^{p(|x|)}$ *such that (*$\langle x, w\rangle \in A \Leftrightarrow x \in L$ *is)* $> \frac{3}{4}2^{p(|x|)}$.

Let us derive some properties of these operators which will be useful later.

Proposition 6.6 *(a)* $\oplus\cdot\mathcal{P} = \oplus\mathcal{P}$; $P\cdot\mathcal{P} = \mathcal{PP}$; $BP\cdot\mathcal{P} = \mathcal{BPP}$.
For any complexity classes $\mathcal{C}$, $\mathcal{C}_1$ *and* $\mathcal{C}_2$:

(b) $BP\cdot\mathcal{C} \subseteq P\cdot\mathcal{C}$.

(c) $\mathcal{C}_1 \subseteq \mathcal{C}_2$ *implies (i)* $BP\cdot\mathcal{C}_1 \subseteq BP\cdot\mathcal{C}_2$*; (ii)* $P\cdot\mathcal{C}_1 \subseteq P\cdot\mathcal{C}_2$*; (iii)* $\oplus\cdot\mathcal{C}_1 \subseteq \oplus\cdot\mathcal{C}_2$.

Lemma 6.7 (Amplification property) *Let $\mathcal{C}$ be a complexity class which is closed under majority reductions, that is, if $L \in \mathcal{C}$ then $L' \in \mathcal{C}$ where*

$$L' = \{\langle x_1, x_2, ..., x_k \rangle : \text{ more than half of } x_i\text{'s are in } L\}.$$

Let $A \in BP \cdot \mathcal{C}$ and $p(n)$ be any polynomial. Then there is a set $B \in \mathcal{C}$ and a polynomial $q(n)$, such that for all x, the number of w's in $\{0,1\}^{q(|x|)}$ for which ($x \in A \iff \langle x, w\rangle \in B$) is at least $(1 - 2^{-p(|x|)})2^{q(|x|)}$, that is, the error probability is at most $2^{-p(|x|)}$.

Proof. Omitted. ∎

Corollary 6.8 *$BP \cdot \oplus\mathcal{P}$ is an amplifiable probabilistic class.*

Lemma 6.9 (Absorption property of $BP\cdot$) *For any class $\mathcal{C}$ closed under majority reductions, $BP \cdot BP \cdot \mathcal{C} \subseteq BP \cdot C$.*

Proof. Omitted. ∎

Corollary 6.10 *$BP \cdot BP \cdot \oplus\mathcal{P} \subseteq BP \cdot \oplus\mathcal{P}$.*

Proposition 6.11 (Absorption property of $\oplus\cdot$ operator) *Let the complexity $\mathcal{C}$ be closed under many-one polynomial time reductions, $\oplus \cdot \oplus \cdot \mathcal{C} \subseteq \oplus \cdot C$. Therefore, $\oplus \cdot \oplus\mathcal{P} \subseteq \oplus\mathcal{P}$.*

Lemma 6.12 *$\oplus \cdot BP \cdot \oplus\mathcal{P} \subseteq BP \cdot \oplus\mathcal{P}$.*

Proof. Let $L \in \oplus \cdot BP \cdot \oplus\mathcal{P}$. Then there is a language $L' \in BP \cdot \oplus\mathcal{P}$ and a polynomial $p(n)$ such that for all x, $x \in L$ iff number of y's in $\{0,1\}^{p(|x|)}$ such that $\langle x, y\rangle$ is in L' is odd. Since $BP \cdot \oplus\mathcal{P}$ is an amplifiable class, there is a language $L'' \in \oplus\mathcal{P}$ and a polynomial $q(n)$ such that for all $\langle x, y\rangle$, the fraction of z's in $\{0,1\}^{q(|\langle x,y\rangle|)}$ such that ($\langle x, y\rangle \in L' \iff \langle\langle x, y\rangle, z\rangle \in L''$) is $\geq 1 - \frac{1}{2^{2p(|\langle x,y\rangle|)}}$. Let $p'(n)$ be a polynomial so that for any x and any $y \in \{0,1\}^{p(|x|)} : |\langle x, y\rangle| = p'(|x|)$.

Since there are at most $2^{p(|x|)}$ different y's, and for any given y, there are at most $2^{-2p(p'(|x|))}$ fraction of z's which are wrong witnesses; it follows that for any x, there is at least $1 - 2^{-2p(p'(|x|))}$ fraction of witnesses z's in $\{0,1\}^{q(|\langle x,y\rangle|)}$ which work correctly for all y's in $\{0,1\}^{p(|x|)}$, i.e. for every $\langle x, y\rangle$. It follows now that $L \in BP \cdot \{K\}$ where the set K is defined as $\{\langle x, z\rangle : |z| = q(p'(|x|))$ and the number of y's in $\{0,1\}^{p(|x|)}$ such that $\langle\langle x, y\rangle, z\rangle \in L''$, is odd $\}$. Also it is easily seen that $K \in \oplus \cdot \oplus\mathcal{P} = \oplus\mathcal{P}$. Therefore, $L \in BP \cdot \oplus\mathcal{P}$. ∎

Theorem 6.13 *$PH \subseteq \mathcal{P}(\#\mathcal{P})$.*

To prove this theorem, we will proceed indirectly. We will first show that PH $\subseteq$ BP $\cdot \oplus\mathcal{P}$; that is, any language in PH is rcognized by a program of the following kind.

> Input x ($|x| = n$)
> Randomly check for $y \in \{0,1\}^{p(n)}$
> Parity check for $z \in \{0,1\}^{q(n)}$
> $P(x, y, z)$.

Finally, to prove Toda's theorem, we will show that BP $\cdot \oplus\mathcal{P}$ is contained in $\mathcal{P}(\#\mathcal{P})$.

6.3 Toda's first theorem: $\mathbf{PH} \subseteq \mathbf{BP} \cdot \oplus\mathcal{P}$

For this part, it is enough to show that $\Sigma_k^{\mathrm{P}} \subseteq \mathrm{BP} \cdot \oplus\mathcal{P}$, for all $k \geq 0$. We will do this by induction on k. To keep the presentation simple, we will only consider $k = 1$ and $k = 2$; the argument for $k > 2$ is similar.

Theorem 6.14 *$\mathcal{NP} \subseteq \mathcal{RP} \cdot \oplus\mathcal{P}$.*

Proof. Let $L \in \mathcal{NP}$. Suppose the following program recognizes L.

Program A
Input x ($|x| = n$).
Existentially check for $y \in \{0,1\}^n$
$P(x,y)$.

Based on the predicate P, we will produce a nondeterministic program B of the following form.

Program B
Input x, w ($|x| = n$, $|w| = p'(n)$).
Existentially check for $y \in \{0,1\}^n$
$Q(x,w,y)$.

The predicate Q will be defined (using P) in such a way that, if $x \notin L$, (that is, $\forall y \neg P(x,y)$), then for all w, the above program rejects; on the other hand, if $x \in L$, then for most values of w, the number of accepting paths of B will be odd. We thus have the following program for recognizing L.

Program B
Input x ($|x| = n$).
Randomly check for $w \in \{0,1\}^{p'(n)}$
Parity check for $y \in \{0,1\}^{q'(n)}$
$Q(x,w,y)$.

Note that for $x \in L$, this program accepts with probability close to 1, and for $x \notin L$, the program rejects with probability 1. It follows that $L \in \mathcal{RP} \cdot \oplus\mathcal{P}$.

It now remains to show how to obtain program B. We use the input w of program B to assign weights to the guesses y of A. We think of w as an element of $[0..t-1]^{p(n)}$; the weight of y is

$$w(y) \;=\; \langle w, y\rangle \;=\; \sum_{i=1}^{p(n)} w(i)y[i].$$

This, weight function will enable us to isolate one y from the several possible y's that make $P(x,y)$ true. In fact, we have the following remarkable fact.

Lemma 6.15 (Isolation lemma) *Let $F \subseteq \{0,1\}^{p(n)}, F \neq \emptyset$. If w is chosen at random from $[0..t-1]^{p(n)}$, then with probability at least $p(n)/t$, there is a unique y of maximum weight in F.*

Proof. Omitted. ∎

Now, consider the following program.

Inputx, w ($|x| = n$, $w \in [0..t-1]^{p(n)}$).
Existentially check for bits $b_0, b_1, \ldots, b_{(t-1)p(n)}$
Existentially check for $y_0, y_1, \ldots, y_{(t-1)p(n)} \in \{0,1\}^{p(n)}$
for $i = 0, 1, \ldots, (t-1)p(n)$ do
{ if $b_i = 0$ and $y_i \neq 0^{p(n)}$ then reject
if $b_i = 1$ or $\langle w, y_i \rangle < i$ or $\neg P(x, y)$ then reject
}
Accept.

It follows from the lemma that if $x \in L$, then for a fraction $t/p(n)$ of the w there are an even number of accepting paths; on the other hand if $x \notin L$, then there is exactly one acceptng path. It is straightforward to modify this program so that the number of accepting paths is odd when $x \in L$ and even when $x \notin L$. The resulting program is the required program B. The probability of error is at most $p(n)/t$, which can be made less than $1/4$ by choosing $t = 4p(n)$. Furthermore, by Corollary 6.8, the error can be made exponentially small. ∎

Theorem 6.16 $\Sigma_2^{\mathrm{P}} \subseteq BP \cdot \oplus\mathcal{P}$.

Proof. (Sketch.) Let $L \in \Sigma_2^{\mathrm{P}}$, and let the following Σ_2^{P}-program recognize L.

Input x ($|x| = n$).
Existentially check for $y \in \{0,1\}^{p(n)}$
Universally check for $z \in \{0,1\}^{q(n)}$
$P(x, y, z)$.

Proceedings exactly as in Theorem 6.14, we may replace the existential quantifier by a BP and a parity quantifier. (Details omitted.) Furthermore, the universal quantifier corresponds to a co-$\mathcal{NP}$ computation. From Theorem 6.14, we have co-$\mathcal{NP} \subseteq$ co-$\mathcal{RP} \cdot \oplus\mathcal{P} \subseteq \mathrm{BP} \cdot \oplus\mathcal{P}$. Thus, we have shown that

$$\Sigma_2^{\mathrm{P}} \subseteq \mathrm{BP} \cdot \oplus \cdot \mathrm{BP} \cdot \oplus\mathcal{P}.$$

By Lemma 6.12 the right hand side is contained in $\mathrm{BP} \cdot \mathrm{BP} \cdot \oplus\mathcal{P} \subseteq \mathrm{BP} \cdot \oplus\mathcal{P}$, where we used Corollary 6.10 for the last containment. ∎

6.4 Toda's second theorem: BP $\cdot \oplus\mathcal{P} \subseteq P(\#\mathcal{P})$

In this section, we will show that every language in $\mathrm{BP} \cdot \oplus\mathcal{P}$ is computable in polynomial time using at most one query to the oracle of some function in $\#\mathcal{P}$.

Theorem 6.17 $BP \cdot \oplus\mathcal{P} \subseteq P(\#\mathcal{P}[1])$.

Let $L \in \mathrm{BP} \cdot \oplus\mathcal{P}$. Then, we have a program in the following form that recognizes L.

Input x ($|x| = n$)
Randomly check for $y \in \{0,1\}^{p(n)}$
Parity check for $z \in \{0,1\}^{p_2(n)}$
$P(x, y, z)$.

Now, we are allowed to present to the oracle a nondeterministic program and an input to that program; the oracle will give us the number of accepting paths of the program. We wish to exploit the power of such an oracle to recognize L.

Fix y above, and consider the number of accepting paths that are generated as z varies. We only know that this number is odd or even for most values of y according as $x \in L$ or $x \notin L$. We will now modify the parity check part to ensure that the following property holds for the new parity check.

(*) If the original number of accepting paths was even, then the number of accepting paths now is divisible by $2^{p_1(n)+1}$, and if the original number of accepting paths was odd, the number of accepting paths now is $-1 \pmod{2^{p_1(n)+1}}$.

After modifying the parity check to ensure this property, we replace the random check and parity check quantifiers by existential quantifiers. Let m be the number of accepting paths of this nondeterministic program. Let $M = -m \pmod{2^{p_1(n)+1}}$ (i.e. $M \in \{0, 1, \ldots, 2^{p_1(n)+1} - 1\}$). Then, M is precisely the number of y's in the random check of original program for which the parity check succeeds. Thus, $x \in L$ iff $M > 2^{p_1(n)}$. We can then recognize L by using one call to a suitable $\#\mathcal{P}$ oracle.

It remains to show how we can modify the parity check to ensure propery (*). We will use the following.

Lemma 6.18 *Let* $p(x) = (x^t + 1)^t + 2^t - 1$. *Then,*

$$p(x) = \begin{cases} 0 \pmod{2^t} & \text{if } x \text{ is even} \\ -1 \pmod{2^t} & \text{if } x \text{ is odd} \end{cases}.$$

Proof. Omitted. ∎

To show (*), we first observe that for every nondeterministic program A, we can obtain another program A_1, such that if A has k accepting paths then A_1 has k^t accepting paths. By adding an extra accepting path to A_1, we can easily obtain a program A_2 that has $k^t + 1$ accepting paths whenever A has k accepting paths. By applying the same procedure to A_2 (this time adding $2^t - 1$ additional paths instead of one), we obtain a program $\hat{\mathsf{A}}$ that has $(k^t + 1)^t + 2^t - 1$ accepting paths whenever A has k accepting paths. To get (*), we combine this observation with Lemma 6.18.

Remarks

The Isolation lemma (Lemma 6.15) is due to Mulmuley, Vazirani and Vazirani [KVV87]. Toda's theorems appeared in [Tod89].

Lecture 7

Arthur-Merlin Games

Lecturer: Jaikumar Radhakrishnan **Date:** 24 February 1998

We return now to the class $\mathcal{NP}$, which we view as the class of languages L whose proof of membership can be efficiently verified. In this framework, we have a program that takes as input a string x and a proof (also coded as a string) and after a polynomial time computation decides to accept or reject. If it accepts then surely x is in the language L (that is, nothing can cause the program to accept if $x \notin L$), and if $x \in L$ then there does exist a proof that satisfies the program (thus causing it to accept).

In this model, we require that the proof of membership be given first and all at once. However, when trying to model the process of one party trying to convince another of the truth of some statement, it is natural to expect that the two parties exchange messages many times. For example, the prover might present some evidence, and the verifier, while scrutinizing it, might require some additional pieces of evidence along the way. Or, say, the verifier might ask the prover questions and decide to accept if it considers the answers satisfactory.

Would interaction add to the power of the framework we considered above? Is it possible that if our polynomial time machine were allowed to ask questions and receive answers from an infinitely powerful agent, then it could be convinced of membership for even languages outside of $\mathcal{NP}$? Unfortunately, the answer is 'No'. The verifier is deterministic; its actions are determined entirely by its input and the messages it receives. In particular, the questions it could ask a prover are also determined by the input and the answers it received to the previous questions. Thus, the prover (who we assume is infinitely powerful) can right in the beginning provide all the answers in a single message. The verifier can then compute as usual and consult this message whenever it is in need of an answer to some question (without even bothering to write the questions down). Hence, we conclude that adding interaction will not give the system any more power, as long as the verifier is deterministic.

Randomness and interaction. Let us now allow the verifier to toss coins. As before, we have two agents—a prover and a verifier. The prover is assumed to be all powerful. However, the verifier is now a polynomial time randomized program. The two parties interact according to some rules (the protocol) after which the verifier decides to accept or reject.

We must now postulate the conditions that a protocol must satisfy in order for it to recognize a language. First, if the input is in the language, the verifier must accept with high probability. That is, there must exist a prover such that, for all inputs in the language, for most coin toss sequences of the verifier, the protocol concludes with an accept. On the other hand, if the input is not in the language, no prover, however malicious, can cause the verifier to accept for more than a small fraction of the coin toss sequences.

This outline is rather informal. Before we make this precise and consider the complexity classes arising from such proof system, let us consider an example. Consider the problem of *graph-nonisomorphism*. That is, the input is a pair of graphs $\langle G_1, G_2 \rangle$ and the prover wishes to convince the verifier that they are *not* isomorphic.[1] The language corresponding to this problem,

$$\text{NON-ISO} = \{\langle G_1, G_2\rangle : G_1 \text{ is not isomorphic to } G_2\},$$

is not known to be in $\mathcal{NP}$, so we do not know if an efficient proof method with a deterministic verifier exists. Consider the following protocol. Assume that G_1 and G_2 are both graphs with vertex set $\{1, 2, \ldots, n\}$.

Verifier: Pick $i \in \{1, 2\}$ at random ($\Pr[i = 1], \Pr[i = 2] = 1/2$). Randomly permute the vertices of the graph G_i (each of the $n!$ permutations is equally likely); send the resulting graph H to the prover.

Prover: If H is isomorphic to G_1 then send 1, else send 2.

Verifier: Check if the answer sent by the prover is the same as i.

Is this a good protocol for the language NON-ISO? Suppose G_1 and G_2 are not isomorphic. Can the prover always send the correct answer j? Yes, because H can be isomorphic to only one of G_1 and G_2. Since the prover is all powerful, she can know the graph from which H originated. Thus there is a prover that can cause this verifier to accept for every coin toss sequence.

It remains still to show that if the two graphs are isomorphic, then no prover can cause the verifier to accept with significant probability. Suppose G_1 and G_2 are isomorphic. Then t is intuitively clear that any prover, when given a graph H , cannot predict which graph it originated from; in fact the original graph is as likely to be G_1 as it is to be G_2. Thus the probability that the verifier accepts is exactly $1/2$. (The reader should verify this formally.) To reduce the probability of error, we can run a large number of parallel executions of this protocol. That is in the first step the verifier independently chooses $i_1, i_2, \ldots, i_m \in \{1, 2\}^m$ randomly so that each i_j can be 1 or 2 with equal probability, independently of the others. Then the verifier sends to the prover m graphs $H_1, H_2, \ldots, H_m$ where H_j is obtained from G_{i_j} by randomly permuting its vertices. Then for $j = 1, 2, \ldots, m$, the prover must correctly tell the verifier the graph from which H_j originated (that is the value of i_j). It can be verified that this reduces the probability of error to $(1/2)^m$.

7.1 Arthur vs. Merlin

In this framework, there are two parties: *Arthur* the verifier, and *Merlin* the prover. They are expected to alternately become active and send messages or make moves. These messages, in the end, determine if the input is to be accepted. In order to keep the proof system efficient, we require that the total time taken be bounded by a polynomial. Arthur will be a randomized program; that is, he will be allowed to toss coins and determine the next message to be sent based on the outcome of the tosses, the message he has received and the input. Merlin, on the other hand will be assumed to be all powerful. His messages to Arthur will be determined by the input, the messages it has received, and the outcome of Arthur's coin tosses[2].

Now, the only source of unpredictability in Arthur's behaviour is the randomness resulting from coin tosses. Hence, the omnipotent Merlin can completely determine Arthur's actions using the outcome of the coin tosses (which our framework allows him to know) and the input.

[1]The graph G_1 and G_2 are said to be isomorphic if there is a way of renaming the vertices of one graph to make it identical to the other.

[2]Notice that the protocol for graph non-isomorphism presented above does not conform to this framework; making the coin tosses available to the prover would enable her to always give the right answer, even for inputs not in the language. We will deal with this subtlety later in this lecture.

In our definition below, we therefore do not require Arthur to compute his messages; all that he does is toss a certain number of coins and send the outcome to Merlin. However, to make our description sensible, and the purport of Arthur's message clear, Arthur will be allowed to compute his messages instead of tamely tossing coins and relegating the rest to Merlin.

Thus we have the following notion of a *protocol*. The protocol specifies who is to start communicating, the number $t(n)$ of messages to be exchanged by the parties, the length $l(n)$ of these messages and the acceptance criterion $R(\cdot)$, where n is the length of the input. If it is Arthur's turn to send the ith message, then the ith message consists of a string of $l(n)$ random bits. We denote Arthur's messages by $r_1, r_2, \ldots$. We model Merlin's actions using a function $M : \{0,1,\#\} \to \{0,1\}$. If Merlin has received messages $r_1, r_2, \ldots r_k$ so far and it his turn to communicate now, then his next message will be $M(x\#r_1\#r_2\# \ldots \#r_k)$, where x is the input . At the end the acceptance criterion R is enforced. Suppose on input x the execution produces the messages $r_1, m_1, r_2, m_2, \ldots, r_k, m_k$, then we accept iff $R(x\#r_1\#m_1\#r_2\#m_2 \ldots \#r_k)$ is *true*.

We will assume that $l(n)$ and $t(n)$ are polynomials and R is a polynomial time computable predicate. Note that the functions M is not specified by the protocol. Also, for any input x, and any Merlin, acceptance is a random event determined by the outcome of Arthur's coin tosses.

Definition 7.1 (Arthur-Merlin games) *We say that $L \in \mathsf{AM}[q(n)]$ if there exists a protocol $\Pi = (l(n), t(n), R)$, with the number of moves $t(n) = q(n)$, with Arthur making the first move, such that for all $x \in \{0,1\}^*$, Π accepts x with error probability at most $1/4$, that is,*

$$x \in L \Rightarrow \exists M : \{0,1,\#\}^* \to \{0,1\}^{l(n)} \ \Pr[\Pi \text{ accepts } x] \geq 3/4;$$
$$x \notin L \Rightarrow \forall M : \{0,1,\#\}^* \to \{0,1\}^{l(n)} \ \Pr[\Pi \text{ accepts } x] \leq 1/4.$$

In this case we say that Π recognizes L. The class $\mathsf{AM}[poly]$ is the class of languages recognized by Arthur-Merlin protocols making a polynomial number of moves. Similarly, we define $\mathsf{MA}[q(n)]$, where Merlin makes the first move. When $q(n)$ is a small constant, we will denote the class $\mathsf{AM}[q]$ and $\mathsf{MA}[q]$ by a string of A's and M's such as AM for $\mathsf{AM}[2]$ and MAM for $\mathsf{MA}[3]$ etc.

Note how the acceptance probabilities resemble those for the class $\mathcal{BPP}$. In fact, there is a version of these games where the error probabilities resemble those for the class $\mathcal{PP}$.

Definition 7.2 *We say that L is recognized by a* game against nature *if there is a protocol Π such that for all $x \in \{0,1\}^*$,*

$$x \in L \Rightarrow \exists M : \{0,1,\#\}^* \to 0,1^{l(n)} \ \Pr[\Pi \text{ accepts } x] > 1/2;$$
$$x \notin L \Rightarrow \forall M : \{0,1,\#\}^* \to 0,1^{l(n)} \ \Pr[\Pi \text{ accepts } x] \leq 1/2.$$

Theorem 7.3 *The class of languages recognized by games against nature is PSPACE.*

Proof. See Homework 1, Problem 9. [For one direction simulate the games using a PSPACE-program; for the other, recall the alternating computation characterization of PSPACE (Theorem 3.15), interpret existential checks as Merlin's moves and universal checks as Arthur's moves. Then use a trick similar to Theorem 5.10.] ∎

As in the case of $\mathcal{BPP}$, the error bounds in Definition 7.1 can be made exponentially small.

Theorem 7.4 $L \in \mathsf{AM}[q(n)]$ *iff there is a $q(n)$-move protocol that recognizes L with error probabilities $2^{-m(n)}$ for any polynomial $m(n)$.*

Proof. The 'if' direction is trivial. For the only if direction we show how the error can be reduced from $1/4$ to $2^{-m(n)}$. The idea is to run many copies of the protocol in parallel and use the answer returned most often. The calculations are identical to those for Theorem 5.9. We do not repeat them here. ∎

7.2 Finite levels of the AM hierarchy

Theorem 7.5 MA $\subseteq AM$

Proof. Let $L \in$ MA. Then there exists a two move protocol with Merlin making the first move that recognizes L. Hence L is recognized by a program of the following form with error probability at most $1/4$.

> Input x ($|x| = n$)
> Existentially check for $y_1 \in \{0,1\}^{l(n)}$ if
> Randomly check for $y_2 \in \{0,1\}^{l(n)}$ if
> $R(x, y_1, y_2)$.

The idea now is to first reduce the error in the randomized check. This would enable us to perform the existential and randomized checks in the opposite order. Consider the language L' consisting of pairs $\langle x, y_1 \rangle$ ($|x| = n, |y_1| = l(n)|$) such that

$$\langle x, y_1 \rangle \in L' \iff \Pr_{y_2 \in \{0,1\}^{l(n)}}[R(x, y_1, y_2)] \geq \frac{3}{4}.$$

Since the error probability of the protocol is at most $1/4$, we have

$$\langle x, y_1 \rangle \notin L' \iff \Pr_{y_2 \in \{0,1\}^{l(n)}}[R(x, y_1, y_2)] \leq \frac{1}{4}.$$

Hence we have the following randomized algorithm for recognizing L' with error probability at most $1/4$.

> Input $\langle x, y_1 \rangle$ ($|x| = n$, $|y_1| = l(n)$).
> Randomly check for $y_2 \in \{0,1\}^{l(n)}$ if
> $R(x, y_1, y_2)$

Since L' is $\mathcal{BPP}$, we have, using characterization (5.1)-(5.2) and the Theorem 5.9, that L' is recognized with error probability at most $2^{-(l(n)+2)}$ by a program of the following form.

> Input (x, y_1) ($|x| = n, |y_1| = l(n)$)
> Randomly check for $y_2' \in \{0,1\}^{l'(n)}$ if
> $R'(x, y_1, y_2')$

Returning to our original language L, we get the following program for recognizing it with error probability at most $2^{-(l(n)+2)}$

> **Program A**
> Input x ($|x| = n$)
> Existentially check for $y_1 \in \{0,1\}^{l(n)}$ if
> Randomly check for $y_2' \in \{0,1\}^{l'(n)}$ if
> $R'(x, y_1, y_2')$

We claim that the following program accepts L with error probability at most $1/4$.

Program B
Input x ($|x| = n$)
Randomly check for $y'_2 \in \{0,1\}^{l'(n)}$ if
Existentially check for $y_1 \in \{0,1\}^{l(n)}$ if
$R'(x, y_1, y'_2)$

To verify the claim we need to consider two cases.

1. $x \in L$: By the properties of program A there exists a $\hat{y} \in \{0,1\}$ so that for all but $2^{-(l(n)+2)} \cdot 2^{l'(n)}$ of the strings $y'_2 \in \{0,1\}^{l'(n)}$, $R'(x, y_1, y'_2)$ is true.

   Hence for all these string y'_2 there exists a string $y_1 \in \{0,1\}^{l(n)}$ such that $R'(x, y_1, y_2)$. Thus Program B accepts x with probability at least $1 - 2^{-(l(n)+2)} \geq 3/4$.

2. $x \notin L$: For $y_1 \in \{0,1\}^{l(n)}$, let

$$\mathcal{E}(y_1) = \{y'_2 \in \{0,1\}^{l'(n)} : R'(x, y_1, y'_2)\}.$$

   Since Program A has error probability at most $2^{-(l(n)+2)}$, we have that $|\mathcal{E}(y_1)| \leq 2^{l'(n)} \cdot 2^{-(l(n)+2)}$. Hence,

$$|\bigcup_{y_1 \in \{0,1\}^{l(n)}} \mathcal{E}(y_1)| \leq \sum_{y_1 \in \{0,1\}^{l(n)}} |\mathcal{E}(y_1)| \leq 2^{-(l(n)+2)} \cdot 2^{l(n)} \cdot 2^{l'(n)} \leq 2^{l'(n)}/4.$$

   Thus, for at least $(3/4)2^{l'(n)}$ strings y'_2, there exists no $y_1 \in \{0,1\}^{l(n)}$ such that $R'(x, y_1, y'_2)$ is *true*. Thus the above algorithm rejects such inputs with probability at least $3/4$.

To complete the proof that we just observe that the required protocol can easily be obtained from Program B. ∎

In fact, the method used above gives the following result.

Theorem 7.6 MAM $\subseteq$ AM

Proof. (Sketch) Let $L \in$ MAM. Then we have a program of the following form that recognizes L with error probability at most $1/4$.

Input x ($|x| = n$)
Existentially check for $y_1 \in \{0,1\}^{l(n)}$ if
Randomly check for $y_2 \in \{0,1\}^{l(n)}$ if
Existentially check for $y_3 \in \{0,1\}^{l(n)}$ if
$R(x, y_1, y_2, y_3)$.

The idea now is to decrease the error by amplifying the probabilities (see Theorem 7.4) for the AM type protocol in the last 2 steps. This gives a program that recognizes L with error probability at most $2^{-(l(n)+2)}$.

Input x ($|x| = n$)
Existentially check for $y_1 \in \{0,1\}^{l(n)}$ if
Randomly check for $y'_2 \in \{0,1\}^{l'(n)}$ if
Existentially check for $y'_3 \in \{0,1\}^{l'(n)}$ if
$R'(x, y_1, y'_2, y'_3)$

Now we switch the first two checks to obtain the following program.

Randomly check for $y_2' \in \{0,1\}^{l'(n)}$ if
Existentially check for $y_1 \in \{0,1\}^{l(n)}$ if
Existentially check for $y_3' \in \{0,1\}^{l'(n)}$ if
$R'(x, y_1, y_2', y_3')$.

A straightforward calculation (see proof of Theorem 7.5) shows that this recognizes L with error probability at most $(1/4)$. As usual we collapse the last two existential checks into one and as before transform this program into a protocol. ∎

Theorem 7.7 *For all k,* $\mathsf{AM}[k] \subseteq \mathsf{AM}$

Proof. Repeatedly apply the method of Theorems 7.5 and 7.6 and reduce the chain of alternating A's and M's to just AM. Details omitted. ∎

Note: It is important in the above Theorem for k to be a constant. The reason is that in each application of Theorem 7.5 or Theorem 7.6, we need to increase the running time by a polynomial factor (to bring the error down). We cannot afford to do this more than a constant number of times, because we wish to keep the running time bounded by a polynomial.

7.2.1 Arthur Merlin games and PH

Clearly , $\mathcal{BPP} \subseteq \mathsf{AM}$ because $\mathcal{BPP}$ is the same as $\mathsf{AM}[1]$. Also $\mathcal{NP} \subseteq \mathsf{AM}$ because languages in $\mathcal{NP}$ can be recognized by protocols where Arthur does nothing and Merlin sends just one message.

It is natural therefore to ask how big the class AM is. Does it fit in the polynomial time hierarchy? Indeed, the following theorem shows that AM is contained in the second level of the hierarchy.

Theorem 7.8 *(a)* $\mathsf{AM} \subseteq \Pi_2^{\mathrm{P}}$.

(b) $\mathsf{MA} \subseteq \Sigma_2^{\mathrm{P}} \cap \Pi_2^{\mathrm{P}}$.

Proof.

(a) Let $L \in \mathsf{AM}$. The idea is to first reduce the error probability and replace the randomized check (Arthur's move) by a Universal check followed by an existential check. Now we have two adjacent existential checks; these can be collapsed into one yielding a Π_2^{P} program for recognizing L.

The randomized check can be replaced by a universal check followed by an existential check using the technique we employed to show that $\mathcal{BPP} \subseteq \Pi_2^{\mathrm{P}}$. The details are left to a homework.

(b) That $\mathsf{MA} \subseteq \Pi_2^{\mathrm{P}}$ follows from part (a) because $\mathsf{MA} \subseteq AM$ (Theorem 7.5). On the other hand, $\mathsf{MA} \subseteq \Sigma_2^{\mathrm{P}}$ is an easy consequence of Theorem 5.12. The details are left as homework.

∎

We have seen that $\mathcal{BPP} \cup \mathcal{NP} \subseteq \mathsf{AM}$. Also we have an upperbound $\mathsf{AM} \subseteq \Pi_2^{\mathrm{P}}$. How large is the class AM? We saw earlier that permitting the verifier to use randomness enabled us to recognize languages that were in co-$\mathcal{NP}$, but were not known to be in $\mathcal{NP}$. Can all of co-$\mathcal{NP}$ be recognized by such protocols?

7.2.2 AM vs. co-$\mathcal{NP}$

We shall show that if co-$\mathcal{NP}$ is contained in AM then the polynomial hierarchy collapses. First we need a lemma.

Lemma 7.9 *If co-$\mathcal{NP} \subseteq$ AM then co-AM $\subseteq$ AM*

Proof. Let $L \in$ co-AM. Then L is recognized by a program of the following form (Why?).

Program A
Input x $(x = n)$
Randomly check for $y_1 \in \{0,1\}^{p_1(n)}$ if
Universally check for $y_2 \in \{0,1\}^{p_2(n)}$ if
$R(x, y_1, y_2)$.

We shall also assume that the error probability is at most $1/10$. Consider the language corresponding to the last universal check, that is,

$$L = \{\langle x_1, y_1\rangle : |y_1| = p_1(|x|) \wedge \forall y_2 \in \{0,1\}^{p_2(|x|)} R(x, y_1, y_2)\}.$$

Clearly L is in co-$\mathcal{NP}$. Hence by the assumption in the theorem, we have that $L \in$ AM. Hence L is recognized with probability at most $1/10$ by a program of the following form.

Program B
Input $\langle x_1, y_1\rangle$ $(|x| = n,\ |y| = p_1(n))$.
Randomly check for $y_2' \in \{0,1\}^{p_2'(n)}$ if
Existentially check for $y_3 \in \{0,1\}^{p_3(n)}$ if
$R'(x, y_1, y_2', y_3)$.

We may also rewrite Program A as follows.

Input x $(|x| = n)$
Randomly check for $y_1 \in \{0,1\}^{p_1(n)}$ if
Randomly check for $y_2' \in \{0,1\}^{p_2'(n)}$ if
Existentially check for $y_3 \in \{0,1\}^{p_3(n)}$ if
$R'(x, y_1, y_2', y_3)$

We claim that the probability of error at most $1/4$. Roughly the reason is as follows. By the property of Program A the error for the first random check is at most $1/10$, and by the property of Program B, the error in the second random check is also at most $1/10$. Thus the total probability of error $1/10 + 1/10 \leq 1/4$ as required. We leave the details to the reader. ∎

We are now in a position to prove the result claimed above.

Theorem 7.10 *If co-$\mathcal{NP} \subseteq$ AM then PH$\subseteq$ AM $\subseteq \Pi_2^{\mathrm{P}}$*

Proof. Suppose co-$\mathcal{NP} \subseteq$ AM. We will show that, for each $k \geq 1$, $\Sigma_k^{\mathrm{P}} \subseteq$ AM. The case $k - 1$ is trivial, since $\Sigma_1^{\mathrm{P}} \subseteq \mathcal{NP} \subseteq$ AM. Assume that $k \geq 2$ and $\Sigma_{k-1}^{\mathrm{P}} \subseteq$ AM; we will now show that $\Sigma_k^{\mathrm{P}} \subseteq$ AM. Let $L \in \Sigma_k^{\mathrm{P}}$. Then there is a language $L' \in \Pi_{k-1}^{\mathrm{P}}$ such that L is recognized by a program of the form

Input x $(|x| = n)$
Existentially check for $y_1 \in \{0,1\}^{p_1(n)}$ if
$(x, y) \in L'$.

Since $\Sigma_{k-1}^{\mathrm{P}} \subseteq \mathsf{AM}$, we have $L' \in$ co-AM. Then using Lemma 7.9, we have that $L' \in \mathsf{AM}$. Thus we have a program of the following form that recognizes L with probability of error at most 1/4.

> Input x ($|x| = n$)
> Existentially check for $y_1 \in \{0,1\}^{p_1(n)}$ if
> Randomly check for $y_2 \in \{0,1\}^{p_2(n)}$ if
> Existentially check for $y_3 \in \{0,1\}^{p_3(n)}$ if
> $R'(x, y)$.

But then, $L \in \mathsf{MAM}$. By Theorem 7.6, $L \in \mathsf{AM}$. Thus Σ_k^{P} is contained in AM, completing the induction. ∎

It is therefore believed that co-$\mathcal{NP}$ is not likely to have short interactive proofs. This intuition is strengthened by the following relativized result, which suggests that co-$\mathcal{NP}$ does not have polynomial length interactive proofs.

Theorem 7.11 *There exists an oracle B such that co-$\mathcal{NP}(B) \not\subseteq \mathsf{AM}[poly](B)$*

The theorem suggests that perhaps not all languages in co-$\mathcal{NP}$ have polynomial interactive proofs even if no bound is imposed on the number of moves. However this is an important example of a relativized separation that is false in the real world — we will see later in the course, that without oracles co-$\mathcal{NP} \subseteq \mathsf{AM}[poly]$.

7.3 Public coins vs. Private coins

As remarked earlier, the protocol for graph non-isomorphism given in the last lecture does not conform to the Arthur-Merlin games framework. Protocols where the verifier does not disclose the outcome of the coins directly are called *private coin* protocols, and the class of languages recognized (error $\leq 1/4$) by such protocols using k-moves is denoted by $\mathsf{IP}[k]$. Thus, we know that NON-ISO $\in \mathsf{IP}[2]$.

The following theorem shows that private coin protocols can always be replaced by public coin protocols (that is Arthur-Merlin games).

Theorem 7.12 *For any polynomial $q(n)$, $\mathsf{IP}[q(n)] \subseteq AM[q(n) + 2]$.*

This is a deep and beautiful result. We will not be able to present its proof in our lectures. We will content ourselves by providing a public coin protocol for recognizing NON-ISO. Some of the ideas employed in the proof of the above theorem will already be encountered in the special case.

First, we need to prepare some background related to hashing.

Definition 7.13 (Linear hash function) *Let D be a $b \times k$ Boolean matrix. Let $h_D : \{0,1\}^k \to \{0,1\}^b$ be the linear function defined by $h_D(x) = Dx$ (we are using modulo 2 arithmetic). A* random linear function *is obtained by selecting the bk entries randomly and independently.*

For $C \subseteq \{0,1\}^n$, we let $h(C) = \{h(x) : x \in C\}$.

Lemma 7.14 *Let $m, b > 0$, $C \subseteq \{0,1\}^m$, and $c = |C|/2^b \leq 1$. Let $h : \{0,1\}^m \to \{0,1\}^b$ be a random linear function and z a random element of $\{0,1\}^b$. Then*

$$\Pr[z \in h(C)] \geq c - \frac{c^2}{2}.$$

Proof. We first have the following claims.

1. For $x, y \in \{0,1\}^m$ $(x \neq y)$, $\Pr[h(x) = h(y)] = 2^{-b}$.
2. $\Pr[z \in h(C)] \geq \sum_{x \in C} \Pr[z = h(x)] - \sum_{x,y \in C \wedge x \neq y} \Pr[z = h(x) = h(y)]$.

For 1, observe that the probability of a bit of $h(x)$ agreeing with a bit of $h(y)$ is exactly 1/2 (these events being independent for the various bits). To see 2, for each $w \in \{0,1\}^m$, define the event

$$\mathcal{A}_w \equiv (z = h(w)).$$

Then, $\Pr[z \in h(C)] = \Pr[\bigcup_{w \in C} \mathcal{A}_w]$, which by the inclusion-exclusion principle gives 2.

Now, for $x, y \in C$ $(x \neq y)$, $\Pr[z = h(x) = h(y)] = 2^{-2b}$. Thus, from 1 and 2, we have

$$\Pr[z \in h(C)] \geq \frac{|C|}{2^b} - \binom{|C|}{2}/2^{2b} \geq \frac{|C|}{2^b} - \frac{|C|^2}{2} \cdot \frac{1}{2^{2b}} = c - \frac{c^2}{2}.$$

∎

Theorem 7.15 *NON-ISO* $\in$ *AM*.

Proof. To understand the main idea of the proof, let us make a simplifying assumption: the graphs we are provided have no non-trivial automorphisms, that is, every permutation of the vertices results in a new graph. Suppose the input graphs are G_1 and G_2. Assume they have the same vertex set $\{1, 2, \ldots, n\}$. Consider the set

$$\text{LIKE}(G_1, G_2) = \{H : H \cong G_1 \vee H \cong G_2\}.$$

How big is $\text{LIKE}(G_1, G_2)$. If $G_1 \not\cong G_2$, then $|\text{LIKE}(G_1, G_2)| = 2n!$; otherwise $|\text{LIKE}(G_1, G_2)| = n!$ (Why?). Now Merlin has to convince Arthur that $|\text{LIKE}(G_1, G_2)|$ is more than $n!$. Let $\mathcal{G}_n$ be the set of all the $2^{\binom{n}{2}}$ graphs on n vertices. Thus if Arthur picks a graph in $\mathcal{G}_n$ at random, he is twice as likely to pick a graph in $\text{LIKE}(G_1, G_2)$ when the graphs are not isomorphic than when the graphs are isomorphic. Unfortunately, $n!$ is a much smaller number than $2^{\binom{n}{2}}$; so Arthur cannot pick a random graph G in $\mathcal{G}_n$ and expect Merlin to convince him that the graph is isomorphic to one of G_1 and G_2. To get around this problem we use random hash functions to map $\mathcal{G}_n$ to a domain of smaller size.

Assume that the elements of $\mathcal{G}_n$ are coded as strings of length $\binom{n}{2}$. Let $q(n) = \lceil \log_2 n! \rceil + 2$. Take a random linear hash function $h : \{0,1\}^{\binom{n}{2}} \to \{0,1\}^{q(n)}$, and apply Lemma 7.14 with $C = \text{LIKE}(G_1, G_2)$. If $G_1 \not\cong G_2$, then $c = 2n!/2^{q(n)}$ and

$$\Pr[z \in h(\text{LIKE}(G_1, G_2))] \geq c - c^2/2 = c(1 - c/2) \geq 3c/4.$$

On the other hand, if $G_1 \cong G_2$, then $|h(\text{LIKE}(G_1, G_2))| \leq |\text{LIKE}(G_1, G_2)| = n!$. Thus, $\Pr[z \in h(\text{LIKE}(G_1, G_2))] \leq n!/2^{q(n)} = c/2$. We can use this fact to give a protocol as follows.

Arthur: Choose a random linear hash function $h : \{0,1\}^{\binom{n}{2}} \to \{0,1\}^{q(n)}$ and $z \in \{0,1\}^{q(n)}$ and send them to Merlin.

Merlin: If there exists a $\hat{G} \in \text{LIKE}(G_1, G_2)$ such that $h(\hat{G}) = z$, then send $\hat{G}$ to Arthur along with a proof that $\hat{G}$ is isomorphic to one of G_1 and G_2 (that is an index $i \in \{1, 2\}$ and a permutation π of $\{1, 2, \ldots, n\}$).

Arthur: Check that $h(\hat{G}) = z$; verify that $\hat{G}$ is in LIKE(G_1, G_2).

The discussion above shows that if $G_1 \ncong G_2$, then the faithful Merlin will convince Arthur with probability at least $3c/4$; on the other hand, if $G_1 \cong G_2$, then no Merlin can convince Arthur with probability more than $c/2$. We have thus achieved a constant separation in the acceptance probabilities for the two cases. This can indeed be converted to a $1/4 : 3/4$ separation. (We leave the details to the reader).

Now let us consider the problem without the assumption that G_1 and G_2 have no non-trivial automorphisms. Consider the following set

$$\text{ALIKE}(G_1, G_2) = \{\langle H, \pi \rangle : (H \cong G_1 \vee H \cong G_2) \text{ and } \pi \text{ is an automorphism of } H\}.$$

Note that if $G_1 \cong G_2$ then $|\text{ALIKE}(G_1, G_2)| = n!$ and if $G_1 \ncong G_2$, then $|\text{ALIKE}(G_1, G_2) = 2 \cdot n!$ (Why?). The rest of the proof remains the same as before, except that now Merlin has to provide not only the proof that graph $\hat{G}$ is isomorphic to one of G_1 and G_2, but also an automorphism π of $\hat{G}$. Arthur, in turn, has to check that $h(\langle \hat{G}, \pi \rangle) = z$, $\hat{G}$ is isomorphic to one of G_1 and G_2, and that π is indeed an automorphism of $\hat{G}$. We omit the details. ∎

Corollary 7.16 *If the graph isomorphism problem is $\mathcal{NP}$-complete, then PH collapses to* AM.

The reader might have noticed that in the private coin protocol of the introduction, the prover could convince the verifier with probability 1 whenever the graphs were not isomorphic. The public coin protocol presented above has two-sided error. However, it is possible to provide a different protocol where non-isomorphic graphs will be accepted without fail. (The protocol will err only for pairs of isomorphic graphs, whom it will sometimes accept). Indeed, this is a special case of a general theorem.

Theorem 7.17 *If $L \in \mathsf{AM}[q]$, then there exists an Arthur-Merlin protocol with at most $q + 1$ moves, where the error is restricted to inputs not in the language.*

Remarks

Interactive proof systems were introduced independently by Goldwasser, Micali and Rackoff [GMR89] and Babai [Bab85] (see also Babai and Moran [BM88]). The article of Goldwasser [Gol89] and the lecture notes of Beigel's course [BCD$^+$93] contain most of the material we propose to discuss. The elegant and compelling example of graph-nonisomorphism is due to Goldreich, Micali and Wigderson [GMW86].

The results relating to the finite levels of the AM hierarchy appeared in the original paper of Babai [Bab85] (see also [BM88]). Theorem 7.8 is from [BM88]. Theorem 7.10 is due to Boppana, Hastad and Zachos [BHZ87]. The relativized separation in Theorem 7.11 was shown by Fortnow and Sipser [FS89].

Theorem 7.12 was shown by Goldwasser and Sipser [GM89]; Theorem 7.15 follows from their result. The direct proof presented in the lecture is from [BHZ87]. Linear hash functions were introduced by Carter and Wegman [CW79]. The original form of Lemma 7.14 is due to Sipser [Sip83]; the simplified version used by us in the lecture is due Boppana [GM89]. Theorem 7.17 is due to Goldreich, Mansour and Sipser [GMS87].

Lecture 8

AM[poly] = PSPACE

Lecturer: Jaikumar Radhakrishnan **Date:** 3 March 1998

In the last lecture we saw that all language in PH can be recognized in (deterministic) polynomial time given an oracle for $\#\mathcal{P}$. This discovery led researchers to study the class $\#\mathcal{P}$ more closely. In particular, the *permanent value problem*, was examined more closely because it was known to be $\#\mathcal{P}$-complete. The permanent value problem is the following: given a $n \times n$ matrix $M = \{m_{i,j}\}$ with 0-1 entries, determine

$$\mathrm{perm}(M) = \sum_{\sigma \in \mathcal{S}_n} \prod_{i=1}^{n} m_{i,\sigma(i)}$$

where $\mathcal{S}_n$ is the set of all permutations of $\{1, 2, \ldots, n\}$. That is, the permanent function is similar to the determinant, with the difference that we ignore the sign of the permutation σ.

The inherently algebraic definition of the permanent function prompted the application of interesting algebraic techniques in interactive proofs. These then led to the startling conclusion that every function in $\#\mathcal{P}$ can be computed using an Arthur-Merlin protocol in polynomial time; that is, given an input x, Merlin will supply a value y and convince Arthur that indeed $f(x) = y$. In other words, the language

$$L_f = \{\langle x, f(x)\rangle : x \in \{0,1\}^*\},$$

is in AM[poly]. It follows, therefore, that PH $\subseteq$ AM[poly] (Why?). Later these techniques were applied to show that the class AM[poly] is the same as PSPACE. However, the original proofs have since been simplified, and now these results can be described without using the permanent function.

In this lecture, we will study these results. First we will show that PH $\subseteq$ AM[poly]. In the second half of the lecture we will extend this to get PSPACE = AM[poly].

8.1 PH $\subseteq$ AM[poly]

We proceed indirectly. We first show that any function in $\#\mathcal{P}$ can be computed using an Arthur-Merlin protocol. We first see how Arthur-Merlin protocols compute functions. On input x, Merlin first supplies a value y and then interacts with Arthur for polynomial amount of time. In the end, the protocol either accepts or rejects. We require the following properties of the protocol.

1. There is a Merlin such that, for all inputs x, the value y supplied by Merlin is always $f(x)$ and then
$$\Pr[\text{Arthur accepts}] = 1.$$

2. For all Merlin and all inputs x, if the value y supplied by Merlin is not $f(x)$ then

$$\Pr[\text{Arthur accepts}] \leq 1/4.$$

We will use the function #3-SAT defined in Homework 2, problem 3. This function takes a Boolean expression φ in 3-CNF and returns the number satisfying assignments it has. It is known that #3-SAT is complete for $\#\mathcal{P}$ in the following sense. Fix a function f in $\#\mathcal{P}$; then for any input x, we can obtain, in polynomial time, a 3-CNF expression φ_x such that $f(x) = \#\text{3-SAT}(\varphi_x)$.

Thus to show that every function in $\#\mathcal{P}$ is computable using an Arthur-Merlin protocol, it suffices to provide a protocol for computing the function #3-SAT. For the rest of this section, we will mainly be concerned with deriving such a protocol. That PH $\subseteq$ AM[poly] is a relatively easy consequence.

8.1.1 Arithmetization of #3-SAT

Consider a 3-CNF expression $\varphi(x_1, x_2, \ldots, x_n)$. we denote the number of satisfying assignments of φ as $\#\varphi$. Our first task is to produce an arithmetic expression whose value is exactly $\#\varphi$. Consider a clause c of φ. Say the three variables in φ are x_i, x_j and x_k. We wish to obtain a polynomial that evaluates to the same value as c for all 0-1 assignments to these variables; that is, the polynomial should be 0 if c evaluates to false under the assignment, and 1 if c evaluates to true. For example if x_i and x_j appear negated in c but x_k appears non-negated, then the polynomial we want is $1 - x_i x_j (1 - x_k)$. It is easy to see that such a polynomial p_c, exists for each clause c. Let

$$F(x_1, x_2, \ldots, x_n) = \prod_c p_c.$$

For example if $\varphi(x_1, x_2, x_3, x_4) = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_2 \vee \overline{x}_3 \vee \overline{x}_4)$, then

$$F(x_1, x_2, x_3, x_4) = (1 - (1 - x_1)x_2(1 - x_3))(1 - (1 - x_2)x_3 x_4).$$

The important property of $F(x_1, x_2, \ldots, x_n)$ is that for any assignment $\sigma \in \{0,1\}^n$, $\varphi(\sigma)$ is true iff $F(\sigma) = 1$. Then it is easy to see that

$$\#\varphi = \sum_{x_1=0}^{1} \sum_{x_2=0}^{1} \cdots \sum_{x_n=0}^{1} F(x_1, x_2, \ldots, x_n). \tag{8.1}$$

The expression on the right evaluates to a number (call this expression E_0), whose value Merlin claims is y. Can Merlin convince Arthur? In the course of our proof we will need to consider various polynomials. For example,

$$E_1(x) = \sum_{x_2=0}^{1} \cdots \sum_{x_n=0}^{1} F(x_1, x_2, \ldots, x_n)$$

is a polynomial in x_1. In general, we have the polynomial

$$E_i(x) = \sum_{x_{i+1}=0}^{1} \cdots \sum_{x_n=0}^{1} F(x_1, x_2, \ldots, x_n).$$

Note that the degree of any variable in F (and hence in any intermediate polynomial E_i) is at most $3m$ where m is the number of clauses in φ. We will use the fact that this degree is polynomially bounded to obtain a protocol for verifying Merlin's claim that the expression on the right in (8.1) evaluates to y.

8.1.2 The protocol

The idea is as follows. Merlin first sends the value y. He must now convince Arthur that $E_0 = y$. Note that

$$E_0 = E_1(0) + E_1(1).$$

So, Arthur now asks for the polynomial $E_1(x_1)$. Note that the degree of x_1 in $E_1(x_1)$ is at most the degree in $F(x_1, x_2, \ldots, x_n)$; hence it is bounded by 3m. Also, the coefficients for the various powers of x_1 are bounded because we are operating modulo a prime number bigger then 2^{n+1}. Now, Merlin sends the polynomial, that is, the $3m+1$ coefficients each coded in N bits. We do not know if the polynomial Merlin has sent is really $E_1(x_1)$, so we will call it $\hat{E}_1(x_1)$. Arthur checks if $y = \hat{E}_1(x_1) + \hat{E}_2(x_1)$. Verify that

$$\hat{E}_1(x_1) = \sum_{x_2=0}^{1} \sum_{x_3=0}^{1} \cdots \sum_{x_n=0}^{1} F(x_1, x_2, \ldots, x_n) = E_1(x_1).$$

How does Merlin convince Arthur that is indeed true? here is the main idea. Arthur just picks a random value $\alpha_1 \in \{0, \ldots, p-1\}$ for x_1. Now he can easily evaluate $\beta_1 = \hat{E}_1(x_1)(\alpha_1)$ (degree is small and coefficients are bounded). So, he is now left with the task of verifying that

$$\beta_1 = \sum_{x_2=0}^{1} \sum_{x_3=0}^{1} \cdots \sum_{x_n=0}^{1} F(\alpha_1, x_2, \ldots, x_n) = E_1(\alpha_1).$$

The idea is the same as before. Observe that

$$\hat{E}_1(\alpha_1) = E_2(\alpha_1, 0) + E_3(\alpha_1, 1).$$

So, Arthur will just ask for the polynomial $E_2(\alpha_1, x_2)$ in x_2. Say Merlin sends $\hat{E}_2(x_2)$. So Arthur now checks if $\beta_1 = \hat{E}_2(\alpha_1, 0) + \hat{E}_2(\alpha_1, 1)$, and is left now with the task of verifying that

$$\hat{E}_2(x_2) = \sum_{x_3=0}^{1} \sum_{x_4=0}^{1} \cdots \sum_{x_n=0}^{1} F(\alpha_1, x_2, \ldots, x_n) = E_1(\alpha_1, x_2).$$

As before this is accomplished by picking a random value from $\{0, \ldots, p-1\}$ for x_2, say α_2, and checking if

$$\hat{E}_2(\alpha_2) = \beta_2 =? \sum_{x_3=0}^{1} \sum_{x_4=0}^{1} \cdots \sum_{x_n=0}^{1} F(\alpha_1, \alpha_2, x_3, \ldots, x_n) = E_1(\alpha_1, \alpha_2).$$

This situation is the same as before, except that we have one summation less. Thus we continue this way, eliminating one summation each time by substitution of a random value for the variable. In the end, we would have got rid of all the summations and be left with the task of verifying that

$$\beta_n = E_n(\alpha_1, \alpha_2, \ldots, \alpha_n) = F(\alpha_1, \alpha_2, \ldots, \alpha_n).$$

But this, Arthur can do on his own by evaluating F after substituting the corresponding values of $F(\alpha_1, \alpha_2, \ldots, \alpha_n)$. We state the protocol formally as follows

Set $N = mn$
Merlin: Sends a prime p such that $2^N < p < 2^{N+1}$
and a proof that p is a prime. (Remember PRIMES $\in \mathcal{NP}$.)
Also, the value y for $\#\varphi$.

Arthur: Checks the proof for p to be a prime.
Set $\beta_0 \leftarrow y$.
for i = 1 to n do
Merlin: Sends $\hat{E}_i(x_i)$, a polynomial over Z_p of degree at most $3m$.
Arthur: Checks that $\hat{E}_i(0) + \hat{E}_i(1) = \beta_{i-1}$ else Reject and Halt.
Chooses $\alpha_i \in Z_p$ randomly.
$\beta_i \leftarrow \hat{E}_i(\alpha_i)$
end
Arthur: If $\beta_n = F(\alpha_1, \alpha_2, \ldots, \alpha_n)$ then
Halt and Accept
Else
Halt and Reject

To show that this protocol is correct we need to establish two things. First, that there exists an honest Merlin, who succeeds with probability 1. This is easy, for such a Merlin needs to send the prime p (with a proof of primality) correctly and then send the polynomial $\hat{E}_i(x_i) = E_i(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}, x_i)$ truthfully. It is easy to see that this strategy always leads to acceptance.

Next we need to show that if $y \neq \#\varphi$, then no Merlin can convince Arthur with probability greater than 1/4. Before we formally prove this let us see how the dishonest Merlin could cheat Arthur. We know that initially (why?)

$$\beta_0 \neq E_1(0) + E_1(1).$$

Now Merlin has to send $\hat{E}_1(x_1)$. If $\hat{E}_1(x_1) = E_1(x_1)$, then Arthur will notice that $\beta_0 \neq \hat{E}_1(0) + \hat{E}_1(1)$ and reject immediately. So Merlin must contrive a polynomial $\hat{E}_1(x_1)$ different from $E_1(x_1)$. Now Arthur chooses a random value α_1 for x_1. What is the probability that

$$\hat{E}_1(\alpha_1) = \beta_1 = E_1(\alpha_1)?$$

If this happens then α_1 is a root of the polynomial $\hat{E}_1(x_1) - E_1(x_1)$, a nonzero polynomial of degree at most $3m$. Hence the probability of this is at most $3m/p$. We have chosen p large to guard against this event.

So with high probability we have $\beta_1 \neq E_1(\alpha_1)$. That is, Merlin is left again with a lie. An identical argument will lead us to conclude that if Arthur is not extremely unlucky then the lie will persist in the next step, that is, $\beta_2 \neq E_2(\alpha_1, \alpha_2)$. After n steps, Merlin will still be left with a lie, that is,

$$\beta_n \neq E_n(\alpha_1, \alpha_2, \ldots, \alpha_n) = F(\alpha_1, \alpha_2, \ldots, \alpha_n).$$

But now Arthur would reject immediately. The only thing we need to worry about is hitting a root of a non-zero low degree polynomial at one of the steps. The probability that this happens is at most $3m/p$ for any step; thus the total probability of error is at most $3mn/p$. We now proceed to state this reasoning formally.

Lemma 8.1 *If $y \neq \#\varphi$ then* $\Pr[\textit{Arthur accepts}] \leq 3mn/p$.

Proof. Fix a Merlin who provides $y \neq \#\varphi$. Suppose Arthur accepts. Then Merlin must have provided polynomials $\hat{E}_i(x_i)$, for $i = 1, 2, \ldots, n$ such that

$$\beta_{i-1} = \hat{E}_i(0) + \hat{E}_i(1).$$

Since Arthur did accept in the end, we have

$$\beta_n = \hat{E}_n(\alpha_n) = E_n(\alpha_1, \alpha_2, \ldots, \alpha_n).$$

On the other hand we know that Merlin told a lie to begin with, that is, $\beta_0 \neq E_0$. Hence, there was a stage i such that

$$\beta_{i-1} \neq E_{i-1}(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}) \wedge \beta_i = E_i(\alpha_1, \alpha_2, \ldots, \alpha_i).$$

For $i = 1, 2, \ldots, n$, define the event $\mathcal{A}_i$ as follows

$$\mathcal{A}_i \equiv \beta_{i-1} \neq E_{i-1}(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}) \wedge \beta_i = E_i(\alpha_1, \alpha_2, \ldots, \alpha_i).$$

That is $\mathcal{A}_i$ represents the situation when Merlin entered the ith stage with a lie and left with a truth. We wish to estimate $\Pr[\mathcal{A}_i]$. Suppose $\mathcal{A}_i$ holds. Our definition gives

$$E_{i-1}(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}) = E_i(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}, 0) + E_i(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}, 1). \tag{8.2}$$

Since Arthur did not reject in the ith stage, we have

$$\beta_{i-1} = \hat{E}_i(0) + \hat{E}_i(1). \tag{8.3}$$

Thus, from (8.2) and (8.3), we have that as polynomials $\hat{E}_i(x_i) \neq E_i(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}, x_i)$. Now

$$\begin{aligned}
\Pr[\mathcal{A}_i] &= \Pr[\beta_{i-1} \neq E_{i-1}(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}) \wedge \beta_i = E_i(\alpha_1, \alpha_2, \ldots, \alpha_i)] \\
&\leq \Pr[\hat{E}_i(x_i) \neq E_i(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}, x_i) \wedge \hat{E}_i(\alpha_i) = E_i(\alpha_1, \alpha_2, \ldots, \alpha_i)] \\
&\leq \frac{3m}{p}.
\end{aligned}$$

Then,

$$\Pr[\text{Arthur accepts}] \leq \Pr[\bigcup_{i=1}^{n} \mathcal{A}_i] \leq \sum_{i=1}^{n} \Pr[\mathcal{A}_i] \leq \frac{3mn}{p}.$$

∎

Theorem 8.2 *PH* $\subseteq \mathsf{AM}[poly]$.

Proof. Follows from the correctness of the protocol for #3-SAT, the #$\mathcal{P}$-completeness of #3-SAT and Toda's second theorem. Details omitted. ∎

8.2 PSPACE $\subseteq \mathsf{AM}[\mathbf{poly}]$

8.2.1 Quantified Boolean Formulas

For a 3-CNF Boolean expression $\varphi(x_1, x_2, \ldots, x_n)$, we may think of the satisfiability problem as determining the truth value of the statement

$$\exists x_1, \exists x_2, \ldots, \exists x_n \; \varphi(x_1, x_2, \ldots, x_n).$$

If we generalize this idea and allow universal quantifiers in addition to existential quantifiers, then we get quantified Boolean formulas. For example, $\forall x_1 \exists x_2 \; (x_1 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_2)$ is a quantified Boolean formula. In fact, it is a true quantified Boolean formula. The combination of quantifiers in this manner might suggest to the reader some similarity with the polynomial time hierarchy. Indeed there does exist such a connection, but we will not pursue it here. Our purpose now is to describe the connection between quantified Boolean formulas and the class PSPACE. Consider the set of true quantified Boolean formulas,

$$\mathrm{QBF} = \{\Phi : \Phi \text{ is a true quantified Boolean formula}\}.$$

Theorem 8.3 *QBF is PSPACE-complete (under polynomial time many-one reductions).*

We shall not prove it here. A proof of this can be constructed using the connection between PSPACE and polynomial time alternating computation (Theorem 3.15). This theorem tells us that for any language L in PSPACE, there is polynomial time computable function f_l, such that, for all $x \in \{0,1\}^*$,

$$x \in L \iff f(x) \in f(x) \in \mathrm{QBF}.$$

Thus, if we have an Arthur-Merlin protocol for recognizing QBF, then we have a protocol for recognizing any language in PSPACE. In the rest of this lecture, we will provide a protocol for recognizing QBF.

8.2.2 Arithmetization of QBF

Consider the quantified Boolean formula

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

where $\varphi(x_1, x_2, \ldots, x_n)$ is a 3-CNF expression. (We will assume that all quantified Boolean expressions given to us have this form.) How does one arithmetize this? To begin with, we know how to arithmetize $\varphi(x_1, x_2, \ldots, x_n)$ to obtain the polynomial $F(x_1, x_2, \ldots, x_n)$. Recall that F has degree at most $3m$ (where m is the number of clauses in φ) and agrees with φ for all 0-1 assignments to the variables. Now consider the expression $\forall x_n\ \varphi(x_1, x_2, \ldots, x_n)$. This expression has $n-1$ free variables, and for each substitution of values to these variables the expression itself evaluates to either true of false. We want a polynomial that has the same behaviour. We already know how to arithmetize φ to obtain the polynomial F. Using F we can now write a new polynomial $\prod_{x_n=0}^{1} F(x_1, x_2, \ldots, x_n)$, which stands for

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

Next consider the previous quantifier $\exists x_{n-1}$. We are interested in finding a polynomial for

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

assuming we know how to build the polynomial $G(x_1, x_2, \ldots, x_{n-1})$ for $\forall x_n\ \varphi(x_1, x_2, \ldots, x_n)$. The polynomial we seek is given by

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

We denote this by

$$\coprod_{x_{n-1}=0}^{1} G(x_1, x_2, \ldots, x_{n-1}).$$

In other words the polynomial for $\exists x_{n-1} \forall x_n \varphi(x_1, x_2, \ldots, x_n)$ is

$$\coprod_{x_{n-1}=0}^{1} \prod_{x_n=0}^{1} F(x_1, x_2, \ldots, x_n).$$

We have thus arrived at the following method for transforming the quantified Boolean formula Φ into an arithmetic expression using operators $\coprod$ and $\prod$.

1. Replace the 3-CNF expression $\varphi(x_1, x_2, \ldots, x_n)$ by the polynomial $F(x_1, x_2, \ldots, x_n)$ as in the previous section.

2. Replace the quantifier $\exists x_i$ by the operator $\coprod_{x_i=0}^{1}$.

3. Replace the quantifier $\forall x_i$ by the operator $\prod_{x_i=0}^{1}$.

We will denote the final expression thus obtained by $\mathcal{E}_0$. For example, if $\Phi = \forall x_1 \exists x_2 (x_1 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_2)$, then

$$\mathcal{E}_0 = \prod_{x_1=0}^{1} \coprod_{x_2=0}^{1} (1 - (1 - x_1)(1 - x_2))(1 - x_1 x_2)).$$

The expression $\mathcal{E}_0$ always evaluates to a 0 or a 1.

Lemma 8.4 *$\mathcal{E}_0 = 1$ iff Φ is true.*

Proof. Intuitively, this is obvious. A formal proof will show the following somewhat stronger statement using induction on the number of quantifiers.

> Let $\Phi(x_1, x_2, \ldots, x_r)$ be a quantified Boolean expression with r free variables, and let $\mathcal{E}(x_1, x_2, \ldots, x_r)$ be the arithmetic expression obtained from it by using the procedure above. Then for each substitution σ of values for $x_1, x_2, \ldots, x_r$ from $\{0, 1\}^r$, $\mathcal{E}(\sigma) = 1$ iff Φ is true.

We omit the details. ∎

We have thus arrived at a way of representing quantified Boolean expressions using arithmetic operations. Can we now devise a protocol based on this? Let us consider the arithmetic expression

$$\prod_{x_1=0}^{1} \coprod_{x_2=0}^{1} \cdots \prod_{x_n=0}^{1} F(x_1, x_2, \ldots, x_n)$$

and try to use the ideas of the last section. In the first step Arthur eliminates the first operator $\prod_{x_1=0}^{1}$ and asks Merlin to send us the polynomial $G(x_1)$ corresponding to the rest of the arithmetic expression, that is

$$G(x_1) = \coprod_{x_2=0}^{1} \prod_{x_3=0}^{1} \cdots \prod_{x_n=0}^{1} F(x_1, x_2, \ldots, x_n).$$

Once Arthur has the polynomial $\hat{G}(x_1)$, he check that $\hat{G}(0) \cdot \hat{G}(1) = 1$ (for Merlin claims that $\Phi \in \text{QBF}$) and pick a random value α_1 for x_1 and compute $\beta_1 = \hat{G}(\alpha_1)$. Then Merlin must convince Arthur that

$$\beta_1 = \coprod_{x_2=0}^{1} \prod_{x_3=0}^{1} \cdots \prod_{x_n=0}^{1} F(\alpha_1, x_2, \ldots, x_n).$$

We then proceed by eliminating the operators one after other, and in the end check that $\beta_n = F(\alpha_1, \alpha_2, \ldots, \alpha_n)$. But ...

The problem: Consider the polynomial $G(x_1)$. How big is the degree? Unfortunately, the degree can be exponential in n. For example if

$$\mathcal{E}_0 = \prod_{x_1=0}^{1} \prod_{x_2=0}^{1} \prod_{x_3=0}^{1} x_1,$$

then $G_1(x) = x_1^4$. In general, each operator for the other variables has the potential of doubling the degree, leading, in the end, to an exponential degree. But what is wrong with the exponential degree? The trouble exponential degree is that the polynomial will have exponentially many coefficients, and Arthur will not be able to read it in polynomial time.

The solution: We will ensure that the degree of any variable in intermediate stages of the polynomial never goes above two. How? Indeed, the polynomial $F(x_1, x_2, \ldots, x_n)$ that we obtain from the arithmetizing a 3-CNF expression φ may not have this property. The idea is as follows. We need a polynomial that agrees with $F(x_1, x_2, \ldots, x_n)$ on assignments consisting of 0's and 1's. What it does for other values is quite irrelevant. Note, moreover, that for $x_i = 0$ and $x_i = 1$, $x_i^k = x_i$ for all $k > 0$. That is, the desired polynomial can be obtained from $F(x_1, x_2, \ldots, x_n)$ by simply erasing all exponents. (that is replacing them 1). For example, if $F(x_1, x_2, x_3) = x_1^2 x_2 + x_3^2 x_2^2 + x_1^3 x_3 + x_1 x_2$, then the reduced polynomial is $\hat{F}(x_1, x_2, x_3) = 2x_1x_2 + x_1x_3 + x_2x_3$. To apply this method in general we define a new operator, Rx_i, which when applied to a polynomial reduces the powers at all occurrences of x_i to 1 . Thus,

$$\mathcal{R}x_3[x_1^2x_2 + x_3^2x_2^2 + x_1^3x_3 + x_1x_2] = x_1^2x_2 + x_3x_2^2 + x_1^3x_3 + x_1x_2,$$

and

$$\hat{F}(x_1, x_2, x_3) = \mathcal{R}x_1\mathcal{R}x_2\mathcal{R}x_3F(x_1, x_2, x_3).$$

Thus, by reducing the degree of all variables, we obtain the required equivalent of $F(x_1, \ldots, x_n)$. But, even now the other operators ($\prod$ and $\coprod$) might cause the degrees to go up. Remember again, that the polynomials we use are of interest only at 0-1 substitutions for the variables. Any transformation that does not change the value at 0-1 substitutions is acceptable. In particular, we may reduce the degree after each application of $\prod$ or $\coprod$. This will ensure that the degree at intermediate stages never goes above 2; it might reach the value of 2 after an $\prod$ or $\coprod$ operator is applied, but will revert to 1 when the reduce operator is applied.

8.2.3 The revised arithmetization

Recall that we wish to arithmetize a quantified Boolean Formula of the form

$$\Phi = \forall x_1 \exists x_2 \forall x_3 \cdots \forall x_n \varphi(x_1, x_2, \ldots, x_n).$$

First we replace $\varphi(x_1, x_2, \ldots, x_n)$ by the polynomial $F(x_1, x_2, \ldots, x_n)$ and reduce its degree. This gives us the arithmetic expression

$$\mathcal{R}x_1\mathcal{R}x_2 \cdots \mathcal{R}x_nF(x_1, x_2, \ldots, x_n).$$

Now we come to $\forall x_n \varphi(x_1, x_2, \ldots, x_n)$. Naturally this leads to

$$\prod_{x_n=0}^{1} \mathcal{R}x_1\mathcal{R}x_2 \cdots \mathcal{R}x_nF(x_1, x_2, \ldots, x_n),$$

and then to

$$\mathcal{R}x_1\mathcal{R}x_2\cdots\mathcal{R}x_{n-1}\prod_{x_n=0}^{1}\mathcal{R}x_1\mathcal{R}x_2\cdots\mathcal{R}x_nF(x_1,x_2,\ldots,x_n).$$

Let us do one more step, that is, for $\exists x_{n-1}\forall x_n\varphi(x_1,x_2,\ldots,x_n)$. The corresponding arithmetic expression is

$$\mathcal{R}x_1\mathcal{R}x_2\cdots\mathcal{R}x_{n-2}\coprod_{x_{n-1}=0}^{1}\mathcal{R}x_1\mathcal{R}x_2\cdots\mathcal{R}x_{n-1}\prod_{x_n=0}^{1}\mathcal{R}x_1\mathcal{R}x_2\cdots\mathcal{R}x_nF(x_1,x_2,\ldots,x_n).$$

In general, we replace the quantifier $\exists x_i$ by $\mathcal{R}x_1\mathcal{R}x_2\cdots\mathcal{R}x_{i-1}\coprod_{x_i=0}^{1}$ and the quantifier $\forall x_i$ by $\mathcal{R}x_1\mathcal{R}x_2\cdots\mathcal{R}x_{i-1}\prod_{x_i=0}^{1}$. Thus, the resulting expression has $n+(n-1)+(n-2)+\ldots+1=n(n+1)/2$ reduce operators in addition to the 'n' operators of the form $\prod_{x_1=0}^{1}$, $\coprod_{x_i=0}^{1}$. We call this expression $\mathcal{E}_0$. For this refined arithmetization, we have the following lemma.

Lemma 8.5 $\mathcal{E}_0 = 1$ *iff* Φ *is true.*

Proof. Omitted. ∎

8.2.4 The protocol

The idea of the last section can now be put to use without any trouble. Arthur and Merlin communicate to eliminate one operator at each stage. In the end, no operators are left and Arthur checks the required condition directly by computing.

Initially, he wishes to be convinced that

$$y = \beta_0 =? \prod_{1=0}^{1}\mathcal{R}x_1\coprod_{x_2=0}^{1}\cdots\mathcal{R}x_1\cdots\mathcal{R}x_n\ F(x_1,x_2,\ldots,x_n).$$

So he asks Merlin for the polynomial

$$G_1(x_1) = \mathcal{R}x_1\coprod_{x_2=0}^{1}\cdots\mathcal{R}x_n\ F(x_1,x_2,\ldots,x_n).$$

This a polynomial of degree 1! So Merlin can easily provide this to Arthur. Call the polynomial that Merlin sends be $\hat{G}(x_1)$. Arthur verifies that $\prod_{x_1=0}^{1}\hat{G}(x_1)=\beta_0$ (by computing $\hat{G}(0)\cdot\hat{G}(1)$). Then to guard against Merlin sending him the wrong polynomial, he picks a random value α_1 for x_1, and obtains $\beta_1 = G(\alpha_1)$. He now needs to be convinced that

$$\beta_1 = (\mathcal{R}x_1\coprod_{x_2=0}^{1}\cdots\mathcal{R}x_n\ F(x_1,x_2,\ldots,x_n))[\alpha_1].$$

Here, the $[\alpha_1]$ means that the polynomial in the parenthesis (with free variable x_1) is to be evaluated at the value α_1.

The next operator operator, $\mathcal{R}x_1$, is dealt with similarly now. Arthur asks for the polynomial

$$G_2(x_1) = \coprod_{x_2=0}^{1}\mathcal{R}x_1\cdots\mathcal{R}x_nF(x_1,x_2,\ldots,x_n).$$

On receiving $\hat{G}_2(x_1)$ from Merlin, Arthur checks if $\beta_1 = (\mathcal{R}x_1\hat{G}_2(x_1))[\alpha_1]$ picks a value α'_1 for x_1 and computes $\beta_2 = \hat{G}_2(\alpha'_1)$ and now needs to be convinced that

$$\beta_2 = (\coprod_{x_2=0}^{1} \mathcal{R}x_1 \cdots \mathcal{R}x_n F(x_1, x_2, \ldots, x_n))[\alpha'_1].$$

In general, when Arthur needs to be convinced that

$$\beta = (\mathcal{R}x_i\ p(x_1, x_2, \ldots, x_1, \ldots, x_r))[\alpha_1, \ldots, \alpha_i, \ldots, \alpha_r],$$

he proceeds as follows. First, rewrite this as

$$\beta = (\mathcal{R}x_i\ p(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}, x_i, \alpha_{i+1}, \ldots, \alpha_r))[\alpha_i].$$

Now Arthur asks for the polynomial $G(x_i) = p(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}, x_i, \alpha_{i+1}, \ldots, \alpha_r)$, and gets $\hat{G}(x_i)$ say. He verifies that $(\mathcal{R}x_i\hat{G}(x_i))[\alpha_i] = \beta$, computes $\beta' = G(\alpha'_i)$ by choosing a random value α' for x_i, and reduces the task to the question

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

The case of the $\coprod_{x_i=0}^{1}$ operators as that of the $\prod_{x_i=0}^{1}$. We omit its discussion.

The entire protocol for QBF can now be stated as follows.

```
Merlin: a prime p ∈ [4n(3m+n), 8n(3m+n)] and a proof that p is a prime.
Arthur: β ← 1.
   for i = 0 to n−1 do
     Merlin: q, linear polynomial over Z_p.
     Arthur: If i is odd (resp. even) then
             check that ∏_{x=0}^{1} q(x_i) = β
             (resp. ∐_{x_i=0}^{1} q(x_i) = β) else Halt and Reject.
     Arthur: Choose α_{i+1} uniformly and randomly in Z_p.
     Arthur: β ← q(α_{i+1})
      for j = 1 to i+1 do
        Merlin: q a polynomial over F_p with degree at most 2
                (resp. 3m) if i < n (resp. i=n).
        Arthur: Check that Rx_j(q)(α_j) = β else Halt and Reject.
        Arthur: Choose a new α_i uniformly and randomly in F_p.
        Arthur: β ← q(α_i)
      end
   end
Arthur: If β = F(α_1, α_2, ldots, α_n) then
   Halt and Accept
  Else
   Halt and Reject
```

Lemma 8.6 *(a) If $\Phi \in QBF$ then there exists a Merlin such that* $\Pr[\textit{Arthur accepts}] = 1$.

(b) If $\Phi \notin QBF$ then for all Merlin $\Pr[\textit{Arthur accepts}] \leq 1/4$.

Proof. Part (a) is obvious.

For part (b) the analysis is the analysis is the same as in the case of #3-SAT. The error is introduced only if we pick a root of some low degree polynomial some stage. While dealing with the last n reduce operators, the polynomial may have degree at most $3m$. For the remaining reduce operators the degree is at most 2. For each of the other operators the degree is 1. Thus the sum of the degrees of all the polynomials encountered is at most $n(3m+n)$. It follows from our choice of the prime p in the protocol that the probability of error is at most $n(m+n)/p \leq 1/4$. Details omitted. ∎

Theorem 8.7 *PSPACE* $\subseteq \mathsf{AM}[poly]$.

Corollary 8.8 *PSPACE* $= \mathsf{AM}[poly]$.

Proof. AM[poly] $\subseteq$ Games Against Nature $\subseteq$ PSPACE $\subseteq$ AM[poly]. ∎

Remarks

The permanent value problem was shown to be $\#\mathcal{P}$-complete by Valiant [Val79]. The result $P(\#\mathcal{P}) \subseteq \mathsf{AM}$[poly] was shown by Lund, Fortnow, Karloff and Nisan [LFKN90] (see also [LFKN92]); their original proof used the $\#\mathcal{P}$-completeness of the permanent value problem. The proof given above is taken from Lund's Ph. D. thesis [Lun91]

The language QBF was shown to be PSPACE-complete by Stockmeyer and Meyer [SM73] (see also [BDG87a, p. 71]). Theorem 8.7 was shown by Shamir [Sha90] (see also [Sha92]); the proof described above is due to Shen [She92], who introduced the reduce operator to keep the degrees under control. An alternative proof of this result, attributed to Hartmanis, appears in [BCD^{+}93].

Homework 2

Date: 10 March 1998 **Due date:** 7 April 1998

Problems.

1. Prove the following statements using the idea of *self-reducibility:* the solution for an instance can be obtained efficiently from the solutions to some related but smaller instances.
   (a) $\mathcal{NP} \subseteq \mathcal{BPP}$ implies $\mathcal{NP} \subseteq \mathcal{RP}$.
   (b) Show that if PSPACE has polynomial size circuits then PSPACE $\subseteq \Sigma_2^{\mathrm{P}}$. (Assume that QBF is PSPACE-complete.)

2. Use Homework 1 (Problem 1) to show the following.
   (a) NLog is randomized log-space reducible to ULog. (Use the isolation lemma.)
   (b) Show that NLog $\subseteq$ ULog/poly. (See the proof of $\mathcal{BPP} \subseteq \mathcal{P}$/poly.)

3. Show that the class $\mathcal{PP}$ is closed under complement. That is, show that if a language L is in $\mathcal{PP}$, then its complement $\overline{L}$ is also in $\mathcal{PP}$.

4. Show that $\mathsf{AM} \subseteq \Pi_2^{\mathrm{P}}$.

   Hint: Use the idea described in class for showing that $\mathcal{BPP} \subseteq \Pi_2^{\mathrm{P}}$ (Theorems 5.12, 7.8).

5. The function #SAT takes as input a Boolean expression φ and returns the number of satisfying assignments of φ. The function #3SAT takes as input Boolean expressions in 3-CNF and returns the number of satisfying assignments it has.
   (a) Show that #SAT and #3SAT are in $\#\mathcal{P}$.

   We say that a function $F : \{0,1\}^* \to \mathbf{N}$ is $\#\mathcal{P}$-complete if

   - $F \in \#\mathcal{P}$.
   - For all functions $G \in \#\mathcal{P}$, there is a polynomial time computable function $f_G : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$, $F(f_G(x)) = G(x)$.

   Thus, if we can compute F in polynomial time, then we can compute any function in $\#\mathcal{P}$ in polynomial time.

   (b) Assume that #SAT is $\#\mathcal{P}$-complete. Show that #3SAT is also $\#\mathcal{P}$-complete.

   Hint: Show how to transform a Boolean expression into a 3-CNF expression keeping the number of satisfying assignments the same (see Theorem 4.5, Cook's theorem).

6. The classes EXP and NEXP are defined as follows.

$$\begin{aligned} \text{EXP} &= \bigcup_{k\geq 0} \text{DTime}(2^{n^k}); \\ \text{NEXP} &= \bigcup_{k\geq 0} \text{NTime}(2^{n^k}). \end{aligned}$$

(a) Show that if $\mathcal{P} = \mathcal{NP}$, then EXP = NEXP.

Hint: Consider the language

$$L' = \{\langle x, \lceil \mathsf{N} \rceil, 1^t \rangle : \text{the nondeterministic machine } \mathsf{N} \text{ accepts } x \text{ in at most } t \text{ steps}\}.$$

Observe that $L' \in \mathcal{NP}$.

(b) Show that if $\mathcal{NP} \subseteq \mathrm{DTime}(n^{\log n})$, then EXP = NEXP.

7. Show the following.

   (a) If the functions F and G are in $\#\mathcal{P}$ then their product $F \times G$ is also in $\#\mathcal{P}$.

   (b) If a language L is in $\oplus\mathcal{P}$, then L^* is also in $\oplus\mathcal{P}$. Here we define L^* to be the language

   $$\{\langle x_1, x_2, \ldots, x_k \rangle : \forall i \; x_i \in L\}.$$

   (c) $\mathcal{P}(\oplus\mathcal{P}) = \oplus\mathcal{P}$.

   (d) Show that $\oplus\mathcal{P}$ is closed under majority reductions (see Lemma 6.7). Conclude that $\mathrm{BP} \cdot \oplus\mathcal{P}$ is an amplifiable probabilistic class.

8. (a) Let $\mathcal{F}$ be a field and $p(x)$ a polynomial with coefficients from $\mathcal{F}$ of degree d. Suppose p is not identically 0 over $\mathcal{F}$. Show that $p(x) = 0$ for at most d values $x \in \mathcal{F}$.

   (b) Let $p(x_1, x_2, \ldots, x_m)$ be a polynomial with coefficients from the field $\mathcal{F}$ of combined degree at most d. (That is, the sum of the degrees of the variables in any term is at most d; for example, the polynomials $x_1^2 x_2 + x_3^2 x_4^2 + x_2 + 7x_1 + 11$ (over reals) has combined degree 4.) Suppose p is not identically 0 over $\mathcal{F}^m$. Let $I \subseteq \mathcal{F}$. Show that p takes the value 0 for at most $d/|I|$ fraction of the values $x \in I^m$.

9. Show the following.

   (a) $\mathcal{NP} \subseteq \mathrm{PCP}(o(\log n), 1) \Rightarrow \mathcal{P} = \mathcal{NP}$.

   (b) $\mathrm{PCP}(\log n, \mathrm{poly}) \subseteq \mathcal{NP}$.

Lecture 9
Probabilistically Checkable Proofs

Lecturer: Jaikumar Radhakrishnan **Date:** 10 March 1997

We saw in the last lecture that the class AM[poly] coincides with PSPACE. Earlier (Homework 1, Problem 9), we observed that PSPACE contains precisely those languages that are recognized by *games against nature.* The only difference between Arthur-Merlin games and games against nature is in the acceptance criterion: in Arthur-Merlin games, the probabilities of acceptance for inputs in the language and inputs not in the language must be well separated (for example, 3/4 : 1/4), whereas in games against nature any non-zero difference is enough. We now know that this distinction in the acceptance criterion makes no difference to the power of these games. A similar difference was encountered earlier between the classes $\mathcal{BPP}$ and $\mathcal{PP}$. However, it is not considered likely that these two classes are the same. Indeed, if they are identical, then the polynomial hierarchy would collapse (why?).

Having thus determined the extent of the class AM[poly], we now turn to some variations in the model. The first variation is obtained by the introduction of multiple provers.

The multiple prover model In this model we have one verifier V interacting with many provers, $P_1, P_2, \ldots, P_r$, according to some fixed protocol Π. It is assumed that a prover's response depends only on the input and the message she has received so far; in particular, a prover cannot base her response on what transpires between the verifier and the other provers. We say that a protocol Π *recognizes* a language L if the following conditions are satisfied.

1. There exist provers $P_1, P_2, \ldots, P_r$, such that for all inputs $x \in L$, $\Pr[\Pi \text{ accepts}] = 1$.
2. For all inputs $x \notin L$, for all provers $P_1, P_2, \ldots, P_r$, such that $\Pr[\Pi \text{ accepts}] \leq \frac{1}{4}$.

We say that a language L is in the class MIP if there exists a polynomial time multiprover protocol accepting L.

The class PCP The variation of interactive proofs we consider next is called *probabilistically checkable proofs.* In this model, the prover is non-adaptive – she must write down all her answers in the beginning. Imagine that the prover first writes down an exponentially long table of values (this is the proof string supplied by the prover), and then the verifier tosses coins to determine which entries of the table are to be read. Once the contents of these locations are available, the verifier performs a polynomial time (deterministic) computation and decides to accept or reject.

In the model of probabilistically checkable proofs, two resources (besides the running time of the verifier, which is assumed to be polynomially bounded) are taken into account. These two resources are the number of random bits of used and the number of bits of the proof string read (the query bits).

A probabilistic polynomial time program with 'random access' to a proof string is called a $(r(n), q(n))$-*verifier* if, on inputs of length n, it uses $O(r(n))$ random bits and reads $O(q(n))$ bits of the proof. We say that a verifier V *recognizes* a language L if

- for all inputs $x \in L$, there exists a proof string such that $\Pr[V \text{ accepts}] = 1$;
- for all inputs $x \notin L$ and all proof strings $\Pr[V \text{ accepts}] \leq \frac{1}{4}$.

The class $\mathrm{PCP}(r(n), q(n))$ consists of precisely those languages that are recognized by some $(r(n), q(n))$-verifier.

9.1 MIP and PCP

The classes MIP and PCP(poly, poly) are related; in fact, they are the same. In this section, we prove this result and describe how this new class is related to the conventional complexity classes.

Lemma 9.1 *$MIP \subseteq PCP(poly, poly)$.*

Proof. Let $L \in \mathrm{MIP}$. Say L is recognized by a protocol Π that runs in time n^k using r provers $P_1, P2, \ldots, P_r$ (we assume that $r < 2^{n^k}$). We will use Π to construct an $(n^{k'}, n^{k'})$-verifier that recognizes L. The table to be used by this verifier will store all possible responses of the provers in the protocol Π. Note that the message sent by any prover can depend only on the messages received by it previously. Hence the cells of our table will be indexed by tuples of the form $\langle i, m_1\#m_2\#\ldots\#m_k\rangle$, where $i \in \{1, 2, \ldots, r\}$, and $m_1, m_2, \ldots, m_k \in \{0, 1\}^*$. The content of this cell will be the string that prover P_i sends as the response when it receives its kth message m_k, having previously received the $k-1$ messages $m_1, m_2, \ldots, m_{k-1}$.

Now our verifier simulates the protocol Π, and whenever in need of consulting a prover, he just looks up the corresponding cell in the table (we assume that he remembers all the messages that he sent to the provers, so that he can produce the address). Now a routine back and forth argument lets us conclude that the verifier accepts the input x with probability p iff there exist provers that induce the original protocol to accept with probability p. Moreover, it can easily be checked that the verifier that we just constructed is an $(n^{k'}, n^{k'})$-verifier, for some constant k'. It follows that $L \in \mathrm{PCP}(n^{k'}, n^{k'})$. ∎

Lemma 9.2 *$PCP(poly, poly) \subseteq MIP$.*

Proof. Let L be recognized by the (n^k, n^k)-verifier V. This time we will use V to construct a two prover protocol Π for recognizing L. Recall that V obtains information from a table. The two provers, P_1 and P_2, of our protocol Π are both expected to return values according to the table. That is, the queries to these provers will be addresses of cells in the table; their response should be the content corresponding to the address supplied. The protocol is as follows. Here V_Π is the verifier of the protocol Π.

> Input x ($|x| = n$).
> For $i = 1, 2, \ldots, n^{2k}$
> {
> V_Π simulates V by consulting prover P_1 whenever in need of contents of any cell.
> Let the queries asked by the verifier be $q_1, q_2, \ldots, q_l$ and
> the corresponding responses be $r_1, r_2, \ldots, r_l$ (note $l \leq n^k$).
> If these cause the original verifier V to accept then
> {
> Pick $i \in \{1, 2, \ldots, r\}$ at random and check that
> the response of P_2 to q_i is r_i; otherwise, reject and halt.

```
        }
      else reject and halt.
    }
    Accept.
```

Thus the verifier uses the first prover as the table and simulates the old verifier; then he checks that the two provers are consistent by picking a random query to see if their responses match. Why does this work?

1. For $x \in L$, both provers answer faithfully based on the correct proof (with respect to verifier V). It is easy to see that the above protocol then accepts with probability 1.

2. For $x \notin L$, we need to show that the probability of acceptance is at most $1/4$. We claim that in any iteration of the loop, the probability of acceptance is at most $(1 - 1/(10n^k))$, irrespective of what happened in the previous iterations. It will then follow that the overall probability of acceptance is at most

$$\left(1 - \frac{1}{10n^k}\right)^{n^{2k}} \leq \exp(-\frac{n^k}{10}) \leq \frac{1}{4}.$$

To prove the claim, consider the situation at the beginning of an iteration. If the answers provided by the prover P_1 causes the verifier to reject with probability at least $1/(10n^k)$, then we are done. Otherwise, construct the table T of answers based on the responses that the second prover (i.e. P_2) would give at this stage for the various question; that is, in this table cell q will contain the value returned by prover P_2 for the query q. Since $x \notin L$, the original verifier accepts T with probability at most $1/4$. Since P_1 causes the verifier to reject with probability less that $1/(10n^k)$, for a fraction $(1 - 1/(10n^k) - 1/4) \geq 2/3$ of the coin toss sequences, the verifier when using table T would reject, but while using the responses of P_1 would accept. It follows that with probability at least $2/3$, for one of the queries in $\{q_1, q_2, \ldots, q_r\}$, the answer returned by P_1 is not consistent with the corresponding entry in T. Thus with probability $(2/3) \cdot (1/r)$ the verifier would pick this query and consequently reject. Since $r \leq n^k$, the probability of acceptance is at most

$$1 - \frac{2}{3n^k} < 1 - \frac{1}{10n^k}$$

as claimed.

∎

Theorem 9.3 *MIP* $= PCP(poly, poly)$.

This shows that the classes MIP and PCP(poly, poly) are the same; but, how big are they?

Theorem 9.4 *MIP* $\subseteq$ *NEXP*.

Proof. Let $L \in$ MIP. Then Theorem 9.3 implies that there exists a (n^k, n^k)-verifier that recognizes L. Fix an input $x \in \{0,1\}^*$. The idea is as follows. We guess the contents of the table used by this verifier and then, using it, compute the probability of acceptance of the verifier for the input x. If the probability is more than $1/4$, then we accept; otherwise, we reject. Since the table is exponentially long (why?), it can be guessed by a nondeterministic exponential time program. Our definitions guarantee that if $x \in L$, then there is at least one guess that would lead to acceptance, and if $x \notin L$, then all guesses lead to rejection. ∎

Thus, MIP is no bigger than NEXP. On the other hand we have the following remarkable theorem.

Theorem 9.5 *NEXP* $\subseteq$ *MIP.*

This discovery was the corner stone of all later developments in the area of Interactive Proofs. We will not prove this theorem here. Instead, we will move on to results that use the techniques developed while proving this result. It is hoped that after studying these results, the reader would be able to construct a proof of the above theorem on her own.

PCP and $\mathcal{NP}$: With this we begin the main topic of this course. The following is a straightforward consequence of the definition of the class $\mathcal{NP}$.

Proposition 9.6 $\mathcal{NP} = PCP(0, poly)$

This observation completely characterizes the class PCP when no coins are tossed and no bound is imposed on the number of bits of the proof to be read. A major part of our discussion from now on will be devoted to understanding the role of randomness in probabilistically checkable proofs. In particular, we would like to know if the number of bits of the proof read by the verifier can be reduced if the use of some randomness is permitted. In the rest of this lecture, we will study the first of a series of results that unravel the role played by the parameters, $r(n)$ and $q(n)$, in PCP$(r(n), q(n))$. As a first step we will show that with $O(\log n)$ random bits, the number of bits can be reduced to poly $\log n$. This result will form a building block for later lectures, where the number of query bits needed will be reduced even further.

9.2 $\mathcal{NP} \subseteq \mathbf{PCP}(\log n, \mathbf{poly} \log n)$

It will be enough to show that the set 3-SAT is in PCP$(\log n, \text{poly} \log n)$.

9.2.1 Arithmetization of 3-SAT

Consider a 3-SAT expression φ. In the following, we will assume that the clauses and variables of φ are numbered; c will denote the index of a clause and v the index of a variable. A 3-CNF expression is a conjunction of clauses; these clauses themselves are disjunctions of literals. We will represent the information contained in the expression φ using the six predicates $\chi_1(c, v)$, $\chi_2(c, v)$, $\chi_3(c, v)$, $s_1(c)$, $s_2(c)$ and $s_3(c)$. $\chi_i(c, v)$ is true precisely if v is the ith variable of the clause c; that is, for example,

$$\chi_1(c, v) = \begin{cases} 1 & \text{if } v \text{ first variable in clause } c \\ 0 & \text{otherwise} \end{cases} .$$

Similarly, we define χ_2 and χ_3. The predicate $s_i(c)$ is true precisely if the ith variable in clause c is non-negated. In particular,

$$s_1(c) = \begin{cases} 1 & \text{if the first variable in } c \text{ is not negated} \\ 0 & \text{if the first variable in } c \text{ is negated} \end{cases} .$$

Before we describe the arithmetization of 3-SAT using these predicates, we need to fix some conventions.

- For technical reasons, that will become clear later, we will assume that the prover is trying convince the verifier that $\exists y \; \varphi(X, y)$; here we think of X as the input and y as the witness (compare Cook's Theorem, Lecture 3). The input X is known to the verifier. The variables and clauses of φ are named using strings in $\{0, 1, \ldots, h-1\}^m$ (from now on $[h]$ will stand for $\{0, 1, \ldots, h-1\}$). We will use the first component of the string to distinguish between input variables and witness variables – if the first component of the string is 0, then the string is the index of an input variable; otherwise it is the index of a witness variable.

- The prover provides us the assignment A for all variables. That is $A = (A_0, A_1)$ where A_0 assigns values to the input variables and A_1 to the witness variables. Then the proof will convince the verifier that A satisfies φ, and that the part A_0 in the assignment is consistent with the input value supplied for X.

Low degree extensions: Assume that $\mathcal{F}$ is a field containing $[h]$. In our application h will be much smaller than $|\mathcal{F}|$. Let $f : [h]^t \to \{0, 1\}$. Now f is defined only on a part of the domain $\mathcal{F}^t$. We wish to extend f and obtain a function $\hat{f}$ that is defined over the entire domain. Moreover, we want $\hat{f}$ to be represented by a low degree polynomial of t variables. This extension is obtained as follows. For $\sigma \in [h]^t$, let P_σ be the polynomial of degree $h-1$ in each variable, such that $P_\sigma(x) = 0$ for all $x \in [h]^t - \{\sigma\}$ and $P_\sigma(\sigma) = 1$. Such a polynomial exists and is unique (why?). Then, for $x \in \mathcal{F}^t$,

$$\hat{f}(x) = \sum_{\sigma \in [h]^t} P_\sigma(x) \cdot f(x). \tag{9.1}$$

We leave it to the reader to convince herself that $\hat{f}$ has the required properties.

In our application, we will use this method to arithmetize the expression φ. For example, we think of χ_1 as a function from $[h]^{2m}$ to $\{0, 1\}$. Then, $\hat{\chi}_1$ will denote the polynomial with $2m$ variables of degree at most $h-1$ in each variable that agrees with χ_1 everywhere in $[h]^{2m}$. Similarly, we may obtain low degree extensions $\hat{\chi}_1$, $\hat{\chi}_2$, $\hat{s}_1$, $\hat{s}_2$ and $\hat{s}_3$. Note that the polynomial representations of these functions can be obtained by the verifier using the formula (9.1) since she has complete information about these functions. The identity of the witness variable is determined entirely by the last $m-1$ components in its index. Thus, we may think of the input assignment A_0 as a function from $[h]^{m-1}$ to $\{0, 1\}$;, then the verifier can write down the polynomial representation of $\hat{A}_1$ (using variables $x_2, x_3, \ldots, x_m$).

The proof provided by the prover contains the low degree extensions $\hat{A}_0$ and $\hat{A}_1$. To distinguish between the prover's and the verifier's versions of $\hat{A}_0$, we refer to them as $\hat{A}_0^P$ and $\hat{A}_0^V$ respectively. Note that the low degree extension of A, that is $\hat{A}$, can be obtained from $\hat{A}_0$ and $\hat{A}_1$ using the formula

$$\hat{A}(x_1, x_2, \ldots, x_m) = (1 - x_1)\hat{A}_0^P(x_2, x_3, \ldots, x_m) + x_1 \hat{A}_1(x_2, x_3, \ldots, x_m). \tag{9.2}$$

Verification Assume that the prover has provided the assignment A in the form of a low degree extension. We assume that the tables for $\hat{A}_0$ and $\hat{A}_1$ have one entry for each element of $\mathcal{F}^{m-1}$. The verifier now needs to check that the assignment given by the prover satisfies φ. We now proceed to arithmetize this check. For each clause c, we need to verify that

$$\mathrm{CC}(c, A) = \sum_{v_1, v_2, v_3 \in [h]^m} \prod_{j=1}^{3} \chi_j(c, v_j)(s_j(c) - A(v_j)) = 0. \tag{9.3}$$

We need to somehow combine these conditions arithmetically and obtain a single sum. For this we pick random weights $R(c)$ for each clause c and obtain the linear combination of the equations (9.3) for the various clauses. That is, we check that

$$\sum_c R(c) \cdot \mathrm{CC}(c, A) = 0. \tag{9.4}$$

This last equation (9.4) is our arithmetization of 3-SAT.

9.2.2 The protocol

In this section, we will sketch the various steps in the protocol. The details, as well as the analysis, will be presented presented in the next lecture.

How to choose the random weights: The obvious strategy of independently picking one random weight for each clause c requires too many random bits. We will instead pick a small number of random numbers and use them to generate all the random weights.

Pick $r_1, r_2, \ldots, r_m \in \mathcal{F}$ randomly. We wish to use these numbers to generate one random weight $R(c)$ for each clause c. Recall that $c \in [h]^m$. Let $c = c_1, c_2, \ldots, c_m$. Then define

$$R(c) = r_1^{c_1} r_2^{c_2} \ldots r_m^{c_m}.$$

Note that R is determined if the random values $r_1, r_2, \ldots, r_m$ are fixed. Since R is a function from $[h]^m$ to $\mathcal{F}$, we may consider its low degree extension $\hat{R}(c)$. Now a representation of $\hat{R}(c)$ can be obtained using the formula (9.1). However, we may obtain it more naturally as follows. We write $R(c) = R_1(c_1)R_2(c_2)\ldots R_m(c_m)$, where $R_i(c_i) = r_i^{c_i}$. First, we construct the degree $h-1$ extension $\hat{R}(x_i)$ for each R_i. Then we multiply these extensions to obtain the extension $\hat{R}(x_1, x_2, \ldots, x_m)$. We only have to describe how the extensions $\hat{R}_i(x_i)$ are obtained. First observe that for $j = 0, 1, \ldots, h-1$, there exists a degree $h-1$ polynomial $p_j(x)$ such that $p_j(x) = 0$ for $j \in [h] - \{j\}$, and $p_j(j) = 1$. Indeed, we may take $p_j(x) = \prod_{i \in [h]-\{j\}} \frac{x-i}{j-i}$. Now, the required low degree polynomial $\hat{R}(x)$ is defined by $\hat{R}_i(x) = \sum_{j=0}^{h-1} p_j(x) r_i^j$.

Sum check: We now return to the verification of the equation (9.4). The idea is to first replace the various functions appearing in the equation by their low degree extensions and obtain

$$\sum_{c, v_1, v_2, v_3 \in [h]^m} \hat{R}(c) \prod_{j=1}^{3} \hat{\chi}_j(c, v_j)(\hat{s}_j(c) - \hat{A}(v_j)) = 0. \tag{9.5}$$

Note that the summation above can be broken into $4m$ summations, each ranging over $[h]$. Moreover, the summand is a polynomial of degree at most $7(h-1)$ in any variable. In the next lecture, we will apply the method used in the interactive protocol for $\#\mathcal{P}$ functions, to verify (9.5). Note that in the last step of the protocol the verifier is expected to compute the polynomial at a random point. While the polynomials $\hat{R}$, $\hat{\chi}_i$ and $\hat{s}$, can easily be computed at any point using the formula (9.1), the value of $\hat{A}$ will have to be computed by referring to the proof table and using the formula (9.2). However, are we not making an assumption here? Why is it not possible for the prover to give for $\hat{A}_0^P$ and $\hat{A}_1$ tables that do not correspond to any polynomial at all? In this case, the summand would no more correspond to a low degree polynomial, and our analysis, based as it is on the important property that polynomials of low degree cannot have many roots will not be correct. Thus it is important to ensure that the tables supplied by the prover indeed correspond to low degree polynomials.

Low degree test: It turns out that ensuring complete agreement with some low degree polynomial is not feasible. However, it will suffice for our purposes if we can ensure that there is some low degree polynomial that agrees with the table on a very large fraction (say 0.99) of the entries. This weaker condition can be enforced using low degree tests. We will study these low degree tests later in the course.

Consistency: As observed earlier, we need to verify that the extension provided by the prover for the input variables and the one constructed by the verifier himself are identical. To verify this, we pick a random point and check that the values in the two tables are identical.

Remarks

The multiple prover model is due to Ben-Or, Goldwasser, Kilian and Wigderson [BOGKW88]. The class $\mathrm{PCP}(r(n), q(n))$ was formally defined by Arora and Safra [AS92]; it was implicit in the work of Feige, Goldwasser, Lovász, Safra and Szegedy [FGL$^+$91]. Fortnow, Rompel and Sipser [FRS88] earlier studied a similar model called polynomial time probabilistic oracle Turing machine, and showed that it equivalent in power to MIP (Theorem 9.3). Theorem 9.4 was also shown in [FRS88]; the converse, Theorem 9.5, was shown by Babai, Fortnow and Lund [BFL90, BFL91] (see also Homework 3).

The result $\mathcal{NP} \subseteq \mathrm{PCP}(\log n, \mathrm{poly}\log n)$ is due to Arora and Safra [AS92]. Their proof builds on the works of Babai, Fortnow, Levin and Szegedy [BFLS91], who showed that $\mathcal{NP} \subseteq$ $\mathrm{poly}\log n, \mathrm{poly}\log n)$ and Feige, Goldwasser, Lovász, Safra and Szegedy [FGL$^+$91], who using a different approach showed that $\mathcal{NP} \subseteq \mathrm{PCP}(\log n \cdot \log\log n, \log n \cdot \log\log n)$.

Lecture 10

$\mathcal{NP} \subseteq \mathbf{PCP}(\log n, \mathbf{poly} \log\log n)$

Lecturer: Jaikumar Radhakrishnan **Date:** 17 March 1998

In this lecture, we shall first complete the proof of $\mathcal{NP} \subseteq \mathrm{PCP}(\log n, \mathrm{poly} \log n)$ that we started in the last class. In the second part of the lecture, we will strengthen this and show that $\mathcal{NP} \subseteq \mathrm{PCP}(\log n, \mathrm{poly} \log\log n)$.

10.1 $\mathcal{NP} \subseteq \mathbf{PCP}(\log n, \mathbf{poly} \log n)$ continued ...

In the last lecture, we sketched the different parts of the protocol; in this lecture, we treat them in detail.

Sum check: Recall that the prover is trying to convince the verifier that $\exists y\, \varphi(x,y)$. Here x corresponds the input vector known too the verifier. The prover is expected to convince the verifier of this by providing an assignment for all the variables in φ, such that the the value assigned for x is the input. We have already seen how one can arithmetize the statement $\exists y\, \varphi(x,y)$. We restate below the equation (9.5) that resulted from this arithmetization.

$$\sum_{c,v_1,v_2,v_3 \in [h]^m} \hat{R}(c) \prod_{j=1}^{3} \hat{\chi}_j(c,v_j)(\hat{s}_j(c) - \hat{A}(v_j)) = 0. \tag{10.1}$$

Recall that here $\hat{R}$, $\hat{\chi}_j$ and $\hat{s}_j$ are low degree extensions of functions that the verifier can compute using φ and the random bits (that go into the choice of $\hat{R}$); on the other hand $\hat{A}$ is the low degree extension obtained using the formula (9.2), which we restate below.

$$\hat{A}(x_1,x_2,\ldots,x_m) = (1-x_1)\hat{A}_0^P(x_2,x_3,\ldots,x_m) + x_1\hat{A}_1(x_2,x_3,\ldots,x_m). \tag{10.2}$$

Here, $\hat{A}_0^P$ and $\hat{A}_1$ are not known to the verifier; they are to be provided by the prover in the form of tables. These tables will have $\mathcal{F}^{m-1}$ entries, each an element of $\mathcal{F}$.

The proof of correctness of the protocol we are about to present makes an assumption about the tables $\hat{A}_0^P$ and $\hat{A}_1$: We require that they correspond to some low degree polynomials (of total degree at most mh). Of course, there is nothing that binds the prover to supply tables satisfying these conditions, and it is conceivable that the prover cheats the verifier by departing from this assumption. However, as a first step, our protocol will show how we can ascertain that a 3-CNF expression is satisfiable, provided the prover is somehow bound to supply tables that correspond to low degree polynomials. Later, we will augment the protocol with *low degree tests* so that this assumption is not necessary.

Denote the $4m$ variables of c, v_1, v_2, v_3 by $w_1, w_2, \ldots, w_{4m}$. We will write $Q(w_1, w_2, \ldots, w_{4m})$ instead of $\hat{R}(c)\prod_{j=1}^{3}\hat{\chi}_j(c,v_j)(\hat{s}_j(c) - \hat{A}(v_j))$. Thus (10.1) now becomes

$$\sum_{\langle w_1,w_2,\ldots,w_{4m}\rangle \in [h]^{4m}} Q(w_1,w_2,\ldots,w_{4m}) = 0. \tag{10.3}$$

For $1 \leq i \leq 4m$ and $\alpha_1, \alpha_2, \ldots, \alpha_{i-1} \in \mathcal{F}$, let

$$E_i(x) = \sum_{\langle w_{i+1}, w_{i+2}, \ldots, w_{4m}\rangle \in [h]^{4m-i}} Q(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}, x, w_{i+1}, \ldots, w_{4m}).$$

Note that the degree of $E_i(x)$ is at most the degree of w_i in Q, which is at most $7mh$.

0. Make $\hat{r}(c)$. Set $\beta_0 \leftarrow 0$, $m' = 4m$.
1. For $i = 1$ to m' do
   1.1 Read $\hat{E}_i(x)$, a polynomial over $\mathcal{F}$ of degree at most $7(h-1)$.
   1.2 Check that $\sum_{x \in [h]} \hat{E}_i(x) = \beta_{i-1}$; otherwise, Reject and Halt.
   1.3 Pick $\alpha_i \in \mathcal{F}$ at random, and set $\beta_i \leftarrow \hat{E}_i(\alpha_i)$.
2. Compute $\beta'_{m'} = Q(\alpha_1, \alpha_2, \ldots, \alpha_{m'})$ using the tables for $\hat{R}, \hat{\chi}_i, s_i, \hat{A}_0^P$ and $\hat{A}_1$, and check that $\beta_{m'} = \beta_m$; otherwise Reject and Halt.
3. Pick $\alpha \in \mathcal{F}^{m-1}$ at random and check that $\hat{A}_0^V(\alpha) = \hat{A}_0^P(\alpha)$; otherwise Reject and Halt.

10.1.1 Analysis

Error probability: We do not repeat the analysis of Lecture 7. It is clear that if the 3-CNF expression is satisfiable, then the prover can provide tables $\hat{A}_0^P$ and $\hat{A}_1$ (and suitable polynomials $\hat{E}_i$) that will cause the verifier to accept with probability 1.

Consider a 3-CNF expression that is not satisfiable. We wish to show that in this case the probability of acceptance is small. First, we show that if φ is not satisfiable, then with high probability (10.3) does not hold. Second, assuming that (10.3) does not hold, we bound the probability that the protocol accepts. Third, we consider the possibility that the input vector implicit in the table $\hat{A}_0^P$ is different from the actual input, and bound the probability that the protocol accepts in that case.

If A does not satisfy φ, then there exists a clause c such that $\mathrm{CC}(c, A) \neq 0$. Now consider equation (9.4) and view the left hand side as a polynomial in variables $r_1, r_2, \ldots, r_m$ (recall the definition of $R(c)$). Note that for different clauses c, the sum $\mathrm{CC}(c, A)$ is multiplied by a different monomial in variables $r_1, r_2, \ldots, r_m$. Thus, if some $\mathrm{CC}(c, A)$ is non-zero, then the left hand side of (9.4) is a non-trivial polynomial in variables $r_1, r_2, \ldots, r_m$ of total degree at most $m(h-1)$. It follows that the probability that (10.3) holds when the r_i are chosen randomly and independently is at most $m(h-1)/|\mathcal{F}|$.

After this the situation is identical to the one encountered in the protocol for $\#\mathcal{P}$. Using arguments similar to those used there, one can show that if (10.3) does not hold, then the probability that the protocol accepts is at most $4m \cdot 7mh/|\mathcal{F}|$.

Finally, we consider the consistency check between $\hat{A}_0^V$ and $\hat{A}_0^P$. If these two polynomials are not identical, the protocol accepts with probability at most $(m-1)(h-1)/|\mathcal{F}|$.

For the protocol to accept the input incorrectly, it must err in at least one of the three ways considered above. It follows that the probability of error is at most

$$\frac{m(h-1)}{|\mathcal{F}|} + \frac{(4m)(7mh)}{|\mathcal{F}|} + \frac{(m-1)(h-1)}{|\mathcal{F}|} \leq \frac{30m^2h}{|\mathcal{F}|}.$$

Random bits: First consider the number of random bits needed to construct weights $R(c)$. Recall that we need m random elements of $\mathcal{F}$ to produce R. Thus the total number of random bits used is $O(m \log |\mathcal{F}|)$.

Second, an element of $\mathcal{F}$ is chosen at random in each iteration of step 1.1. There are at most $m' = 4m$ iterations. Thus the number of random bits needed is again $O(m \log |\mathcal{F}|)$.

Finally, in Step 3 of the protocol, $\alpha \in \mathcal{F}^{m-1}$ is chosen at random. This requires $O(m \log |\mathcal{F}|)$ bits.

Thus, in all we need only $O(m \log |\mathcal{F}|$ bits to implement the protocol described above.

Query bits: The proof is read by the verifier in steps 1.1, 2 and 3. In each iteration of step 1.1, a polynomial of degree at most $7(h-1)$ is read. In the at most m' iterations of step 1.1, the total number of bits read is thus at most

$$m'(7(h-1)+1)\lceil \log |\mathcal{F}| \rceil.$$

In step 2, three entries each of the tables $\hat{A}_0^P$ and $\hat{A}_1^P$ are read, requiring at most $6\lceil \log |\mathcal{F}| \rceil$ bits.

Similarly, in step 3, the verifier reads at most $2\lceil \log |\mathcal{F}| \rceil$ bits. Thus, the total number of bits read is at most $30hm\lceil \log |\mathcal{F}| \rceil$.

We must now choose the value of $|\mathcal{F}|$, m and h such that the probability of error is small, the number of random bits used is $O(\log n)$ and the number of bits of the proof read is at most $\text{poly} \log n$. We choose h to be $\lceil \log n \rceil$; then m can be taken to be $\Theta(\log n / \log \log n)$ (to ensure that h^m is more than the number of variables and clauses.) To keep the probability of error small, we must choose $|\mathcal{F}|$ such that $30m^2h/|\mathcal{F}|$ is small. We choose $|\mathcal{F}|$ to be $\Theta((\log n)^3)$. Observe that this choice of parameters automatically ensures that the number of random bits used (that is, $O(m \log |\mathcal{F}|)$) is $O(\log n)$. Similarly, this ensures that the number bits of the proof read is $O((\log n)^2)$.

10.1.2 Low degree tests:

As discussed earlier, our analysis assumes that the prover provides perfect low degree extensions of the assignment and the input. We now wish to remove this constraint on the prover. Thus, the tables provided by the prover need not arise from any low degree polynomial. Instead, we shall augment our protocol with low degree tests, which will ensure that the tables provided by the prover are of good quality.

It would been ideal if we had tests that examined tables and certified (with high probability) that they correspond to some low degree polynomial; that is, such a test would reject, with high probability, any table that has even the slightest defect. However, it is not hard to see that any such test must always read almost the entire table; this is beyond our means, for we wish to read only $\text{poly} \log n$ bits in all.

The low degree test that we use will permit us to conclude that the tables provided by the prover are close to certain low degree polynomials. In other words, if there is no polynomial of low degree that agrees with the table on most of the entries, then the test would reject with high probability.

The details of this test will be presented in a later lecture (Lecture 12); we shall only summarize the various features of this test. Suppose the prover claims that the table T corresponds to a polynomial $p : \mathcal{F}^m \to \mathcal{F}$, such that the total degree of p is at most d.

We will require that the prover provide some additional information in support of her claim. The exact nature of this information will become clear later. For now, we need know only that this information is organized in locations, where each location holds a polynomial in one variable of degree at most d (that is, $O(d \log |\mathcal{F}|)$ bits).

For $f, g : \mathcal{F}^m \to \mathcal{F}$, let $\Delta_d(f,g) = |\{x \in \mathcal{F}^m : f(x) \neq g(x)\}|/|\mathcal{F}^m|$. For a table $T : \mathcal{F}^m \to \mathcal{F}$, let

$$\Delta_d(T) = \min\{\Delta(T,p) : p \text{ is a polynomial over } \mathcal{F} \text{ of total degree at most } d\}.$$

We need a procedure that when given a table T, a degree d, a probability of error $\delta > 0$, and a constant $\epsilon > 0$, guarantees that $\Delta_d(T) \leq \epsilon$. Our test operates as follows. The verifier uses at most $k_1(\epsilon, \delta) \cdot m \log |\mathcal{F}|$ random bits and reads $k_2(\epsilon, \delta)$ cells from the table and the supporting information. He then computes a polynomial time predicate on the information read and decides to accept or reject. The test has the following properties.

1. If the table and the supporting information are correct ($\Delta_d(T) = 0$), then the verifier accepts with probability 1.
2. If $\Delta_d \geq \epsilon$, then the probability that the verifier accepts is at most δ.
3. The verifier uses at most $k_1(\epsilon, \delta) m \log |\mathcal{F}|$ random bits and reads at most $k_2(\epsilon, \delta) d \log |\mathcal{F}|$ bits of the proof (from at most $k_2(\epsilon, \delta)$ different locations).

10.1.3 The revised protocol for $\mathcal{NP} \subseteq \mathbf{PCP}(\log n, \mathbf{poly} \log n)$

To the protocol we already have, we add two low degree tests, one each for $\hat{A}_0^P$ and $\hat{A}_1$. Let us first check that the number of random bits used and the number of bits of the proof read are within limits. The verifier uses $O(m \log |\mathcal{F}|)$ random bits, and reads $O(mh \log |\mathcal{F}|)$ bits. It is easy to check that for our choice of values, $m \log |\mathcal{F}| = O(\log n)$ and $mh \log |\mathcal{F}| = O((\log n)^2)$.

It remains only to prove the correctness of this revised protocol. Clearly, if the prover is honest the verifier of the revised protocol would also accept with probability 1. Now suppose the 3-CNF expression φ is not satisfiable. We have two case. In the first, one of the tables, $\hat{A}_0^P$ and $\hat{A}_1$, is not close to any polynomial of total degree at most mh. Then, the low degree test would reject the proof with probability at least $1 - \delta$. (Say $\delta = 0.1$.)

Thus, we need consider only the remaining case, where both the tables almost perfectly fit some low degree polynomials. We will argue that if these tables were replaced with tables that agree entirely with those polynomials, then the probability of acceptance will not change significantly. Since we have already determined that the probability of acceptance is small when then tables are perfect, it will follow that the probability of acceptance is small even when the tables are only close to perfect.

Fix $d = mh$, and assume $\Delta_d(\hat{A}_0^P), \Delta_d(\hat{A}_1) < \epsilon$ (ϵ to be chosen later) and $\hat{B}_0$ and $\hat{B}_1$ are the tables corresponding to polynomials that fit $\hat{A}_0^P$ and $\hat{A}_1$ (respectively) closely. (Note $\hat{B}_0$ and $\hat{B}_1$ are unique.) The tables $\hat{A}_0^P$ and $\hat{A}_1$ are probed by the verifier in steps 2 and 3. Observe that the total number of locations of these tables that are used by the prover is at most 7 (6 in step 2 and 1 in step 3). Moreover, the location of these probes are chosen randomly and are distributed uniformly over $\mathcal{F}^{m-1}$. If for none of these locations the value provided by the prover (in $\hat{A}_0^P$, $\hat{A}_1$) differs from the values in the corrected tables $\hat{B}_0, \hat{B}_1$, then the final decision of the verifier will be identical in the two cases. Thus,

$$\Pr[\text{The verifier accepts with } \hat{A}_0^P,\, \hat{A}_1] \leq \Pr[\text{The verifier accepts with } \hat{B}_0,\, \hat{B}_1] + 7\epsilon.$$

We have seen already that the first term on the right is $O(m^2 h/|\mathcal{F}|)$. By choosing $|\mathcal{F}| = \Theta((\log n)^3)$ and $\epsilon = 0.01$, we make this quantity less than 0.1.

If the input is encoded: For later application, we need to formulate a version of this protocol where not all bits of the input are required.

Notice that in the above protocol we need the input only in step 3 where we verify that the assignment is consistent with the input. We use a constant number of cells of its low degree extension $\hat{A}_0^V$, and for this we are required to read all the bits of the input. Now suppose that instead of the input a table for the low degree extension $\hat{A}_0^V$ itself is provided. Then, of course, we can just read the values directly and compute based on them. However, we must ensure that the table does correspond to some input. This is done by running low degree tests on the table. The resulting protocol has the following property. If the table provided does not resemble the low degree extension of any input, then we reject with high probability. Otherwise, we accept with high probability if the unique input x the table encodes is in the language (i.e. $\exists y\, \varphi(x,y)$), and reject with high probability if it is not. Instead of reading n bits of the input, the verifier now he reads only poly log log n bits from the table!

10.2 Reducing the number of probes

The protocol for $\mathcal{NP} \subseteq \mathrm{PCP}(\log n, \mathrm{poly}\log n)$ that we just saw uses $O(\log n)$ random bits of reads $O((\log n)^2)$ bits of the proof. The information read by the verifier is located in different parts of the proof. For example, consider the execution of step 1.1; in each of the m' iterations, a polynomial is read from a different location in the proof. We will need to modify this protocol so that the verifier is required to probe only a constant number of locations; however, we still restrict the number of random bits used to $O(\log n)$ and the number of bits of the proof read to poly $\log n$.

First, observe that the locations of the proof probed by the verifier depend only on the random bits; that is, the verifier does not use the actual contents previously read to compute addresses. Thus, if all the random bits are known, then the locations probed by the verifier are fixed.

Assume that on a certain execution of the protocol the verifier needs to know the value stored at locations $a_1, a_2, \ldots, a_k$ (these are addresses of the bits of the proof), where $k = O((\log n)^3)$. We encode these addresses as strings over the set $I = \{0, 1, \ldots, h-1\}$. Since the entire proof has only a polynomial number of bits, we use strings of length $d = O(\log n / \log h)$.

We now think of the proof as a function $\pi : I^d \to \{0,1\}$, and construct the extension of π, called $\hat{\pi} : \mathcal{Q}^d \to \mathcal{Q}$. Here $\mathcal{Q}$ is a large field containing I, and $\hat{\pi}$ is the low degree extension of of π; thus $\deg(\hat{\pi}) \le d|I|$. In order to reduce the number of probes, we now require the prover to supply the function $\hat{\pi}$ as a table.

The verifier makes use of this table as follows. First he picks $a_0 \in \mathcal{Q}^d$ randomly and fits polynomials for each coordinate of the a_i's. That is, for $i = 1, 2, \ldots, d$ and $j = 0, 1, \ldots, k$, he has polynomials $p_i(j) = a_j[i]$. (We shall choose $\mathcal{Q}$ such that $\{0, 1, \ldots, k\} \subseteq \mathcal{Q}$.) Note that each p_i is a polynomial (in $\mathcal{Q}[x]$) of degree at most k, and it can be evaluated at any point in $\mathcal{Q}$ by the verifier, who knows $a_0, a_1, \ldots, a_k$. Consider the function $F(t) = \hat{\pi}(p_1(t), p_2(t), \ldots, p_d(t))$. Since $\hat{\pi}$ has degree at most $d|I|$ and the p_i have degrees at most k, we have that $\deg(F) \le kd|I|$. We choose $|I| = \lceil \log n \rceil$ so that $\deg(F) \le (\log n)^3$. Note that F is completely determined once we fix the random bits of the original protocol r ($O(\log n)$ bits) and a_0 ($d \log |\mathcal{Q}|$ bits); the polynomial F corresponding to r and a_0 is referred to as $F_{a_0,r}$. In addition to the table $\hat{\pi}$ described above, the prover must provide of F (using at most $O(kd|I| \log |\mathcal{Q}|)$ bits for the $kd|I| + 1$ coefficients). Thus the new proof looks as shown in Figure 10.1.

Now, instead of probing the table at location $a_1, a_2, \ldots, a_k$, the prover consults the polynomial $F_{a_0,r}$ and uses the value $F_{a_0,r}(i)$ instead of $\pi(a_i)$. To keep the prover honest the verifier also checks that $F_{a_0,r}(t)$ and $\hat{\pi}(p_1(t), p_2(t), \ldots, p_d(t))$ are the same polynomials by comparing them at a random point. The details are presented in the following protocol.

The revised protocol: The verifier performs the following steps.

1. Pick r and a_0 at random. Using r compute the values of $a_1, a_2, \ldots, a_k$.
2. Read $F_{a_0,r}$ from the proof.

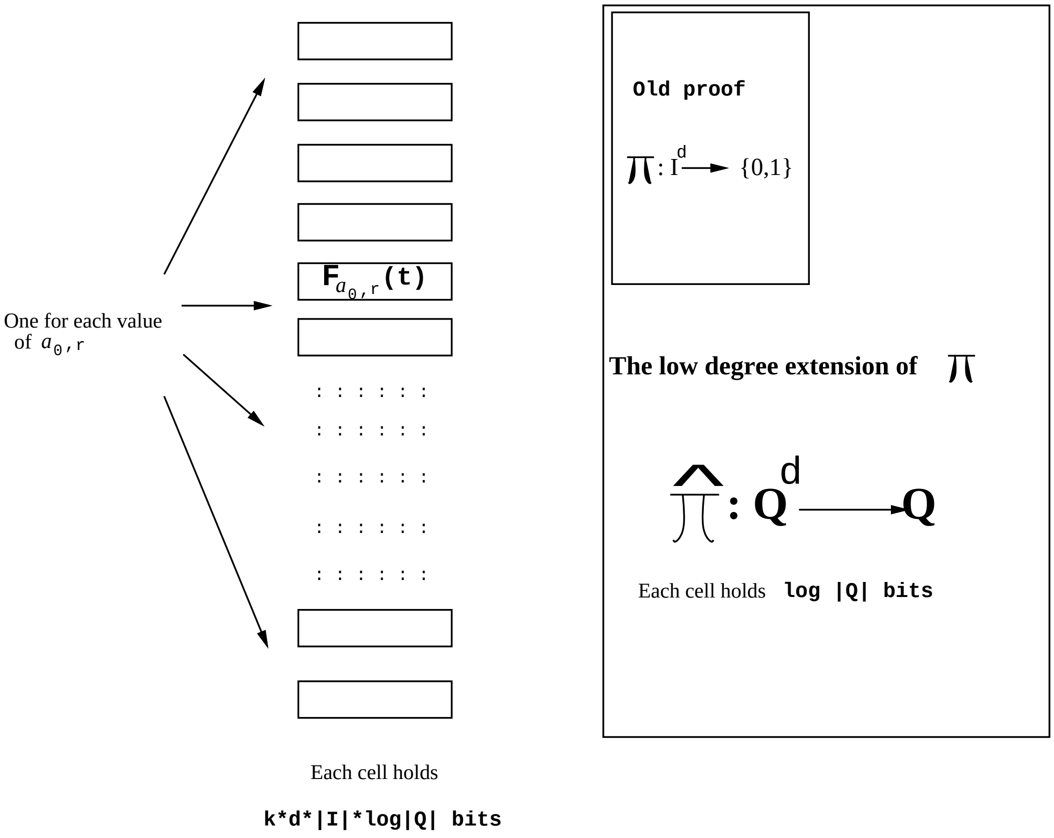

Figure 10.1: The new proof

3. Use the values $v_i = F_{a_0,r}(i)$, for $i = 1, 2, \ldots, k$, instead of reading $\pi(a_i)$ (as one would have in the original protocol). Check that the original protocol would have accepted with these values of v_i.
4. Pick $t^* \in \mathcal{Q}$ at random and verify that $F_{a_0,r}(t^*) = \hat{\pi}(p_1(t^*), p_2(t^*), \ldots, p_k(t^*))$.
5. Check that $\hat{\pi}$ is close to a low degree polynomial (using low degree test).

It is easy to see that all steps except the last can be executed by probing the proof in $O(1)$ locations; that even the last step requires only $O(1)$ proves will be verified later (Lecture 12).

We now argue that the revised protocol is correct. We show that for any input, the probability of acceptance of the new protocol is small whenever the probability of acceptance is small for the original protocol. (It is easy to see that if the verifier using the original protocol accepts with probability 1, then there exists a suitable proof for the revised protocol that would also be accepted with probability 1).

To simplify the analysis, let us make an assumption. Suppose the table for $\hat{\pi}$ perfectly fits a low degree polynomial. In this case it is clear that if the polynomials $F_{a_0,r}$ provided by the prover are correct then the verifier would reject with high probability (equal to the probability when the verifier uses the old protocol and the table $\hat{\pi}$ restricted to I^d). Suppose for some r and a_0, $F_{a_0,r}(i) \neq \hat{\pi}(a_i)$ for some $i \in \{1, 2, \ldots, k\}$. In that case, $F_{a_0,r}(t)$ and $\hat{\pi}(p_1(t), \ldots, p_k(t))$ are different polynomials (because they differ at $t = i$), and consequently differ on all but $\deg(F)$ values of t. Thus if $F_{a_0,r}$ is not supplied correctly, then step 4 of the revised protocol would reject with high probability. It follows that the probability that the new protocol rejects is at least

$$\text{(probability that the old protocol rejects } \hat{\pi} \text{ restricted to } I^d) \cdot (1 - \frac{\deg(F)}{\mathcal{Q}}),$$

and we are done (provided we choose $|\mathcal{Q}| >> \deg(F)$).

Let us now consider what happens if the assumption that the table for $\hat{\pi}$ perfectly fits a low degree polynomial is removed. Using the low degree test in step 5, we can still ensure that $\hat{\pi}$ is ϵ-close (for some small $\epsilon > 0$ to be chosen later) to a low degree polynomial. Let $\hat{\pi}^*$ be the unique low degree function close to $\hat{\pi}$. We then have

$$\Pr_{a \in \mathcal{Q}^d}[\hat{\pi}(a) = \hat{\pi}^*(a)] > 1 - \epsilon.$$

Now, observe that if $t^* \notin \{0, 1, 2, \ldots, k\}$ then $\langle p_1(t^*), p_2(t^*), \ldots, p_d(t^*)\rangle$ is a random element of $\mathcal{Q}^d$ (Why? Hint: a_0 is chosen randomly). Hence, from the point of view of the revised protocol, the tables $\hat{\pi}$ and $\hat{\pi}^*$ are essentially indistinguishable in step 4, and the conclusion of the above paragraph continue to hold. We make this precise as follows.

Let $p(t^*) = \langle p_1(t^*), p_2(t^*), \ldots, p_d(t^*)\rangle$. Since two distinct polynomials of degree at most $kd|I|$ may agree on at most $kd|I|$ values, we have

$$\Pr_{t^* \in \mathcal{Q}}[F(t^*) \neq \hat{\pi}^*(p(t^*)) \mid \exists i\, F(i) \neq \hat{\pi}^*(a_i)] \leq kd|I|/|\mathcal{Q}|.$$

Thus,

$$\begin{aligned}
\Pr[\text{The revised protocol accepts}] \leq\ & \Pr[(\text{The original protocol accepts } \hat{\pi}^* \text{ restricted to } I^d) \\
& \textbf{or } (\exists i\, F(i) \neq \hat{\pi}^*(a_i) \text{ and the revised protocol accepts})] \\
\leq\ & \Pr[\text{The original protocol accepts } \hat{\pi}^* \text{ restricted to } I^d] \\
& + \Pr[\exists i\, F(i) \neq \hat{\pi}^*(a_i) \text{ and the revised protocol accepts}].
\end{aligned}$$

We thus need to bound the second term above. This is at most

$$\begin{aligned}
& \Pr[\hat{\pi}(p(t^*)) \neq \hat{\pi}^*(p(t^*))] \\
& + \Pr[\exists i\, F(i) \neq \hat{\pi}^*(a_i) \text{ and } \hat{\pi}(p(t^*)) = \hat{\pi}^*(p(t^*)) \text{ and the revised protocol accepts}].
\end{aligned}$$

Since $\hat{\pi}$ and $\hat{\pi}^*$ are ϵ-close, the probability that they differ on a random point in $\mathcal{Q}^d$ is at most ϵ. Since for all values of t^* outside $\{0, 1, 2, \ldots, k\}$, $p(t^*)$ is a random point in $\mathcal{Q}^d$. Thus the first term is at most $\epsilon + (k+1)/|Q|$. The second term is at most

$$\begin{aligned}
& \Pr[\hat{\pi}(p(t^*)) = \hat{\pi}^*(p(t^*)) \text{ and the revised protocol accepts} \mid \exists i\, F(i) \neq \hat{\pi}^*(a_i)] \\
\leq\ & \Pr[F(t^*) = \hat{\pi}^*(p(t^*)) \mid \exists F(i) \neq \pi^*(a_i)] \quad \{\text{because the protocol checks } F(t^*) = \hat{\pi}^*(p(t^*))\} \\
\leq\ & kd|I|/|Q|.
\end{aligned}$$

Thus the probability that the revised protocol accepts can be more than the probability that the original protocol accepts by at most

$$\epsilon + (k+1)/|\mathcal{Q}| + kd|I|/|\mathcal{Q}|.$$

To make this quantity small we choose $\mathcal{Q} = \Theta((\log n)^5)$.

10.3 $\mathcal{NP} \subseteq \mathbf{PCP}(\log n, \mathbf{poly} \log\log n)$

We begin by summarizing the discussion of the previous section. The main features of the protocol showing $\mathcal{NP} \subseteq \text{PCP}(\log n, \text{poly} \log n)$ can be stated as follows. The verifier first tosses $O(\log n)$ coins to obtain a random string r. Then, based on $\overline{r}$ and the input length, the verifier constructs a value $\overline{a}$ of length poly $\log n$ (the entries of $\hat{R}$, $\hat{\chi}_i$ and $\hat{s}$ needing the last step of the sum-check protocol; based on $\overline{r}$ and the input, the verifier produces a constant number of entries of the low degree extension of the input assignment (call these bits $\overline{b}$); and based on $\overline{r}$ reads a constant number of cells of the proof. Call these values $\overline{c}_1, \overline{c}_2, \ldots, \overline{c}_t$. After this the verifier checks that $\langle \overline{r}, \overline{a}, \overline{b}, \overline{c}_1, \ldots, \overline{c}_t \rangle$, a string of length poly $\log n$, satisfies polynomial time predicate (or, belongs to, say, the language $L' \in \mathcal{P}$).

Observe that the protocol makes use of the values $\overline{c}_1, \overline{c}_2, \ldots, \overline{c}_t$ only while testing membership in the language L'. The main idea now is to check this last condition without reading all of $\overline{c}_1, \overline{c}_2, \ldots, \overline{c}_t$. We have seen above that membership in a language in $\mathcal{NP}$ can be checked using just a few entries of the low degree extension of the input (whose combined length is much less than the length of the actual input). Since for the task that we now have at hand the $\overline{c}_i$'s serve as input, it is conceivable that we may be able to effect considerable savings if the prover provides the low degree extensions of the $\overline{c}_i$'s instead of their actual values. However, to implement this idea we need to take care of certain technical difficulties.

1. When we had described the efficient protocol under the assumption that the input was presented as a low degree extension, we had assumed that the low degree extension of the entire input was available. In our case, the entire (fragmented) input is $\overline{z} = \langle \overline{r}, \overline{a}, \overline{b}, \overline{c}_1, \ldots, \overline{c}_t \rangle$, and we therefore need the low degree extension $\hat{z}$. Assume that the prover provides us the low degree extensions of $\overline{c}_1, \overline{c}_2, \ldots, \overline{c}_t$, that is, $\hat{\chi}_1, \hat{\chi}_2, \ldots, \hat{\chi}_t$; the low degree extensions of $\overline{r}$, $\overline{a}$ and $\overline{b}$, that is, $\hat{r}$, $\hat{a}$ and $\hat{b}$, can be 'constructed' by the verifier himself. Now the low degree extension of $\overline{z}$ can be constructed as follows. Rename $\langle \overline{r}, \overline{a}, \overline{b}, \overline{c}_1, \ldots, \overline{c}_t \rangle$ as $\langle \overline{z}_0, \overline{z}_1, \ldots, \overline{z}_{t+1} \rangle$ and assume $|\overline{z}_i| = h^l$, so that $|z| = (t+1)h^l$. Since $t+1$ is a constant much smaller than h, we think of $\overline{z} : [h]^{l+1} \to \{0,1\}$.

Then, for $v_1 \in \mathcal{F}$ and $v_2 \in \mathcal{F}^l$, we define $\hat{z}$ as follows.

$$\hat{z}(v_1, v_2) = \sum_{i=0}^{t+1} p_i(v_1) \cdot \hat{z}_i(v_2). \tag{10.4}$$

In other words, to evaluate $\hat{z}$ at the point $\overline{v} = \langle v_1, v_2 \rangle$ we need to read one entry each of $\hat{z}_0, \hat{z}_1, \ldots, \hat{z}_{t+1}$. Here,

$$p_i(x) = \prod_{j=0, j\neq i}^{t+1} (x-j) / \prod_{j=0, j\neq i}^{t+1} (i-j).$$

2. The method described above gives us the value of $\hat{z}(\overline{v})$ at any desired point in $\overline{v} \in \mathcal{F}^{l+1}$. However, for $\hat{z}(\overline{v})$ to be a low degree extension, the $\hat{z}_i$ must be low degree extensions. We must, therefore, run low degree tests for each $\hat{\chi}_i$ to ensure that it is sufficiently close to low degree. This requires only a constant number of cells to be read, and since we are now working over elements of a field of size poly $\log n$, the number of bits read is poly $\log\log n$.

With this we are in a position to prove that membership proofs for every language in $\mathcal{NP}$ can be checked by the verifier using $O(\log n)$ random bits and $O(\text{poly} \log\log n)$ bits of the proof while reading a constant number of cells of the proof. Fix a language $L \in \mathcal{NP}$. The verifier begins as in the proof of the revised protocol for $L \in \text{PCP}(\log n, \text{poly} \log n)$ which was summarized at the beginning of this section. After tossing $O(\log n)$ coins, the verifier has in his possession the values of $\overline{r}, \overline{a}, \overline{b}$ and the locations where the remaining values $\overline{c}_1, \overline{c}_2, \ldots, \overline{c}_t$

are to be found. However, in our present proof, in the locations corresponding to $\overline{c}_i$ their low degree extensions are stored. Now the verifier has to deal with the problem of verifying that $\langle \overline{r}, \overline{a}, \overline{b}, \overline{c}_1, \overline{c}_2, \ldots, \overline{c}_t \rangle \in L'$, where $L' \in \mathcal{P}$. Since this language is in $\mathcal{NP}$, the verifier uses the same protocol recursively. That is, he tosses $O(\log\log n)$ coins to generate a random string $\overline{r'}$. Based on this and the input length, he constructs $\overline{a'}$. He then obtains $\overline{b'}$ by using the formula (10.4). Finally, he reads the t locations in the proof (one of the auxiliary proofs that the prover provides for the language L', one for each coin toss sequence $\overline{r}$), each of length poly $\log\log n$. This way the prover collects a string $\langle \overline{r'}, \overline{a'}, \overline{b'}, \overline{c'_1}, \ldots, \overline{c'_{t'}} \rangle$ of length poly $\log\log n$ on which he performs a polynomial time computation. In, addition he checks that the tables for $\hat{z}_1, \hat{z}_2, \ldots, \hat{z}_t$ are close to linear, by running the low degree tests on them. For this he needs $O(\log\log n)$ random bits, and a constant number of probes.

Overall the number of cells probed is constant (independent of n). Each cell contains poly $\log\log n$ bits. The number of random bits required is $O(\log n)$ for the primary (top level) protocol and $O(\log\log n)$ bits for the the secondary protocol and the extra low degree tests. Hence the total number of random bits is still $O(\log n)$.

SUMMARY

1. The verifier tosses $O(\log n)$ coins to produce the random string $\overline{r}$.
2. He constructs the input $\overline{x} = \langle \overline{r'}, \overline{a'}, \overline{b'}, \overline{c'}_1, \ldots, \overline{c'}_{t'} \rangle$, where $\overline{a'}$ and $\overline{b'}$ do not depend on the bits supplied by the prover, and the locations where $\overline{c'}_1, \overline{c'}_2, \ldots, \overline{c'}_t$ are stored depend only on the random string $\overline{r}$ ($|\overline{c'}_i| =$ poly $\log\log n$).
3. He computes a polynomial time predicate on $\overline{x}$ and decides to accept or reject.

Remarks

The arithmetization employed in this lecture is taken from [FGL$^+$91]. The use of base h encoding instead of binary encoding first appeared in [BFLS91] and was again employed in [AS92] to maintain the number of random bits used at $O(\log n)$. The use of recursion (a key ingredient in all later results) also appeared in [AS92].

The method described in the lecture to restrict the number of probes to a constant is due to Arora, Lund, Motwani, Sudan and Szegedy [ALM$^+$92] (see also Feige and Lovász [FL92]).

Lecture 11

$\mathcal{NP} \subseteq \mathbf{PCP}(\mathbf{poly}, 1)$

Lecturer: Jaikumar Radhakrishnan **Date:** 24 March 1998

The protocol we describe below uses many more random bits than the one we saw last time; but it has the advantage that the number of probes to the proof is a constant.

As before, we imagine that the prover wishes to convince the verifier that $\exists y\, \varphi(x, y)$, where the input x and the 3-SAT expression φ are known to the verifier. The proof provided by the prover will correspond to the satisfying assignment $\langle \overline{x}, \overline{y} \rangle$ for φ. The verifier needs to be convinced of the following.

(a) $\langle \overline{x}, \overline{y} \rangle$ satisfies φ.

(b) $\overline{x}$ corresponds to the input he has.

We next show how (a) and (b) can be ensured

Let $\overline{a}$ denote the combine assignment $\langle \overline{x}, \overline{y} \rangle$ and let $|\overline{a}| = n$ (that is $\overline{a} \in \{0,1\}^n$). Consider a clause C_j of φ. First, we arithmetize this clause and obtain a polynomial $\hat{C}_j(z)$. For example, if $C_j \equiv (x_1 \vee \neg y_2 \vee x_3)$ then $\hat{C}_j = (1 - x_1)y_2(1 - x_3)$. So that $C_j(\overline{a})$ is *true* iff $\hat{C}(\overline{a}) = 0$. Now the verifier needs to check that the assignment $\overline{a}$ provided by the prover satisfies

$$\forall j\, \hat{C}_j(\overline{a}) = 0.$$

The verifier combines these conditions as a weighted sum (parity) and obtains

$$\sum_j r_j \hat{C}_j(\overline{a}) = 0,$$

by choosing the weights $r_j \in \{0,1\}$ uniformly and independently. Thus, if $\exists j\, \hat{C}_j(\overline{a}) \neq 0$, then

$$\Pr_r[\sum_j r_j \hat{C}_j(\overline{a}) \neq 0] = \frac{1}{2}.$$

By collecting terms, the verifier transforms the sum as follows

$$\begin{aligned}
\sum_j r_j \hat{C}_j(\overline{a}) &= \mathcal{E} + \sum_{i \in S_1} \alpha_i a_i + \sum_{\langle i,j \rangle \in S_2} a_i a_j \sum_{\langle i,j,k \rangle \in S_3} a_i a_j a_k \\
&= \mathcal{E} + \sum_{i \in S_1} a_i + \sum_{\langle i,j \rangle \in S_2} b_{ij} + \sum_{\langle i,j,k \rangle \in S_3} c_{ijk} \\
&= \mathcal{E} + \sum_i \alpha_i a_i + \sum_{\langle i,j \rangle \in S_2} \beta_{ij} b_{ij} + \sum_{\langle i,j,k \rangle \in S_3} \gamma_{ijk} c_{ijk}, \qquad (11.1)
\end{aligned}$$

where

1. $\mathcal{E}$ is the constant term (either 0 or 1);

Input: k_1, k_2 and tables $\tilde{A}$, $\tilde{B}$ and $\tilde{C}$.

Output: PASS if and only if the following tests succeed.

Linearity tests: Repeat k_1 times
Pick $x, x' \in \{0,1\}^n$ at random
Verify that $\tilde{A}(x) + \tilde{A}(x') = \tilde{A}(x + x')$
Pick $y, y' \in \{0,1\}^{n^2}$ at random
Verify that $\tilde{B}(x) + \tilde{B}(x') = \tilde{B}(x + x')$
Pick $z, z' \in \{0,1\}^{n^3}$ at random
Verify that $\tilde{C}(z) + \tilde{C}(z') = \tilde{C}(z + z')$

Consistency tests: Repeat k_2 times
Pick $x, y \in \{0,1\}^n$ at random
Verify that SC-$A(x)$ $\times$ SC-$A(y)$ $=$ SC-$B(x \circ y)$
Pick $x \in \{0,1\}^n$ and $y \in \{0,1\}^{n^2}$ at random
Verify that SC-$A(x)$ $\times$ SC-$B(y)$ $=$ SC-$C(x \circ y)$

Figure 11.1: Testing for linearity and consistency

SC-$A(x)$: Pick $r \in \{0,1\}^n$ at random; Return $\tilde{A}(r) + \tilde{A}(x - r)$.

SC-$B(y)$: Pick $r \in \{0,1\}^{n^2}$ at random; Return $\tilde{B}(r) + \tilde{B}(y - r)$.

SC-$C(z)$: Pick $r \in \{0,1\}^{n^3}$ at random; Return $\tilde{C}(r) + \tilde{C}(z - r)$.

Figure 11.2: Self-correction

2. $\overline{b} \in \{0,1\}^{n^2}$ and $\overline{c} \in \{0,1\}^{n^3}$ be defined by $b_{ij} = a_i a_j$ and $c_{ijk} = a_i a_j a_k$ for $i, j, k = 1, 2, \ldots, n$; and

3. α, β and γ are the characteristic vectors of the sets S_1, S_2 and S_3 respectively.

Note that $\mathcal{E}$ and the sets S_1, S_2 and S_3 depend only on φ and $\overline{r}$.

The prover shall provide the values for $\sum_{i \in S_1} a_i$, $\sum_{\langle i,j \rangle \in S_2} b_{ij}$ $\sum_{\langle i,j,k \rangle \in S_3} c_{ijk}$ satisfying (11.1), and supply a proof that they arise from some common assignment. We now describe the mechanism for implementing this. Fix an assignment $\overline{a}$ and recall the definitions of $\overline{b}$ and $\overline{c}$ given above. The prover supplies

1. $A : \{0,1\}^n \to \{0,1\}$ defined by $A(\alpha) = \sum_{i=1}^n \alpha_i a_i$;

2. $B : \{0,1\}^{n^2} \to \{0,1\}$ defined by $B(\beta) = \sum_{i,j=1}^n \beta_{ij} b_{ij}$.

3. $C : \{0,1\}^{n^3} \to \{0,1\}$ defined by $C(\gamma) = \sum_{i,j,k=1}^n \gamma_{ijk} c_{ijk}$.

These functions are to be provided in the form of tables containing one bit for each input vector. Thus, the proof consists of $2^n + 2^{n^2} + 2^{n^3}$ bits. The functions A, B and C as defined above are linear functions and depend only on the value of the assignment $\overline{a}$. To ensure that the tables $\tilde{A}$, $\tilde{B}$ and $\tilde{C}$ provided by the prover are also (close to) linear functions the verifier uses the linearity tests shown in Figure 11.1. (For vectors $x, y \in \{0,1\}^n$, $x \circ y$ is a vector of size n^2 whose components are indexed by pairs (i, j), $i, j = 1, 2, \ldots, n$; it is defined by $(x \circ y)_{ij} = x_i y_j$.) Furthermore, by the consistency checks given in Figure 11.1, the verifier checks that the tables are mutually compatible. To prove the correctness of the linearity tests, we will need the following lemma, whose proof will be discussed in the next section (Lemma 11.5). For $f, \hat{f} : \{0,1\}^n \to \{0,1\}$, let $\Delta(f, \hat{f}) = |\{\overline{x} \in \{0,1\}^m : f(x) \neq \hat{f}(x)\}|/2^n$.

Lemma 11.5 Suppose $\delta \leq 1/3$. If the function $f : \{0,1\}^m \to \{0,1\}$ satisfies

$$\Pr_{\hat{x},\hat{y} \in \{0,1\}^m}[f(\hat{x} + \hat{y}) = f(\hat{x}) + f(\hat{y})] \geq 1 - \delta/2,$$

then there exists a linear function $\hat{f} : \{0,1\}^m \to \{0,1\}$ such that $\Delta(f, \hat{f}) \leq \delta$.

The following lemma allows us to compute a linear function correctly (with high probability) even when the table supplied for it contains some errors. This method is known as *self-correction* (see Figure 11.2).

Lemma 11.1 *Suppose $f, \hat{f} : \{0,1\}^n \to \{0,1\}$, $\hat{f}$ is a linear function and $\Delta(f, \hat{f}) \leq \delta$. Then, for all $x \in \{0,1\}^n$,*

$$\Pr_{r \in \{0,1\}^n}[f(x + r) - f(r) \neq \hat{f}(x)] \leq 2\delta.$$

Proof. Observe that since $\hat{f}$ is a linear function, $\hat{f}(x + r) - \hat{f}(r) = \hat{f}(x)$ and $\Pr[(f(x + r) \neq \hat{f}(x + r)) \vee (f(r) \neq \hat{f}(r))] \leq 2\delta$. ∎

Lemma 11.2 *Fix $\delta, p \in (0, 1)$. There exist constants k_1, k_2 such that, if there does not exist any $\overline{a} \in \{0,1\}^n$. such that $\Delta(A(\overline{a}), \tilde{A}), \Delta(B(\overline{a}), \tilde{B}), \Delta(C(\overline{a}), \tilde{C}) \leq \delta$, then the scheme described in figures 11.1 and 11.2 rejects with probability at least p.*

Proof. We first choose k_1 so that if any of $\tilde{A}$, $\tilde{B}$ and $\tilde{C}$ is not ϵ-close to some linear function, then the linearity check rejects with probability at least p.

By Lemma 11.5, if $\tilde{f}$ (which may be $\tilde{A}$, $\tilde{B}$ or $\tilde{C}$) is not ϵ-close to linear then the linearity test accepts with probability at most

$$(1 - \epsilon/2)^{k_1} \leq \exp(-k_1 \epsilon/2).$$

To make this quantity at most $1 - p$ we choose k_1 large, that is, $k_1 \geq (2/\epsilon) \ln(1/(1 - p))$.

Thus, we may assume that $\tilde{A}$, $\tilde{B}$ and $\tilde{C}$ are ϵ-close to linear. That is, there exist vectors $\overline{a} \in \{0,1\}^n$, $\overline{b} \in \{0,1\}^{n^2}$ and $\overline{c} \in \{0,1\}^{n^3}$ such that $\tilde{A}$, $\tilde{B}$ and $\tilde{C}$ are ϵ-close to the linear functions described by the coefficients $\overline{a}$ $\overline{b}$ and $\overline{c}$ respectively. We next show that the consistency check rejects with probability at least p if $b_{ij} \neq a_i a_j$ for some i, j, or $c_{ijk} \neq a_i a_j a_k$, for some i, j, k.

Fact 11.3 *If X and Y are $n \times n$ $\{0,1\}$-matrices and if $X \neq Y$, then*

$$\Pr_{x \in \{0,1\}^n}[x^T X \neq x^T Y] \geq \frac{1}{2};$$

hence,

$$\Pr_{x,y \in \{0,1\}^n}[x^T X y \neq x^T Y y] \geq \frac{1}{4}.$$

In our first application, we have $X = (a_i a_j)$ and $Y = (b_{ij})$. Note that $x^T(a_i a_j)y = A(x)A(y)$, and $x^T Y y = B(x \circ y)$. Although we do not have any means of obtaining $A(x)$ and $A(y)$ directly, we can use $\tilde{A}$ to obtain these values. Recall that A and $\tilde{A}$ are ϵ-close, so that

$$\Pr_{x,y \in \{0,1\}^n}[A(x) \neq \tilde{A}(x) \mathbf{or}\ A(y) \neq \tilde{A}(y)] \leq 2\epsilon.$$

Using Lemma 11.1, we see that

$$\Pr[\text{SC-}B(x \circ y) \neq B(x \circ y)] \leq 2\epsilon.$$

Thus, if $b_{ij} \neq a_i a_j$ for some i, j, then, using Fact 11.3, we conclude that the consistency test accepts with probability at most $(3/4 + 4\epsilon)^{k_2}$. We set $\epsilon = \min\{\delta, 0.05\}$, so that this quantity is at most

$$(4/5)^{k_2} \leq 1 - p,$$

if

$$k_2 \geq 5 \ln(1/(1-p)).$$

A similar argument leads us to the conclusion that if $c_{ijk} \neq a_i a_j a_k$ for some i, j, k, then the test rejects with probability p. ■

We thus have a mechanism to ensure that the three tables provided by the prover are ϵ-close (if not identical) to tables arising from some assignment $\overline{a}$. Assume that the three tables have passed the linearity and consistency checks, so that (using Lemma 11.2 if the three tables are not are not ϵ-close to tables arising from some assignment then the test reject with probability at least p (ϵ and p will be chosen later).

Let us now return to the problem of verifying that $\overline{a}$ is a satisfying assignment of φ, that is, the equation (11.1). For this we need the values the values $A(\alpha)$, $B(\beta)$ and $C(\gamma)$. We cannot use tables $\tilde{A}$, $\tilde{B}$ and $\tilde{C}$ with confidence, because α, β and γ are not necessarily truly random vectors and the tables may return erroneous values at the locations we probe. Instead, we use SC-$A(\alpha)$, SC-$B(\beta)$ and SC-$C(\gamma)$; by Lemma 11.1, the probability of error in this step is at most 6ϵ. Suppose $\overline{a}$ s not a satisfying assignment. If we repeat this test k_3 times, then we accept with probability at most

$$(1-p) + (1/2 + 6\epsilon)^{k_3}.$$

By Lemma 11.2, we can make $1 - p$ and ϵ arbitrarily small and choose k_3 so that the overall probability of error is small. Simultaneously, we can ensure that the three table encoding the assignment are ϵ-close to a linear function (for any $\epsilon > 0$).

We can thus ensure that the assignment $\overline{a}$ encoded in the tables does satisfy the 3-SAT expression φ. Recall that the prover has to convince the verifier that there exists an assignment $\langle x, y \rangle$ were the first component x is the same as the input $\tilde{x}$ that is given. Let $|x| = |\tilde{x}| = n$. We again use the fact that if $x \neq \tilde{x}$, then

$$\Pr_z[x^T z \neq \tilde{x}^T z] = 1/2.$$

To use this fact, we need to compute $x^T y$ and $\tilde{x}^T y$ for a randomly chosen vector y. Since $\tilde{x}$ is available to us, we can compute $\tilde{x}^T y$ directly. On the other hand, our access to x is only through the table $\tilde{A}$ that encodes $\langle x, y \rangle$. We use the self-corrector SC-$A(z0^{n-n_1})$. Thus, if $x \neq \tilde{x}$ then the probability of acceptance is at most $1/2 + 2\epsilon$. This probability of error can be made small by repeating the test many times.

For later application, we need to consider the case where the input $\tilde{x}$ is not provided to us as a string of length n_1 but as a table of 2^{n_1} bits corresponding to the 2^{n_1} values of $\tilde{x}^T z$ as z varies.

We need to verify two things. First, that the table provided to us as input does correspond to some string of length n_1. This is achieved by means of the linearity test; we ensure that the it is ϵ-close to a linear function. Second, we need to the value for $\tilde{x}^T y$ (as above) for checking that the assignment supplied is consistent with the input. We read this value directly from the table supplied to us (since the vector y is random, we need not employ self-correction). In this case the entire protocol (including the probe for computing $\tilde{x}^T y$) requires only a constant number of bits from the proof.

The number of random bits used in $O(n^3)$. To see this, observe that the linearity and consistency tests use random vectors of length at most $O(n^3)$. Also the number of random weights used is at most the number of clauses in φ, which we may assume is $O(n^3)$. To verify that the assignment is compatible with the input we use $O(n_1)$ random bits. Thus, for any constant $\epsilon > 0$ we can ensure that an incorrect input is accepted with probability at most ϵ using $O(n^3)$ bits.

Theorem 11.4 *$\mathcal{NP} \subseteq PCP(n^3, 1)$.*

11.1 Testing a linear function

Lemma 11.5 *Suppose $\delta \leq 1/3$. If the function $f : \{0,1\}^m \to \{0,1\}$ satisfies*

$$\Pr_{\hat{x},\hat{y}\in\{0,1\}^m}[f(\hat{x}+\hat{y}) = f(\hat{x}) + f(\hat{y})] \geq 1 - \delta/2,$$

then there exists a linear function $\hat{f} : \{0,1\}^m \to \{0,1\}$ such that $\Delta(f, \hat{f}) \leq \delta$.

We need to establish that if f satisfies the hypothesis of the lemma, then there exists a linear function $\hat{f}$ close to it. We have already seen how the value of this linear function can be reconstructed from the values for $\hat{f}$ using self correction. This motivates the following definition.

$$\hat{f}(\hat{x}) = \text{Maj}\{f(\hat{x}+\hat{r}) - f(\hat{r})\}_{\hat{r}\in\{0,1\}^m}.$$

We are now faced with two problems. We need to show that:

1. $\Delta(f, \hat{f}) \leq \delta$, that is, $\Pr_{\hat{x}\in\{0,1\}^m}[f(\hat{x}) \neq \hat{f}(\hat{x})] \leq \delta$.

2. $\hat{f}$ is a linear function.

Claim 11.6 $\Pr_{\hat{x}\in\{0,1\}^m}[f(\hat{x}) \neq \hat{f}(\hat{x})] \leq \delta$.

Proof. By our assumption on f, we have $\Pr_{\hat{x},\hat{y}\in\{0,1\}^m}[f(\hat{x}+\hat{y}) \neq f(\hat{x}) + f(\hat{y})] \leq \delta/2$. Now,

$$\begin{aligned}
\Pr_{\hat{x},hy}[f(\hat{x}+\hat{y}) \neq f(\hat{x}) + f(\hat{y})] &= \sum_{\hat{a}\in\{0,1\}^m} \Pr_{\hat{x}}[\hat{x} = \hat{a}] \cdot \Pr_{\hat{x},\hat{y}}[f(\hat{x}+\hat{y}) \neq f(\hat{x}) + f(\hat{y}) \mid \hat{x} = \hat{a}] \\
&= \sum_{\hat{a}\in\{0,1\}^m} \Pr_{\hat{x}}[\hat{x} = \hat{a}] \cdot \Pr_{\hat{y}}[f(\hat{a}+\hat{y}) \neq f(\hat{a}) + f(\hat{y})] \\
&\geq \sum_{\hat{a}:f(\hat{a})\neq\hat{f}(\hat{a})} \Pr_{\hat{x}}[\hat{x} = \hat{a}] \cdot \Pr_{\hat{y}}[f(\hat{a}+\hat{y}) \neq f(\hat{a}) + f(\hat{y})].
\end{aligned}$$

It follows from the definition of $\hat{f}$ that if $\hat{f}(\hat{a}) \neq f(\hat{a})$ then

$$\Pr_{\hat{y}\in\{0,1\}^m}[f(\hat{a}+\hat{y}) \neq f(\hat{a}) + f(\hat{y})] \geq \frac{1}{2}.$$

Thus we have $\delta/2 \geq \sum_{\hat{a}: f(\hat{a}) \neq \hat{f} \neq \hat{f}(\hat{a})} \Pr_{\hat{x}}[\hat{x} = \hat{a}] \cdot (1/2) = (1/2) \cdot \Pr_{\hat{x}}[f(\hat{x}) \neq \hat{f}(\hat{x})]$, that is,

$$\Pr_{\hat{x}}[f(\hat{x}) \neq \hat{f}(\hat{x})] \leq \delta.$$

∎

We are now left with the task of showing that $\hat{f}$ is linear. Recall, that the value of $\hat{f}(\hat{x})$, according to our definition, is obtained by taking the most frequently occurring value of $f(\hat{x} + \hat{r}) - f(\hat{r})$, as $\hat{r}$ takes all possible values in $\{0,1\}^m$. We next show that our hypothesis on $\hat{f}$ implies that $\hat{f}(\hat{x})$ agrees with $f(\hat{x} + \hat{r}) - f(\hat{r})$ for more than 2/3 of the values of $\hat{r}$. (That it agrees $f(\hat{x} + \hat{r}) - f(\hat{r})$ for at least half of all $\hat{r}$ is obvious since $f(\hat{x} + \hat{r}) - f(\hat{r}) \in \{0,1\}$.)

Claim 11.7 $\forall \hat{x} \in \{0,1\}^m \; \Pr_{\hat{r} \in \{0,1\}^m}[\hat{f}(\hat{x}) = f(\hat{x} + \hat{r}) - f(\hat{r})] > \frac{2}{3}$.

Proof. Fix $\hat{x} \in \{0,1\}^m$ and let $p = \Pr_{\hat{r}}[\hat{f}(\hat{x}) = f(\hat{x} + \hat{r}) - f(\hat{r})]$. (As discussed above $p \geq 1/2$.) We wish to show that $p > 2/3$. By the assumption on f,

$$\Pr_{\hat{a}, \hat{b} \in \{0,1\}^m}[f(\hat{a}) + f(\hat{b}) \neq f(\hat{a} + \hat{b})] \;\leq\; \frac{\delta}{2} \;\leq\; \frac{1}{6}. \tag{11.2}$$

If $\hat{a}$ and $\hat{b}$ are chosen randomly (uniformly and independently) from $\{0,1\}^m$ then $\hat{x} + \hat{a}$ and $\hat{x} + \hat{b}$ also take values in $\{0,1\}^m$ with uniform distribution and independently. Hence

$$\Pr_{\hat{a}, \hat{b}}[f(\hat{x} + \hat{a}) + f(\hat{x} + \hat{b}) \neq f(\hat{a} + \hat{b})] \leq \frac{1}{6}. \tag{11.3}$$

(Note $(\hat{x} + \hat{a}) + (\hat{x} + \hat{b}) = (\hat{a} + \hat{b}) + (\hat{x} + \hat{x}) = \hat{a} + \hat{b}$.) From (11.2) and (11.3), we have

$$\Pr_{\hat{a}, \hat{b}}[f(\hat{x} + \hat{a}) - f(\hat{a}) \neq f(\hat{x} + \hat{b}) - f(\hat{b})] \;\leq\; \frac{1}{6} + \frac{1}{6} \;=\; \frac{1}{3}.$$

Now the left hand side is precisely $2p(1-p)$. Hence $2p(1-p) \leq 1/3$. With $p \geq 1/2$, this implies that $p > 2/3$. ∎

We are now ready to complete the proof of our main lemma. To show that $\hat{f}$ is linear, we shall use the following fact.

> If a function $f : \{0,1\}^m \to \{0,1\}$ satisfies $\forall \hat{x}, \hat{y} \in \{0,1\}^m \; f(\hat{x}) + f(\hat{y}) = f(\hat{x} + \hat{y})$, then f is a linear function.

Claim 11.8 *$\hat{f}$ is linear.*

Proof. In light of the fact stated above, we need only show that

$$\forall \hat{x}, \hat{y} \in \{0,1\}^m \; \hat{f}(\hat{x}) + \hat{f}(\hat{y}) = \hat{f}(\hat{x} + \hat{y}).$$

Fix $\hat{x}, \hat{y} \in \{0,1\}^m$. Then from Claim 11.7, we have

$$\begin{aligned}
\Pr_{\hat{r} \in \{0,1\}^m}[\hat{f}(\hat{x}) = f(\hat{x} + \hat{r}) - f(\hat{r})] &> 2/3 \\
\Pr_{\hat{r} \in \{0,1\}^m}[\hat{f}(\hat{x} + \hat{y}) = f(\hat{x} + \hat{y} + \hat{r}) - f(\hat{r})] &> 2/3,
\end{aligned}$$

and, since $\hat{x} + \hat{r}$ is also a random vector in $\{0,1\}^m$,

$$\Pr_{\hat{r} \in \{0,1\}^m}[\hat{f}(y) = f(\hat{x} + \hat{y} + \hat{r}) - f(\hat{x} + \hat{r})] > 2/3.$$

Thus, with non-zero probability, all three events hold. But then $\hat{f}(\hat{x} + \hat{y}) = \hat{f}(\hat{x}) + \hat{f}(\hat{y})$. That is,

$$\Pr_{\hat{r} \in \{0,1\}^m}[\hat{f}(\hat{x} + \hat{y}) = \hat{f}(\hat{x}) + \hat{f}(\hat{y})] > 0.$$

Since this event does not depend on the vector $\hat{r}$, it must hold with probability 1. ∎

Remarks

The result $\mathcal{NP} \subseteq \mathrm{PCP}(\mathrm{poly}, 1)$ is due to [ALM$^+$92]. The proof presented in this lecture is taken from Sudan's thesis [Sud92]. In this proof a crucial role was played by Fact 11.3, which was first observed by Freivalds [Fre79].

The result proved in the second half, Lemma 11.5, is based on Blum, Luby and Rubinfeld [BLR90]; their result was sharpened by Rubinfeld [Rub90] and Gemell, Lipton, Rubinfeld, Sudan and Wigderson [GLR$^+$91]. The proof presented in the lecture is based on the proof in [BCD$^+$93].

References

[AB87] N. Alon and R. B. Boppana. The monotone circuit complexity of bolean functions. *Combinatorica*, 7(1):1–22, 1987.

[Adl78] L. Adleman. Two theorems on random polynomail time. In *Proceedings of the 19th IEEE Symp. on the Foundations of Computer Science (FOCS)*, pages 75–83, 1978.

[ALM⁺92] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *Proceedings of the 33rd IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 14–23, 1992.

[AS92] S. Arora and S. Safra. Probabilistic checking of proofs; a new characterizations of $\mathcal{NP}$. In *Proceedings of the 33rd IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 2–13, 1992.

[Bab85] László Babai. Trading group theory for randomness. In *Proceeding of the 17th ACM Symposium on the Theory of Coumputing (STOC)*, pages 421–429, 1985.

[BCD⁺93] Richard Beigel, Chih-Ping Chen, Jack Donham, Will Hurwood, Andrzej Krauze, Krikis Martinch, Daniel Milstein, Sophia Paleologou, Pharr Matt, David Rochberg, and Kentaro Toyama. Class notes on interactive proof systems. Technical Report YALEU/DCS/TR-947, Yale University, Department of Computer Science, January 1993.

[BDG87a] José Luis Balcázar, Josep Díaz, and Joaquim Gabbarró. *Structural Complexity I*. Springer-Verlag, 1987.

[BDG87b] José Luis Balcázar, Josep Díaz, and Joaquim Gabbarró. *Structural Complexity II*. Springer-Verlag, 1987.

[BFL90] L. Babai, L. Fortnow, and C. Lund. Non-deterministic exponential time has two-prover interactive prtocols. In *Proceedings of the 31st IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 16–25, 1990.

[BFL91] L. Babai, L. Fortnow, and C. Lund. Non-deterministic exponential time has two-prover interactive prtocols. *Computational Complexity*, 1:3–40, 1991.

[BFLS91] L. Babai, L. Fortnow, L. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *Proceedings of the 23rd ACM Symposium on the Theory of Coumputing (STOC)*, pages 21–31, 1991.

[BHZ87] R. Boppana, J. Hastad, and S. Zachos. Does co-$\mathcal{NP}$ have short interactive proofs. *Information Processing Letters*, 25:127–132, 1987.

[BJ81] C. G. Bennett and Gill J. Relative to a random oracle A, $\mathcal{P}^A \neq \mathcal{NP}^A \neq$ co-$\mathcal{NP}^A$ with probability 1. *SIAM Journal of Computing*, 10:96–113, 1981.

[BLR90] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numberical problems. In *Proceedings of the 22nd ACM Symposium on Theory of Computing (STOC)*, pages 73–83, 1990.

[BM88] László Babai and Shlomo Moran. Arthur-merlin games: a randomized proof system, and a hierarchy of complexity classes. *Journal of Computer and System Sciences*, 36:254–276, 1988.

[BOGKW88] M. Ben-Or, S. Goldwasser, J. Kilian, and A. Wigderson. Multi-prover interactive proofs: How to remove intractability assumptions. In *Proceedings of the 20th ACM Symposium on the Theory of Coumputing (STOC)*, pages 113–131, 1988.

[Boo89] Ronald V. Book. Restricted relativizations of complexity classes. In Juris Harmanis, editor, *Computational Complexity Theory*, volume 38 of *AMS Short Course Lecture Notes (Proceedings of Symposia in Applied Mathematics)*, chapter 3, pages 47–74. American Mathematical Society, 1989.

[CKS81] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[CW79] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[FGL+91] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Approximating clique is almost $\mathcal{NP}$-complete. In *Proceedings of the 32nd IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 1–12, 1991.

[FL92] U. Feige and L. Lovász. Two-prover one round proof systems: Their power and their problems. In *Proceedings of the 24th ACM Symposium on the Theory of Computing*, pages 733–744, 1992.

[Fre79] R. Freivalds. Fast probabilistic algorithms. In *Proceedings of Mathematical Foundations of Coumputer Science*, volume 74 of *Lecture Notes in Computer Science*, pages 57–69. Springer-Verlag, 1979.

[FRS88] L. Fortnow, J. Rompel, and M. Sipser. On the power of multi-prover interactive protocols. In *Proceedings of the third IEEE conference on Structure in Complexity Theory (Structures)*, pages 156–161, 1988.

[FS89] L. Fortnow and M. Sipser. Are there interactive protocols for co-$\mathcal{NP}$ langauges? *Information Processing Letters*, 28:148–156, 1989.

[GLR+91] P. Gemmell, R. Lipton, R. Rubinfeld, M. Sudan, and A. Wigderson. Self-testing/correcting for polynomials and for approximate functions. In *Proceedings of the 22nd ACM Symposium on Theory of Computing (STOC)*, pages 32–42, 1991.

[GM89] S. Goldwasser and Sipser M. Private coins versus public coins in interactive proof systems. In Silvio Micali, editor, *Randomness and Computation*, volume 5 of *Advances in Computing Research*, chapter 4, pages 73–90. JAI Press, 1989.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18:186–208, 1989.

[GMS87] O. Goldreich, Y. Mansour, and M. Sipser. Interactive proof systems: Provers that never fail and random selection. In *Proceedings of the 28th IEEE Symp. on the Foundations of Computer Science (FOCS)*, pages 449–461, 1987.

[GMW86] Shafi Goldwasser, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity and a methodology of cryptographic protocol design. In *Proceedings of the 27th IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 174–187, 1986.

[Gol89] Shafi Goldwasser. Interactive proof systems. In Juris Harmanis, editor, *Computational Complexity Theory*, volume 38 of *AMS Short Course Lecture Notes (Proceedings of Symposia in Applied Mathematics*, chapter 6, pages 108–128. American Mathematical Society, 1989.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979. Reprinted by Narosa Publishing House, 1992.

[KVV87] Mulmuley K., Vazirani U. V., and Vazirani V. V. Matching is as easy as matrix inversion. *Combinatorica*, 7:105–113, 1987.

[Lau83] C. Lauteman. $\mathcal{BPP}$ and the polynomial hierarchy. *Information Processing Letters*, 17(4):215–217, 1983.

[LFKN90] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for intereactive proof systems. In *Proceedings of the IEEE Symp. on the Foundations of Computer Science (FOCS)*, pages 2–10, 1990.

[LFKN92] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for intereactive proof systems. *Journal of the ACM*, 39(4):859–868, October 1992.

[LL90] A. K. Lenstra and H.W. Lenstra Jr. Algorithms in number theory. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science: Algorithms and Complexity*, volume A, chapter 12, pages 673–715. MIT Press, 1990.

[LP81] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1981.

[Lun91] Carsten Lund. The power of interaction. Technical Report 91-01, Department of Coumputer Science, University of Chicago, January 1991. Ph.D. Thesis.

[Pap94] Christos H. Papdimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[Pra75] V. Pratt. Every prime has a succinct certificate. *SIAM Journal of Computing*, 4:214–220, 1975.

[Raz85a] A. A. Razborov. Lower bounds on the monotone complexity of some boolean functions. *Soviet Math. Dokl.*, 31:354–357, 1985.

[Raz85b] A. A. Razborov. Lower bounds on the monotone network complexity of the logical permanent. *Mathematical Notes*, pages 485–493, 1985.

[Rub90] R. Rubinfeld. *A Mathematical Theory of Self-Checking, Self-Testing and Self-Correcting Programs.* PhD thesis, Computer Science Department, U. C. Berkeley, 1990.

[Sha90] A. Shami. IP = PSPACE. In *Proceedings of the 31st IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 11–15, 1990.

[Sha92] A. Shamir. IP = PSPACE. *Journal of the ACM*, 39(4):869–877, October 1992.

[She92] A. Shen. IP = PSPACE: Simplified proof. *Journal of the ACM*, 39(4):878–880, October 1992.

[Sip83] M. Sipser. A complexity theoretic approach to randomness. In *Proceedings of the 15th ACM Symposium on the Theory of Coumputing (STOC)*, pages 330–335, 1983.

[Sip92] Michael Sipser. The history and status of the $\mathcal{P}$ versus $\mathcal{NP}$ question. In *Proceedings of the 24th ACM Symposium on the Theory of Computating*, pages 603–618, 1992.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.

[SM73] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *Proceedings of the 5th ACM Symposium on the Theory of Coumputing (STOC)*, pages 1–9, 1973.

[Sud92] Madhu Sudan. *Efficient Checking of Polynomials and Proofs and the Hardness of Approximation Problems.* PhD thesis, Computer Science Department, U. C. Berkeley, 1992.

[Tar88] É Tardos. The gap between monotone and nonmonotone circuit complexity is exponential. *Combinatorica*, 8(1):141–142, 1988.

[Tod89] S. Toda. On the computational power of $\mathcal{PP}$ and $\oplus\mathcal{P}$. In *Proceedings of the 30th IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 514–519, 1989.

[Val79] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.