Computational Complexity

CSCI-GA 3350

Subhash Khot
Transcribed by Patrick Lin

Abstract. These notes are from a course in Computational Complexity, as offered in Spring 2014 at the Courant Institute of Mathematical Sciences, a school of New York University. The professor for the course was Subhash Khot. Two guest lectures on Proof Complexity were given by Pratik Worah.

Information about the course can be found at the professor's web page, `http://cs.nyu.edu/~khot`.

The first part of the course includes bread-and-butter complexity. Definitions are given for P and NP, then space complexity, the Polynomial Hierarchy, $\#\mathsf{P}$, Interactive Proofs; relations between them will be discussed. The rest of the course consists of some selected topics, including Proof Complexity, Circuit Lower Bounds, Quantum Computation, Communication Complexity, and Probabilistically Checkable Proofs.

No textbook is followed, but a good reference is the book by Arora and Barak. Other good references are the one by Papadimitriou and the one by Du and Ko. There are good lecture notes online of similar courses, such as the one taught by Trevisan.

General prerequisite knowledge can be found in the books by Sipser and Cormen, Leiserson, Rivest, Stein.

These notes were transcribed by Patrick Lin, and attempt to follow the lectures as faithfully as was possible, but there are possibly errors and inconsistencies.

Revision 1 Aug 2014 03:33.

Contents

CHAPTER 1

Introduction

We will start with a gentle introduction to theoretical computer science. Lecture 1, 27 Jan

1.1. Complexity and Algorithms

One of the subdisciplines under computer science is theoretical computer science. There is a subdivision that is kind of artificial, but we can divide it into research on algorithms design and research on complexity.

For example, under algorithms we know that we can do sorting in $n \log n$ steps (we will not care about constants, which will be hidden by Ω and O notation).

On the other hand, under complexity the main point is to first precisely define computation, using the Turing Machine as the main model of computation, and then study the limitations of computational models.

So whereas from Algorithms we can say that we can do sorting in $n \log n$ time, in complexity we have the following:

Theorem 1.1. *Any comparison-based sorting algorithm takes $\Omega(n \log n)$ steps.*

So we can sort of think of this as positive results (algorithms) and negative results (complexity).

Another nice theorem is the following:

Theorem 1.2. *There is no "fast"[1] algorithm for the Traveling Salesman Problem unless $\mathsf{P} = \mathsf{NP}$[2].*

So it is clear why the positive results are useful. So why are negative results useful?

One is optimality: we know that nobody can do better.

Another is when we desire hard or impossible problems, such as when breaking cryptographic systems.

One of the first questions that will pop out is the question of determinism and non-determinism. We will see what these mean, but this amounts to asking "what is more difficult, finding a proof for a mathematical theorem, or given a proof showing that it is correct?" The statement that finding a proof is more difficult seems self-evident, but we cannot show this fact.

We can also ask from an algorithmic standpoint if we will have a better result if we use randomness.

We can also compare worst-case vs average-case, eg. the Traveling Salesman Problem has cases that are hard, but maybe on average they are easy and that is good enough for whatever case we care about.

[1]We will eventually rigorously define fast to mean in polynomial time.

[2]We have not defined these yet, so for now we can think of this as "unless the world collapses".

Another case is approximations, where we say that we can find tools that within a certain amount of the optimal.

Another is the power of interaction, which ties in very well with cryptographic applications. One way to think about this is that we can interact with the devil, so we can ask questions but the devil might also cheat. So using such interactions with the devil, can we in fact solve problems that we could not solve on our own?

So we will now start formalizing some of the terms we have been using, such as what "algorithm" means or even what a "computational problem" means.

1.2. Definition of Computation and Algorithm

As we intuitively understand, "problems" come in two types. The first is what is known as a "decision" problem where the answer is a simple "yes" or "no", eg. "Is n prime?" or "Is a given graph G 3-colorable?". The other type is a search problem, eg. "Factorize n" or "Find a 3-coloring of G if one exists".

For the purposes of complexity, it is very nice to deal only with decision problems. In a general sense, if we can deal with the decision problems, we have captured the scope of computational problems, since we can reformulate search problems as decision problems.

For example, take the search problem "Factorize n". But we can always come up with a decision problem of the following type:

Suppose we are given three numbers n, a, b, $1 < a < b < n$, does n have a factor in the range a to b? Then in this way we can find the factorization of n using a binary search just using this yes/no problem.

So it is generally the case that if we can solve the decision problem we can solve the search problem.

Decision problems can be cleanly defined as what are called Language Membership problems.

Definition 1.3. An *alphabet* Σ is a finite set of symbols. ◇

Some examples include the binary alphabet $\Sigma = \{0, 1\}$, the English alphabet $\Sigma = \{a, b, \ldots, z\}$, the set of symbols on the keyboard, $\Sigma = \{\texttt{a-z}, \texttt{A-Z}, \texttt{0-9}, \texttt{!}, \texttt{@}, \ldots\}$.

Definition 1.4. Σ^* is the set of all finite length strings over Σ. ◇

For example, $\{0, 1\}^* = \{\varepsilon^{[3]}, 0, 1, 00, 01, 10, 11, \ldots\}$.

Definition 1.5. A *language* L is an arbitrary subset of Σ^*. ◇

For example, on the language $\Sigma = \{0, 1\}$ we can set L to be the set of all binary strings that represent a prime integer. Note that this language is infinite. Another example on $\{a, \ldots, z\}$, the language L to be the set of all words in an English dictionary. This language is finite. Take the alphabet to be the set of keyboard symbols, and the language to be the set of all syntactically correct `C` programs.

Definition 1.6. Given a language L, the corresponding *Language Membership problem* is the computational problem P_L: "Given $x \in \Sigma^*$, is $x \in L$?" ◇

For the three examples we defined, the problem is obvious.

The punchline is the following:

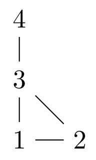
[3]The symbol ε denotes the empty string.

FACT 1.7. *"All" decision problems can be cast as language membership problems for an appropriate language.*

This may require a leap of faith or some experience in these things. Here is another example to help show this.

FACT 1.8. *Every combinatorial object can be encoded as a string.*

EXAMPLE 1.9. Given a graph G we can write down the adjacency matrix of the graph, and then write the adjacency matrix as a string by writing rows one after the other (see Figure 1.10). $\diamond$



$$\longrightarrow \quad \begin{array}{c|c|c|c|c|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ \hline 2 & 1 & 0 & 1 & 0 \\ \hline 3 & 1 & 1 & 0 & 1 \\ \hline 4 & 0 & 0 & 1 & 0 \\ \hline \end{array} \quad \longrightarrow \quad \texttt{0110\#1010\#1101\#0010}$$

FIGURE 1.10. An example encoding of a graph G

So we can encode whatever we want as a string. In particular, we can encode computational models themselves as strings. We write the encoding of an object as surrounded by angle brackets $\langle\ \rangle$.

So now we can define the language $L_{\text{3-color}} = \{\langle G \rangle\} \mid G$ is 3-colorable$\}$. There is a caveat that some strings are not valid encodings, but we can just test for that first.

So we now understand what is a computational problem is: we have an underlying language L and the problem is simply a membership problem for this language.

DEFINITION 1.11. A *computational model* is a black box that solves the problem $P_L =$ "Given x, is $x \in L$?".

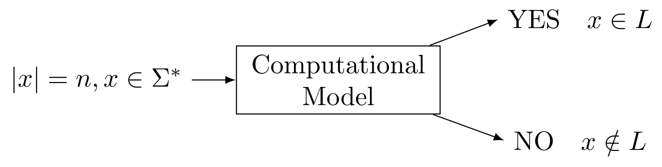


FIGURE 1.12. Diagram of a Computational Model

The model solves P_L correctly if $\forall x \in \Sigma^*$, the decision is correct. $\diamond$

So depending on the model, we may have some restrictions. For example, in finite automata, we have a string but we can only read the input left to right without any repeats. We also have restrictions on the kind of memory the model is allowed access to. For example in pushdown automata the memory takes the form of a stack.

We also have a concept of a step of computation, then the *running time* is simply the number of steps it takes to perform the computation.

The *time complexity* of the machine is $t(n)$ if on every input of length n, the machine stops in time at most $t(n)$. This is a worst-case running time.

1.2.1. The Turing Machine. A *Turing Machine*, defined in the 1930s by Turing before an actual computer was invented, is a model that works as follows:

There is a one-way infinite tape divided into cells, with one symbol in each cell. If the input has length n, then the first n cells contain the input. By convention, the rest of the tape is filled with some blank symbol, say B. The machine has a head/pointer that starts out pointing to the first cell on the tape. See Figure 1.13.

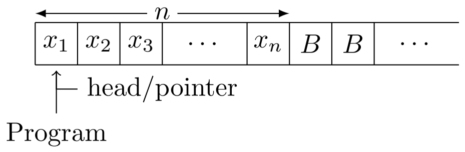


FIGURE 1.13. Diagram of a Turing Machine

The program is a tuple $(Q, \Sigma, \Gamma, q_s, q_a, q_r, \delta)$ where Q is the finite number of states[4]; Σ is the input alphabet; $\Gamma \supseteq \Sigma$ is the input alphabet with extra symbols that are useful; three states $q_s, q_a, q_r \in Q$ are the starting state, accepting state, and rejecting state, respectively; δ is the transition function for the machine, formally a function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$, eg. if the machine is at state q_{12} and the current cell says "a", switch to state q_{14}, write "b" in the cell, and move one cell to the right.

The execution of the machine is as follows: On input x, starting in the initial configuration with the head at the first cell and state at q_s, it can look at the symbol, write a new one if necessary, change states, and move either to the left or to the right. Let the machine run. Then at some point if the machine enters the accept state q_a or the reject state q_r, stop the machine and output the decision: accept, or reject respectively.

Once a Turing Machine M is fixed, on input x there are three possibilities: stops and accepts, stops and rejects, or it runs forever.

For example, one could create a very stupid machine that only moves to the right, so this would run forever.

1.2.2. Algorithms and Decidability. An *algorithm* is a Turing Machine (or program) that always (on every $x \in \Sigma^*$) halts.

DEFINITION 1.14. A language L is *decidable* if it is decided by a Turing Machine that always halts. That is, for $x \in L$ the machine accepts x, and for $x \notin L$ the machine rejects x. $\diamond$

In other words, a language is decidable if there is an algorithm for it.

So we may ask for an algorithm how fast it does it run. But forgetting that, we can ask a more basic question: for every problem, if there is an algorithm for it? An unfortunate fact of life is that this is false.

Here are two problems showing that this is false.

The first is the problem $A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ halts and accepts on } w\}$ where $w \in \Sigma^*$ and M is a Turing Machine. This language is undecidable and thus has no algorithm for it.

[4]This is an important point, that with a finite number of states the machien can decide an infinite number of possibilities.

The second is the language $L = \{\langle P\rangle \mid P \text{ has an integral root}\}$ where P is a multivariate polynomial with integer coefficients. For example, the question does $x - y^2 = 1$ have an integral root? The answer in this case is yes, since we can set $x = 1, y = 0$. But for $x^2 = 17$, the answer is no. This problem was posed by Hilbert in the early 1900s, before the idea of Turing Machines. A striking fact is that there is no Turing Machine that can solve this problem.

We will quickly prove that there exists an undecidable language, using a technique called diagonalization that is also useful in other parts of complexity. This technique was proposed by Cantor to show that the set of reals $[0, 1]$ is uncountable.

Proof. Suppose that the set is countable, that there is a list of reals $r_1, r_2, \ldots$ where $r_i \in [0, 1]$ and every real in $[0, 1]$ appears on this list as some r_j. Each number has a binary representation as an infinite string of 0s and 1s.

$$\begin{array}{c|l}
r_1 & .\underline{0}01101110\ldots \\
r_2 & .1\underline{1}0010110\ldots \\
r_3 & .00\underline{1}100011\ldots \\
\vdots & \vdots \quad \ddots \quad \ddots
\end{array} \qquad r = .100\ldots$$

Figure 1.15. Example of Diagonalization

Look at this as an infinite matrix, then look at its diagonal, flip it, and call it r. By this construction for every j, $r \neq r_j$ since r must differ from r_j on the j-th digit. But this is a contradiction since then r does not appear on this list. □

So the point is to construct some number r that disagrees with every number on the list.

We will use this and the following fact to prove that there is an undecidable problem:

Fact 1.16. *The set of all Turing Machines is countable. That is, Turing Machines have an effective ordering: there is a list* $\langle M_1\rangle, \langle M_2\rangle, \langle M_3\rangle, \ldots$ *such that for every Turing Machine* M *its description appears on this list somewhere.*

We can see this because an encoding of a Turing Machine is just a string and we can have a lexicographic ordering of strings.

Lecture 2, 29 Jan

Proof of Existence of an Undecidable Language. We will show that A_{TM} is undecidable by doing diagonalization on $\langle M_1\rangle$, $\langle M_2\rangle$, $\langle M_3\rangle$, $\ldots$ and on inputs $x_1, x_2, x_3, \ldots$

	x_1	x_2	x_3	$\ldots$
$\langle M_1\rangle$	$\underline{\text{A}}$	R	NH	$\cdots$
$\langle M_2\rangle$	A	$\underline{\text{R}}$	A	$\cdots$
$\langle M_3\rangle$	·	·	$\underline{\text{NH}}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\ddots$

Figure 1.17. Example of Diagonalization on Turing Machines

Then we can construct a language from the results on the diagonal. Then we can create a Turing machine with the following description:

```
$D$ := "On input $x$,
    Find index $i$ such that $x = x_i$.
    Then find $M_i$.
    Assume that $A_{\mathsf{TM}}$ is decidable, then decide if $M_i$ accepts $x_i$.
    If so, reject; else, accept."
```

Then D disagrees with M_i on every input x_i. This is a contradiction, so A_{TM} is undecidable. □

CHAPTER 2

Time-Bounded Computation

In this course we will not concern ourselves with decidability, so henceforth we will only deal with decidable languages, and the main issue will be running time, which as previously stated, is the number of steps the computational model takes. By convention, we can always set an a priori time bound $t(n)$ such that if the time exceeds this, we stop and reject.

It is important to note that the computational model can matter. For example, on a 1-tape Turing Machine the problem of checking if a string is a palindrome can be done in $O(n^2)$ time, but it is possible that a 2-tape Turing Machine can do it in $O(n)$ time[1]. But in another sense the model does not matter, as the following theorem shows:

THEOREM 2.1. *Any k-tape Turing Machine that runs in time $t(n)$ can be simulated by a 2-tape Turing Machine in time $O(t(n) \log t(n))$.*

2.1. The Time Hierarchy Theorem

DEFINITION 2.2. $\mathsf{DTIME}(t(n))$ = class of all languages decided by a Turing Machine in time $O(t(n))$. $\diamond$

Note that the notation hides constants and constant factors, but this is okay because given any Turing Machine we can construct a new machine that runs some arbitrary number of times faster. This is kind of cheating, but it goes like this:

On the new machine, we have a new language such that every ℓ cells on the tape in the original becomes one cell in the new tape, so that the new Turing Machine now operates on these giant blocks, and thus can run ℓ times faster. This is a basic result called the Speed-Up Theorem.

But even then the definition is problematic, because we have not specified what model the language is decided by, especially since by Theorem 2.3, if we move between models we may gain or lose the factor of $\log t(n)$. By convention, we say that the language is be decided by a multi-tape Turing Machine.

Now one may think that if we allow more time, we can solve more problems. In principle this makes sense, and is spelled out by the following theorem:

THEOREM 2.3 (Time Hierarchy Theorem). *Say $t_2(n) \gg t_1(n) \log t_1(n)$[2], then $\mathsf{DTIME}(t_1(n)) \subsetneq \mathsf{DTIME}(t_2(n))$.*

Results of this type are called separation results, and they are very rare. Most of them are proved using diagonalization methods.

[1]In time we will prove that this problem takes $\Omega(n^2)$ time on any model.

[2]That is $\lim_{n\to\infty} \frac{t_2(n)}{t_1(n)\log t_1(n)} = \infty$.

The idea is to have an effective ordering of machines that each run in time $t_1(n)$, and then construct a machine that disagrees with all of them, but that runs in time $t_2(n)$. There are, however, some subtle points in the proof.

Well, $\mathsf{DTIME}(t_1(n)) \subseteq \mathsf{DTIME}^2(t_1(n)\log t_1(n))$. We will proceed to show that $\mathsf{DTIME}^2(t_1(n)\log t_1(n)) \subsetneq \mathsf{DTIME}(t_2(n))$.

FACT 2.4. *The set of all 2-tape Turing Machines that run and finish in time $O(t_1(n)\log t_1(n))$ has effective ordering*[3].

PROOF OF THEOREM 2.3. Create a table as before of Turing Machines and strings in the language. Now there is a technical point that finding the right machine given a language might create a lot of overhead, but we can get out of this by attaching $\#^k$ to the end of every string, so that the overhead becomes bounded by some constant as a function of k.

Then define a new machine as follows:

```
$D :=$ "On input $x\#^k$,
    Find $i$ such that $x = x_i$.
    Then find $M_i$.
    Run $M_i$ on $x_i\#^k$.
    If the time exceeds $t_2(|x| + k)$ stop and reject.
    Else, if $M_i$ accepts, reject; if $M_i$ rejects, accept."
```

Clearly by design D runs in time $t_2(n)$, and for any i, D disagrees with M_i on input $x_i\#^k$ provided the simulation of M_i was carried out to completion. The simulation will always be carried out to completion for sufficiently large k: we see that $t_2(|x_i| + k) \gg O(1) + O(t_1(|x_i| + k)\log t_1(|x_i| + k))$. □

2.2. The class P

We now define the class of languages that are decidable in polynomial time.

DEFINITION 2.5. $\mathsf{P} = \bigcup_{c\in\mathbb{N}} \mathsf{DTIME}(n^c)$ ◇

This is the class of problems that are considered "fast". Now there are objections, such as if $c = 50$ then how can we call this fast? Well there is really no good answer, but in a decent one is that in practice if someone discovers an algorithm that solves the problem in time n^{50}, soon someone will probably find an algorithm that solves the problem in time n^2. Furthermore, in practice, all realistic computational models have running time that are off from each other by a factor of at most polynomial. The end result is that we get some beautiful theory. At a lower level, this definition also helps build a beautiful theory since the set of languages solvable in polynomial time is closed under composition.

The Church-Turing thesis states that any realistic model can be simulated by a Turing Machine with at most polynomial loss in efficiency, though this is now being challenged by newer models, such as the quantum computer.

[3]This statement as stated is probably not exactly true, but something to that effect is true. The trick is to take the list of all machines and attach a "clock" to every single one, such that if the running time of the machine exceeds $t_1(n)\log t_1(n)$, the "clock" forces the machine to stop and reject, and if the machine already finishes before this time bound it is unchanged. Then this gives an effective ordering of machines that finish in time $t_1(n)\log t_1(n)$.

2.3. Non-determinism and the class NP

A *non-deterministic Turing Machine* (NTM) is as follows: on a Turing Machine, we relax the instructions so that given a state and a symbol, we can have a choice between any number of possible moves. More specifically, we move from the transition function moving to a new state and position $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ to a new transition function moving to a subset of all possible new states and positions $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$. If the result is the empty set, we can just say that the machine has no possible moves and so halts and rejects. It is reasonable to restrict the output to only a pair of possible states and positions, which we can do without loss of generality. We will refer to a Turing Machine that operates deterministically as a *deterministic Turing Machine* (DTM).

So what is meant actually by a non-deterministic Turing Machine solving a problem? The idea is that at every move, we end up choosing the "right move" to make. So this is a very unrealistic model, existing only in theory, but we will see why it is important to study.

Formally, we define *acceptance by a non-deterministic Turing Machine* as follows: say N is a non-deterministic Turing Machine and x is an input. Then on any input we have two possible choices, so that if we draw out the possibilities we get a binary tree of possible moves (see Figure 2.6). Then at time $t(n)$ there are $2^{t(n)}$ leaves.

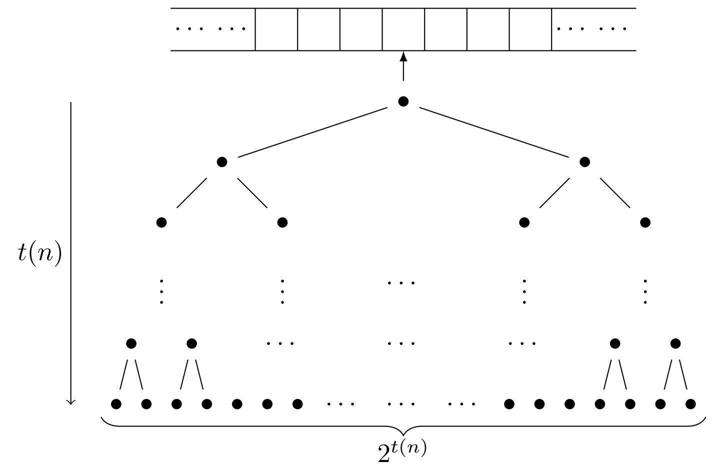


FIGURE 2.6. Binary tree of possible computational paths

Well, we need the computational model to output "Yes" or "No", so we can pick a threshold, say we accept if at least one of the leaves accepts, or if at least half the leaves accepts[4], or any other threshold. For the purposes of the non-deterministic Turing Machine, we say the model accepts if at least one of the leaves accepts:

DEFINITION 2.7. A non-deterministic Turing Machine is said to accept x if there is at least one computational path that accepts. ◇

[4]This leads to a class called PP.

So the non-deterministic Turing Machine N accepts the language L if N has at least one accepting computational path if $x \in L$, and N rejects on every computational path if $x \notin L$.

We have the following definition analogous to that of DTIME:

Definition 2.8. $\mathsf{NTIME}(t(n))$ = class of all languages decided by a non-deterministic Turing Machine in time $O(t(n))$. ⋄

We also have the analogous Time Hierarchy Theorem:

Theorem 2.9 (Time Hierarchy Theorem). *$\mathsf{NTIME}(t_1(n)) \subsetneq \mathsf{NTIME}(t_2(n))$ if $t_2(n) \gg t_1(n) \log t_1(n)$.*

Finally, we have the analogous class NP:

Definition 2.10. $\mathsf{NP} = \bigcup_{c\in\mathbb{N}} \mathsf{NTIME}(n^c)$. ⋄

Clearly $\mathsf{P} \subseteq \mathsf{NP}$, but the big question is does $\mathsf{P} = \mathsf{NP}$?

Lecture 3, 3 Feb

We will show some problems that are known to be in NP but are believed to not be in P:

- $3{-}\mathsf{COL} = \{G \mid G \text{ is 3-colorable}\}$
- $\mathsf{TSP} = \{(G, wt, \ell) \mid G \text{ has a tour of length} \leqslant \ell\}$
- $\mathsf{SAT} = \{\phi \mid \phi \text{ has a satisfying assignment}\}$

All of these problems are of the type "guess and verify". How do these problems fit into the model we have given for NP? Well, for example for $3 - \mathsf{COL}$, at each step we can pick a number from $\{0, 1, 2\}$ representing a color of a node in G, then we can write it down into a string. If this string represents a valid 3-coloring then we accept. Then we can go down nondeterministically and so this is valid.

So we can deal with this fuzzy "guess and verify" where the first phase is to nondeterministically guess, and the second is to deterministically verify.

2.4. NP-completeness

The three problems from the previous section are NP-*complete*; that is, they are the "hardest" problems in NP. If there is any algorithm that solves these algorithms in polynomial time, then all problems in NP can be solved in polynomial time.

We define the notion of reduction, that is, if we can solve one problem then we can solve another.

Definition 2.11. Let A, B be languages. We say that A *reduces to* B in polynomial time, written $A \leqslant_P B$, if there is a DTM M (with output), which we call the *reduction*, that runs in time poly($|x|$), where x is the input, such that for all $x \in \Sigma^*$, $x \in A \iff M(x) \in B$, where $M(x)$ is the output of M on input x. ⋄

Fact 2.12. *If $A \leqslant_P B$, $B \in \mathsf{P}$, then $A \in \mathsf{P}$.*

This follows easily from the definition. For the formal proof, use the facts that $|M(x)| \leqslant \text{poly}(|x|)$ and composition of polynomials is polynomial.

Fact 2.13. *If $A \leqslant_P B$, $B \leqslant_P C$, then $A \leqslant_P C$.*

Using these we can characterize our class of NP-complete problems.

Definition 2.14. B is called NP-*complete* if $B \in \mathsf{NP}$ and for each $A \in \mathsf{NP}$, $A \leqslant_P B$. ⋄

A priori, there is not clear that an NP-complete language should exist.

Let us look at another example of an NP-complete language:

Let $L = \{\langle M, x, 1^t\rangle \mid M$ is a NTM that has accepting comptuation on input x in $\leqslant t$ steps$\}$. Well, $L \in \mathsf{NP}$, because we can use a NTM to simulate M on x. Now take any language $A \in \mathsf{NP}$, then $A \leqslant_P L$. The reduction is kind of cheating, but it is perfectly legitimate. Well, A is accepted by a NTM M_A in time n^c. Map any string $x \mapsto \langle M_A, x, 1^{n^c}\rangle$ where $n = |x|$. Hence $x \in A \iff \langle M_A, x, 1^{n^c}\rangle \iff M_A$ has an accepting computation on x in time $\leqslant n^c$.

So languages that are NP-complete do exist, but this example is a very artificial language.

2.4.1. The Cook-Levin Theorem. The following theorem is one of the most prominent in Computer Science, and shows that very nice problems are NP-complete. With this, once we get our hands on some nice problems like finding a clique, independent set, etc. and we show that they are NP-complete, we can show that many other problems are NP-complete.

THEOREM 2.15 (Cook-Levin Theorem). CIRCUIT$-$SAT *is* NP*-complete.*

A *circuit* is defined as follows: Let $x_1, x_2, \ldots, x_n \in \{0,1\}$ be Boolean variables. Then we can pick any two of them and combine them using an AND or an OR gate. We can negate inputs using a NOT gate. Then we can add more gates whose inputs are any of the original inputs, or the outputs of any other gate. There is one final gate, called the output gate, and this is the output of the entire circuit. Call the circuit C, then for any assignment a of $x_1, x_2, \ldots, x_n$, then we can use the circuit to compute the output $C(a) \in \{0,1\}$.

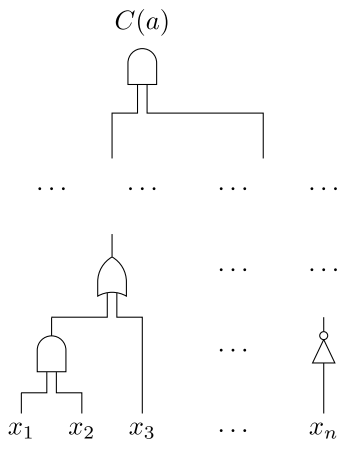


FIGURE 2.16. Example of a Circuit

Then CIRCUIT$-$SAT $= \{C \mid C$ is a circuit which has a satisfying assignment$\}$. That is, there exists some a such that $C(a) = 1$. The size of the circuit is the sum of the number of variables and the number of gates.

Now without loss of generality, we can assume that every algorithm for any problem in NP is of the type "guess and verify".

Observation 2.17. $L \in \mathsf{NP}$ if and only if there exists a polynomial time DTM M such that for $x \in \Sigma^*$, $x \in L \iff \exists y, |y| = |x|^c, M(x,y) = 1$. $\diamond$

Here y is the guess, and $M(x,y)$ is the verification. y here is often called a proof, a witness, or a certificate, for the claim that $x \in L$.

Before we prove the Cook-Levin Theorem, let us look at one final aside:

Fact 2.18 (Pratt's Theorem). *The language* $\mathsf{PRIME} = \{\langle m \rangle \mid m \text{ is prime}\}$ *is in* NP*, where* $|\langle m \rangle| = \log m$ *since* m *is written in a binary representation.*

Note that the naive algorithm of checking all factors is not polynomial time, since it runs in time exponential in the length of the input.

The proof that $\mathsf{PRIME} \in \mathsf{NP}$ is highly non-trivial.

We return to the Cook-Levin Theorem. We will use the following theorem:

Theorem 2.19. *Deterministic Turing Machines are equivalent to Circuits, where the running time of a DTM is equal to the size of the Circuit up to polynomial factors.*

Then after this theorem, the Cook-Levin Theorem follows almost naturally.

The statement of Theorem 2.19 says that for every DTM M, there exists a circuit C such that on every input x, $M(x) = C(x)$. The issue with making such a statement is that circuits only take strings of fixed length, whereas any Turing Machine can run on strings of any length. Other than this issue, which we will fix, the statement is true.

In particular, the theorem gives the following: let M be a polynomial time DTM. Then for any $n \geqslant 1$, one can construct in poly(n) time a circuit C_n with n inputs such that for every input $x \in \{0,1\}^*$[5] of length n, $C_n(x) = M(x)$. Furthermore, $|C_n| \leqslant \text{poly}(n)$.

Let us see how Theorem 2.19 implies the Cook-Levin Theorem.

Proof of Theorem 2.15. Let $L \in \mathsf{NP}$. Then $L \leqslant_P \mathsf{CIRCUIT{-}SAT}$, where the reduction sends $x \mapsto C_x$ where $x \in L \iff C_x$ has a satisfying assignment. Well, there is a polynomial time "verifier" M such that for all $x \in \{0,1\}^*$, then $x \in L \iff \exists y, |y| = \text{poly}(|x|), M(x,y) = 1$. Then by Theorem 2.19, we have $x \in L \iff \exists y, |y| = \text{poly}(|x|), C(x,y) = 1$, where C is a circuit equivalent to M on inputs of length $|x| + |y|$. This is also equivalent to saying that $C_x(y) = 1$, because we can think of x as fixed, that is, it is hardwired in the circuit, so the circuit really only depends on y. Finally, this is equivalent to saying that C_x has a satisfying assignment and thus $C_x \in \mathsf{CIRCUIT{-}SAT}$. In conclusion, $x \in L \iff C_x \in \mathsf{CIRCUIT{-}SAT}$. $\square$

We will now prove Theorem 2.19.

Proof of Theorem 2.19. Let M be a polynomial time DTM running in time n^k. Call the initial configuration C_0. (A technical point: since the machine must run in time n^k the length of the tape can be limited to n^k.) Now from configurations C_i to C_{i+1} any changes must be local, that is, any changes must happen inside a window of size 3. Then without loss of generality we may assume that at configuration C_{n^k}, the head is at the beginning of the tape and is in state q_a or q_r.

[5]Normally we use some arbitrary alphabet Σ but without loss of generality we may assume our alphabet is binary.

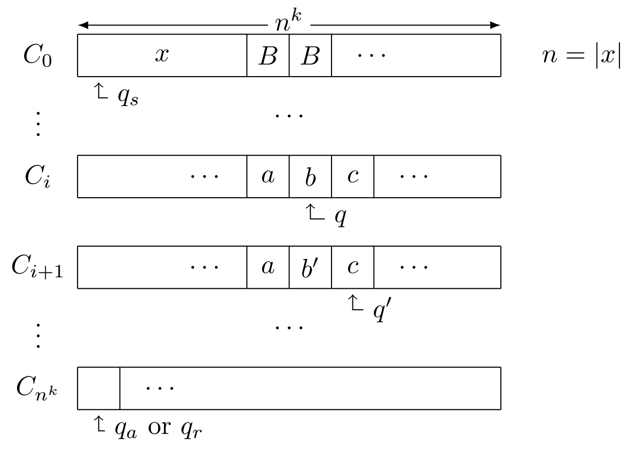


FIGURE 2.20. Example of the sequence of configurations

Now think of each configuration as a binary string, where each configuration is uniquely determined from the previous one. Now in fact, this is determined very locally, and there is a little circuit that can capture this local change. Then the final state of the machine can be captured using the final output gate. In this way we have a circuit that completely captures the Turing Machine (Figure 2.21).

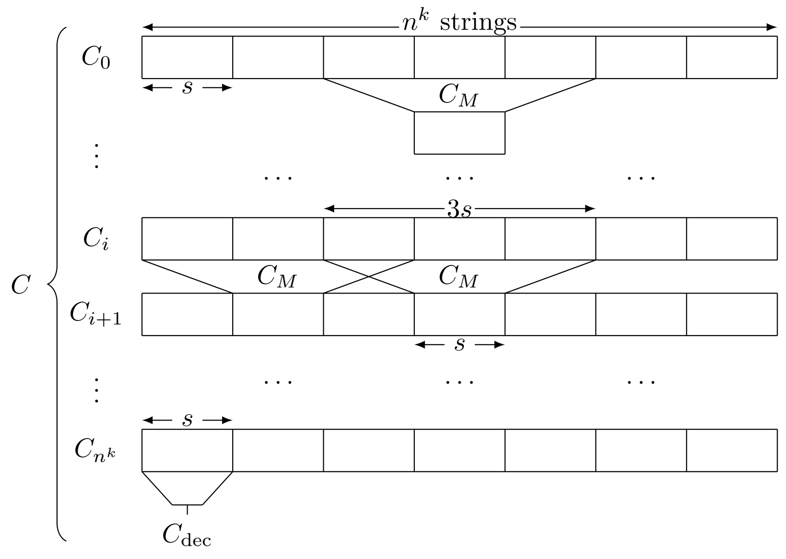


FIGURE 2.21. Example of the circuit created from the configurations

Formally, we write every configuration of the Turing Machine as a bit-string of length $n^k s$, where $s = \log_2(|\Gamma \cup \Gamma \times Q|)$, where each cell can be written using s bits. Then let $C_M : \{0,1\}^{3s} \to \{0,1\}^s$ be a function (and therefore a circuit) that implements one move of the Turing Machine M. We can combine all of these into a giant circuit that uniquely determines each row that follows the previous one. Then we have the final circuit $C_{\text{dec}} : \{0,1\}^s \to \{0,1\}$ that outputs the decision. Putting it all together we have a circuit C that agrees with M on all inputs of length n. $\square$

The Cook-Levin Theorem is usually given in terms of $\mathsf{3{-}SAT}$, so let us show that $\mathsf{CIRCUIT{-}SAT} \leqslant_P \mathsf{3{-}SAT}$.

A $\mathsf{3{-}SAT}$ formula is a set of variables $x_1, x_2, \ldots, x_n$ which is the AND of clauses where each clause is composed of at most 3 literals. Then $\mathsf{3{-}SAT}$ is the language of all such formulas with a satisfying assignment.

Given a circuit, for every gate we can write it as a constraint, eg. $y_1 = x_1 \wedge x_2$, $y_{11} = \neg y_{10}$, etc. At the end we have $y_{\text{out}} = 1$. Then these constraints can be written as a small number of clauses, so that we can write the entire circuit as a $\mathsf{3{-}SAT}$ formula, such that the circuit is in $\mathsf{CIRCUIT{-}SAT}$ if and only if the formula has a satisfying assignment.

For example, we will write $y_1 = x_1 \wedge x_2$ as a small number of clauses: well, we have $y_1 \implies x_1 \wedge x_2$ and $x_1 \wedge x_2 \implies y_1$. So we have $\neg y_1 \vee (x_1 \wedge x_2)$ which becomes $(\overline{y_1} \vee x_1) \wedge (\overline{y_1} \vee x_2)$ and $\neg(x_1 \wedge x_2) \vee y_1$ which becomes $\overline{x_1} \vee \overline{x_2} \vee y_1$. So we get a set of three clauses.

2.4.2. Some Properties of NP-Completeness. The following describes the standard way of showing a problem is NP-complete:

FACT 2.22. *Suppose B is NP-complete. If we have a problem $C \in \mathsf{NP}$ such that $B \leqslant_P C$, then C is NP-complete.*

The following terminology is important:

DEFINITION 2.23. B is called NP-hard if for all $A \in \mathsf{NP}$, $A \leqslant_P B$. $\diamond$

Note that B does not have to be in NP, and if it is, then B is also NP-complete.

The following is a consequence of the previous discussion:

FACT 2.24. $\mathsf{P} = \mathsf{NP}$ *if there is a polynomial time algorithm solving any NP-complete problem.*

Define $\mathsf{EXP} = \bigcup_{c \in \mathbb{N}} \mathsf{DTIME}(2^{n^c})$.

FACT 2.25. $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{EXP}$.

We already knew the first part, and the second part is because $\mathsf{NTIME}(t(n)) \subseteq \mathsf{DTIME}(2^{O(t(n))})$, which we can do by trying all of the computational paths by brute force.

Now by the Time Hierarchy Theorem, we know that $\mathsf{P} \subsetneq \mathsf{EXP}$, and it is conjectured that the other inclusions are also proper.

CHAPTER 3

Space-Bounded Computation

Now we will study the space or memory requirement of an algorithm. Lecture 4, 5 Feb

There are some nice theorems that connect the space requirement to the time requirement of a model.

We will start with the Turing Machine, as before. A naive way is to say that if the machine never reaches past a cell $t(n)$ on an input of length n, then that is the space requirement. But this kind of excessively punishes the algorithm. For example, even to read to the end of the input, the machine needs to reach n, so we can only work in models whose space requirement is at least n.

There are interesting things to do even in space that is less than n, provided one defines the space properly enough. One can even do things in space $\log n$.

For example when we are working with a very large file on a hard disk, we can still read small chunks of the file into memory and do things on those chunks; then the algorithm only works on the amount that is in memory, not the amount that is on the disk.

So we need to make a more refined definition when dealing with space-bounded computation.

Definition 3.1. A Turing Machine runs in space $S(n) \geqslant \log n$ if the input tape is read-only, and there are one or more "work tapes" of length $\leqslant S(n)$. Then $S(n)$ is the *space complexity* or *space requirement* of the machine. $\diamond$

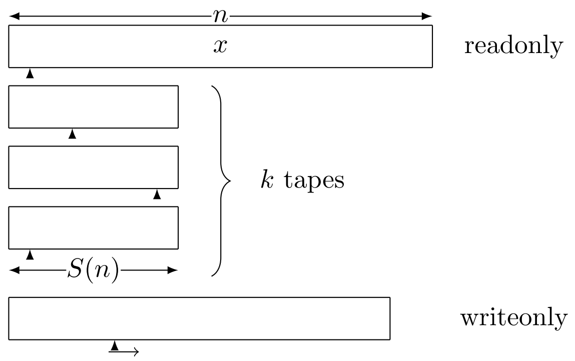


Figure 3.2. Example of a logarithmic-space Turing Machine model

Once we say that the input tape is read-only, then it is fair to say that it is not part of the memory usage. Then there are k "work tapes", each of which has length $S(n)$ and with its own pointer that moves independently. These are both read-and-write.

We can have a model that also has an optional extra write-only output tape, that is not considered part of the memory usage. The pointer on the output tape only moves to the right.

Now the question is why is the space at least $\log n$? Well, even to represent the input we need $\log n$ bits, so if we have less that $\log n$ bits we run into problems. People have studied models with $o(\log n)$ space but we will not look into these.

3.1. Deterministic and Non-Deterministic Space

So we can define the deterministic and non-deterministic space definitions $\mathsf{DSPACE}(S(n))$ and $\mathsf{NSPACE}(S(n))$. So now we have a bound on space but no bound on time, but Theorem 3.6 will show that if we have a bound on space, we automatically have a bound on time.

The most interesting amount of space is when there is logarithmic or polylogarithmic space:

Definition 3.3. $\mathsf{L} = \mathsf{DSPACE}(\log n)$ and $\mathsf{NL} = \mathsf{NSPACE}(\log n)$. ◇

And of course we have the following:

Definition 3.4. $\mathsf{PSPACE} = \bigcup_{c \in \mathbb{N}} \mathsf{DSPACE}(n^c)$. ◇

But we will see that PSPACE is very large; for example it even contains the class NP. This can be seen that we can reuse the space and we do not have a bound on time, so we can use exponential time but only polynomial space. In fact PSPACE contains the entire Polynomial Hierarchy, as we will see.

3.2. Space vs. Time

Fact 3.5. $\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE}$.

We already know all of these except $\mathsf{NL} \subseteq \mathsf{P}$. Further from the Space Hierarchy Theorem[1] we know that $\mathsf{L} \subsetneq \mathsf{PSPACE}$, but we don't know if all of the other inclusions are proper, though it is conjectured that this is so.

Theorem 3.6. $\mathsf{DSPACE}(S(n)) \subseteq \mathsf{DTIME}(2^{O(S(n))})$. *In particular,* $\mathsf{L} \subseteq \mathsf{P}$.

Proof. If we have a machine that uses at most $S(n)$ space, there are only so many configurations of the machine, and the machine moves only from one configuration to the next, and we can build a graph of these configurations and check if there is a path from the starting configuration to an accept or reject configuration.

So what do we see in the configuration? We have the contents of all of the work tapes, the state of the machine, and the position of the pointers.

Now the number of configurations is $\leqslant 2^{O(S(n))} \cdot O(1) \cdot n \cdot (S(n))^k$. This is because there are only $2^{O(S(n))}$ configurations for the work tapes, a constant number of tapes, n positions on the input tape, and $(S(n))^k$ positions on the work tapes. So the whole thing is $\leqslant 2^{O(S(n))}$.

Now the directed configuration graph G_x is defined as follows: the vertices are the configurations, and (c, c') if c can move to c' in one step.

Without loss of generality we may assume the accept and reject configurations are unique, then we have M accepts input x if and only if "there is" a path from C_{start} to C_{accept}.

[1]We will not prove this, but it is similar to the Time Hierarchy Theorem.

Hence if the machine is deterministic from each configuration there is only one path going out of it; if it is non-deterministic without loss of generality there can be at most two. So if the machine is deterministic there is one and only one path from the start state to the accept state; if the machine is non-deterministic the machine accepts if there is at least one path from the start state to the accept state.

And so we can just check if this exists in time linear in the size of the graph. □

Note that this proof also shows that $\mathsf{NSPACE}(S(n)) \subseteq \mathsf{DTIME}(2^{O(S(n))})$.

3.3. L and NL

Now we will focus on L and NL, which we will consider as sort of the analogs of P and NP. And we will have a sense of completeness for these classes. In fact the problem we just described for finding a path from one vertex of a node to another, called $s-t$ connectivity, or ST−CONN, is NL-complete.

Here is a problem that is in L: PALINDROMES $\in$ L. At every step, keep an counter i which increments and a counter $n+1-i$ which decrements, then we can look up the two indices and compare. Each counter takes at most $\log n$ space.

$\mathsf{ST{-}CONN} = \{\langle G, s, t\rangle \mid \text{there is a path } s \text{ to } t \text{ in } G\} \in \mathsf{NL}$. Start with $v = s$, then for n steps, move non-deterministically to a neighbor of v; if $v = t$ accept, otherwise if we have reached n steps, reject. It is easy to see that this can be done in logarithmic space, as all one needs to do is to keep a counter. We will show that this problem is NL-complete, once we define what completeness means.

3.3.1. NL-completeness.

Recall that a problem is NP-complete if for any problem in NP, we can reduce to it in polynomial time. We can stick with the same definition, but we run into the problem: since $\mathsf{NL} \subseteq \mathsf{P}$ we can just solve the problem in polynomial time and there is no point in reducing. Well note that reductions in NP runs in the lower class P, so for NL we can have the reductions run in the lower class L. And this turns out to be a good definition.

Definition 3.7. A reduces to B in log-space, $A \leqslant_{\log} B$, if there is a DTM M with output such that for $x \in \Sigma^*$, $x \in A \iff M(x) \in B$, such that M runs in space $O(\log|n|)$. ◇

Note that any such reduction is necessarily a polynomial time reduction.

This is a fine definition except that one needs to be a it careful when working with it:

Fact 3.8. *If $A \leqslant_{\log} B$, $B \leqslant_{\log} C$, then $A \leqslant_{\log} C$.*

The proof of this in polynomial time is straightforward, but the proof of this in logarithmic space is a bit tricky.

Proof. Well, since we have $A \xrightarrow{M} B \xrightarrow{M'} C$ taking $x \mapsto M(x) \mapsto M'(M(x))$, and $x \in A \iff M(x) \in B \iff M'(M(x)) \in C$, this seems like a fine definition but we need to make sure that $M'(M)$ runs in logarithmic space.

We can consider this as the configuration in Figure 3.9, where each machine M and M' uses some logarithmic working space.

If we store $M(x)$ the requirement will be in $\mathrm{poly}(n)$, not $\log n$.

The trick is that we do not actually need to store the entire string $M(x)$. What we will do is pretend to store $M(x)$, and store an index, say j, and request $M(x)_j$,

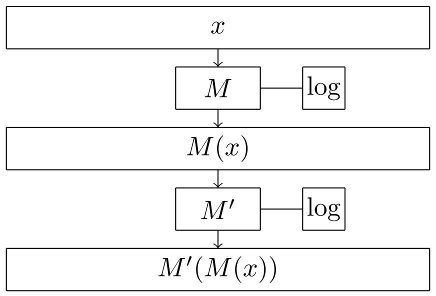


FIGURE 3.9. Configuration of $M'(M(x))$

the j-th symbol of $M(x)$. Then the machine M will run from the beginning and then produce $M(x)_j$, pass it to M', and M' will continue in its operation. □

We can use the same trick to prove the following fact:

FACT 3.10. *If $A \leqslant_{\log} B$ and $B \in \mathsf{L}$, then $A \in \mathsf{L}$. Similarly, if $A \leqslant_{\log} B$ and $B \in \mathsf{NL}$, then $A \in \mathsf{NL}$.*

DEFINITION 3.11. B is NL-complete if $B \in \mathsf{NL}$ and for all $A \in \mathsf{NL}$, then $A \leqslant_{\log} B$. ◇

THEOREM 3.12. *$\mathsf{ST{-}CONN}$ is NL-complete.*

PROOF. We already showed that this problem is in NL, and we already almost showed that this is NL-complete. Start with a problem $A \in \mathsf{NL}$, then on input x create a configuration graph G_x such that $x \in A$ if and only if there is a path from C_{start} to C_{accept} in G_x, if and only if $\langle G_x, C_{\text{start}}, C_{\text{accept}} \rangle \in \mathsf{ST{-}CONN}$. So we only need to check that this can be done in logarithmic space. The reduction is performed by enumerating through all pairs C and C' and outputting the edge (C, C') if we can reach C from C' in one step, which can be done in logarithmic space. □

3.3.2. Savitch's Theorem.

THEOREM 3.13 (Savitch). $\mathsf{NL} \subseteq \mathsf{DSPACE}((\log n)^2)$.

This is surprising, considering what we know about time classes.

The proof is simple but clever. We will give a $O((\log n)^2)$-space algorithm for $\mathsf{ST{-}CONN}$. The algorithm will be written in a recursive algorithm and seem almost stupid. But if one is familiar with how recursion is implemented on a computer, it becomes obvious that it can be done in log-squared space.

PROOF. We will have a subroutine REACH(u, w, k) that outputs YES if and only if there is a path from u to w of length at most k. On $\mathsf{ST{-}CONN}$ we will simply call REACH(s, t, n). Well how do we define REACH? Well if there is a path from u to w of step at most k then there is an intermediate vertex v such that there is a path from u to v of length $\lfloor k/2 \rfloor$ and a path from v to w of length $\lceil k/2 \rceil$, so we can just call recursively for all $v \in G$; if both recursive calls return YES, then output YES. If this fails for all v then output NO. The base case is $k = 1$, where we output YES if and only if either $u = w$ or $(u, w) \in G$.

It is important to know how a recursive algorithm is implemented. A recursive algorithm is implemented on a stack, which contains the activation record for the original call. If the routine calls itself, we add another activation record to the stack. As a call to the routine finishes, we pop the activation record from the stack. So the amount of space required is at most the depth of the recursion, multiplied by the space for one activation record.

In the case of REACH, the depth of the recursion is $O(\log n)$, and all of the information, that is, the counters, can be recorded in $O(\log n)$ space. Thus the total amount of space needed is $O((\log n)^2)$. □

Then $\mathsf{NSPACE}(S(n)) \subseteq \mathsf{DSPACE}((S(n))^2)$, since for a NTM we can create an instance of $\mathsf{ST{-}CONN}$ from the graph of its $2^{O(S(n))}$ configurations.

The following is an analog to the theorems we had previously about time.

Theorem 3.14 (Space Hierarchy Theorem). *If $S_2(n) \gg S_1(n) \geqslant \log n$, then $\mathsf{DSPACE}(S_1(n)) \subsetneq \mathsf{DSPACE}(S_2(n))$ and $\mathsf{NSPACE}(S_1(n)) \subsetneq \mathsf{NSPACE}(S_2(n))$.*

The proof of the DSPACE portion is straightforward, and we do not even need the log factor of the Time Hierarchy Theorem.

3.3.3. NL and coNL. Here is a very interesting theorem, proved by two people, Immerman and Szelepcsenyi, independently:

Theorem 3.15 (Immerman-Szelepcsenyi). *$\mathsf{NL} = \mathsf{coNL}$.*

This theorem extends to the complement of any space class.

We should first define what coNL is.

Definition 3.16. $\mathsf{coNL} = \{\overline{L} \mid L \in \mathsf{NL}\}$, where $\overline{L}$ is the set-theoretic complement of L. ◇

In general, the complent of a class will be defined similarly, so that $\mathsf{P} \subseteq \mathsf{NP}$, and $\mathsf{P} \subseteq \mathsf{coNP}$, since complement is easy to do in a deterministic setting (that is $\mathsf{P} = \mathsf{coP}$). A coNP-complete problem is $\overline{\mathsf{SAT}}$ or UNSAT. It is an open question if $\mathsf{NP} \overset{?}{=} \mathsf{coNP}$, which is not believed to be true. That $\mathsf{UNSAT} \in \mathsf{NP}$ would be surprising because it would mean that one could easily convince us that a formula is unsatisfiable.

But in the space setting, what this means is that $\overline{\mathsf{ST{-}CONN}} \in \mathsf{NL}$. Formally $\overline{\mathsf{ST{-}CONN}} = \{\langle G, s, t\rangle \mid t \text{ is unreachable from } s \text{ in } G\}$. The proof amounts to a non-determistic logarithmic space algorithm such that if t is unreachable from s, there is an accepting path. On the other hand, if there is a s-t path, then every computation rejects. This is surprising.

Lecture 5, 10 Feb

Proof of Theorem 3.15. It is good to think of the algorithm as a composition of two algorithms. The first one is rather straightforward, the second one is tricky. Putting them together is also striaghtforward.

The first algorithm is as follows: suppose we know the number r of vertices in the graph G that are reachable from s. Then count the number of vertices reachable from s in $V \setminus \{t\}$. If the count is r, then accept. More formally, we have a counter count $\leftarrow 0$, then for all $v \in V \setminus \{t\}$, guess a path $s = s_0, s_1, \ldots, s_n$. If $s_n = v$ then increment count.

The second algorithm will successively calculate in rounds the number of vertices reachable from s by paths of length k, then $k+1$, etc. More formally, let r_k

be the number of vertices reachable from s by paths of length $\leqslant k$ for $0 \leqslant k \leqslant n$. Clearly $1 = r_0 \leqslant r_1 \leqslant \ldots \leqslant r_n = r$. So we need to "compute" r_{k+1} given r_k. This is a non-deterministic algorithm, with input G, s, t, r_k. If the algorithm were deterministic, the output would just be r_{k+1}. In this case, the output is that there is at least one computation that outputs the right number r_{k+1} and and all other computations reject.

Then it is easy to combine them: start with $r_0 = 1$, then compute $r_1, r_2, \ldots, r_n$ via algorithm #2, then run algorithm #1.

Now we need to describe how the second algorithm works. Start with a variable count $\leftarrow 0$, that will eventually be the number of vertices reachable on a path of length $\leqslant k+1$. Then for every vertex $v \in V$, set a variable flag $\leftarrow$ false, and for all $u \in V$, guess a path $s = s_0, s_1, \ldots, s_n$. If $s_k = u$, then if $v = u$ or v is in the neighborhood of u, set flag=true. After all $u \in V$ have been tested and flag = true, increment count. Then after looping over all $v \in V$, output count.

There is a caveat, however, which will we fix. The behavior that we want from the algorithm is that there is some path that, provided we guessed correctly, will output the correct value. But the algorithm as it is can output incorrect answers, as it always outputs. For example if it always makes incorrect guesses, the output will be stuck at 0.

Well, what went wrong is as follows: first, we never made use of r_k. Now there are r_k eligible vertices u, so unless we did that we cannot be sure of the correct solution. So we will add in another counter that counts if u is an eligible vertex; if at the end this counter falls short of r_k, then reject. $\square$

We will mention one more theorem that is quite striking.

THEOREM 3.17 (Reingold). UNDIR−ST−CONN, *or s-t connectivity on undirected graphs, is in L.*

Think of this as a maze with n vertices, then we can enter at s and we can only exit at t, and we want to get out of the maze with as little computational space as possible. We will see what this means. In some sense, we are very dumb and have very little memory (note that if we had infinite memory we could just reconstruct the entire graph). So we can only remember a constant number of vertices.

CHAPTER 4

The Polynomial Hierarchy

As a brief overview, we will add to P and NP an infinite hierarchy of classes $\mathsf{P} \subseteq \mathsf{NP} \subseteq \Sigma_2 \subseteq \Sigma_3 \subseteq \dots$. Then $\mathsf{P} = \Sigma_0$ and $\mathsf{NP} = \Sigma_1$. The motivation for studying these is the same as the one for studying NP. We define $\mathsf{PH} = \bigcup_{k=0,1,2,\dots} \Sigma_k$.

A nice result that comes out of this as the following: take the Graph Isomorphism Problem $= \{\langle G, H\rangle \mid G \text{ is isomorphic to } H\}$. Clearly this problem is in NP, but it is not known to be in P. But there is strong evidence to show that this problem is not NP-complete. So this is a candidate for a problem that is between P and NP-complete, and there is a theorem that says that if $\mathsf{P} \neq \mathsf{NP}$ then such problems must exist. The proof comes from the hierarchies.

There are two ways to define this hierarchy of classes.

4.1. Definition using Alternating Quantifiers

Well, we can define NP as the class of languages A such that there is a polynomial time DTM V_A such that for any input x,

$$x \in A \iff \exists y \text{ such that } V_A(x, y) = 1,$$

with $|y| = \text{poly}(|x|)$. The representative for this class is SAT. We also defined the class coNP, and the representative for this class is UNSAT which is equivalent to $\mathsf{TAUT} = \{\phi \mid \text{Every assignment is a satisfying assignment}\}$ by negating ϕ. So we can write a characterization of coNP as follows: it is the class of all languages B such that there is a polynomial time DTM V_B such that for any input x, $x \in B \iff \forall y, V_B(x, y) = 1$.

In fact, we will prove this. Take $B \in \mathsf{coNP}$, then $\overline{B} \in \mathsf{NP}$. Then there exists V such that $x \in \overline{B} \iff \exists y, V(x, y) = 1$. Then

$$x \in B \iff \neg x \in \overline{B} \iff \neg \exists y, V(x, y) = 1 \iff \forall y, V_B(x, y) = 1$$

where $V_B = \neg V$.

Now we look at the definitions and they are almost identical. Then the hierarchy Σ_2, Σ_3 can be defined by adding an alternating sequence of $\exists$ and $\forall$ quantifiers to the definition.

Before we give the formal definition we will give an example of a real life problem where this might occur. Suppose C is a circuit implementing a certain function. Now you want to convince a customer that this is the chip of the smallest size that can implement such functionality. That is, for $\forall C'$ with $|C'| < |C|$, $\exists A$ such that $C'(A) \neq C(A)$. So we have $\forall$ followed by $\exists$ followed by something that can be verified deterministically. This is precisely a problem in Π_2, which we will define.

Definition 4.1. Σ_2 is the class of languages A such that there exists a polynomial time DTM V_A such that $x \in A \iff \exists y_1 \forall y_2,\ V_A(x, y_1, y_2) = 1$. $\diamond$

An example of a problem in Σ_2 is a circuit $C(y_1,\dots,y_n,z_1,\dots,z_n)$, then we ask is there an assignment $y_1,\dots,y_n$ for all assignments $z_1,\dots,z_n$ does C outputs 1? Formally,

$$C \in L \iff \exists(y_1,\dots,y_n)\forall(z_1,\dots,z_n)V_A(C,y_1,\dots,y_n,z_1,\dots,z_n)=1$$

A small point is that $\exists(y_1,\dots,y_n)$ is the same as $\exists y_1\exists y_2\dots$, but it does not matter so much how many instances of $\exists$ appears as much as that they are not interrupted by an instance of $\forall$.

Definition 4.2. $\Pi_2 = \mathsf{co}\Sigma_2$. $\diamond$

Fact 4.3. *Π_2 is the class of languages B such that there exists a polynomial time DTM V_B such that $x \in B \iff \forall y_1\exists y_2,\ V_B(x,y_1,y_2)=1$.*

It is easy to see that $\mathsf{NP} \subseteq \Sigma_2$ since we can just ignore the y_2, and $\mathsf{coNP} \subseteq \Sigma_2$ since we can just ignore y_1. In general we will get the inclusions shown in Figure 4.4.

$$\begin{tikzcd}
 & \mathsf{NP} \arrow[r, dash] \arrow[dr, dash] & \Sigma_2 \arrow[r, dash] \arrow[dr, dash] & \Sigma_3 \arrow[r, dash] \arrow[dr, dash] & \Sigma_4\cdots \\
\mathsf{P} \arrow[ur, dash] \arrow[dr, dash] & & & & \\
 & \mathsf{coNP} \arrow[r, dash] \arrow[ur, dash] & \Pi_2 \arrow[r, dash] \arrow[ur, dash] & \Pi_3 \arrow[r, dash] \arrow[ur, dash] & \Pi_4\cdots
\end{tikzcd}$$

Figure 4.4. Inclusion map of Σ_ks and Π_ks

Definition 4.5. Σ_k is the class of languages A such that there exists a polynomial time DTM V_A such that

$$x \in A \iff \exists y_1\forall y_2\dots Q_k y_k, V_A(x,y_1,\dots,y_k)=1$$

where $Q_k = \exists$ if k is odd, and $Q_k = \forall$ if k is even. $\diamond$

Definition 4.6. $\Pi_k = \mathsf{co}\Sigma_k$, the class of languages B so that there is a polynomial time deterministic Turing Machine V_B so that

$$x \in B \iff \forall y_1\exists y_2\dots Q_k y_k, V_A(x,y_1,\dots,y_k)=1$$

where $Q_k = \forall$ if k is odd, and $Q_k = \exists$ if k is even. $\diamond$

In general we will not need to deal with classes of higher k. It is easy to extend our circuit-based problem to the class $\mathsf{CIRCUIT{-}SAT}_k$, which will be the representative of the class.

Definition 4.7. $\mathsf{CIRCUIT{-}SAT}_k = \{\langle C(y_1,y_2,\dots,y_k)\rangle \mid \exists y_1\forall y_2\exists y_3\dots Q_k y_k, C(y_1,y_2,\dots,y_k)=1\}$, where each y_i is a string. $\diamond$

Fact 4.8. *$\mathsf{CIRCUIT{-}SAT}_k$ is complete for Σ_k under polynomial time reductions.*

This follows from $\mathsf{CIRCUIT{-}SAT}_k \in \Sigma_k$, and for $A \in \Sigma_k$, $A \leqslant_P \mathsf{CIRCUIT{-}SAT}_k$ via basically the Cook-Levin Theorem.

So we described one (syntactic) way of defining these classes. There is another way using what are known as oracle machines. The idea of oracles is very important in complexity, so this is a good opportunity to familiarize ourselves with these oracles. Then we will give definitions of these Σ_k classes using oracles and show that they are equivalent.

4.2. Definition using Oracles

An *oracle* is a black box that can answer our questions in constant time, even if they are hard. So imagine that we have an oracle that can tell us if something is the solution to an NP-hard problem. Then using this oracle, we can solve NP-hard problems easily, so we investigate if we can do even better.

4.2.1. Computation via Oracles.

DEFINITION 4.9. Let A be a language, then M^A is a Turing Machine augmented with an oracle, called an A-*oracle*, such that the machine can ask the oracle a language membership question, given a string z, is $z \in A$? $\diamond$

DEFINITION 4.10. P^A is the class of all languages that can be decided in deterministic polynomial time with access to an A-oracle, an oracle for membership in language A. $\diamond$

Note that $A \in \mathsf{P}^A$, and if $A \in \mathsf{P}$, then $\mathsf{P}^A = \mathsf{P}$. Next, if on input x, if we asked is $z_1 \in A$, is $z_2 \in A$, ..., is $z_\ell \in A$? Well, it is clear that $|\ell| \leqslant \operatorname{poly}(n)$. Furthermore, we can not ask questions whose input is greater than polynomial in the length of the input, that is, $|z_i| \leqslant \operatorname{poly}(n)$. We can formalize this by saying that our model is that in order to ask a question we must write down the question onto a tape, so then if the machine runs in polynomial time the length of the question must be polynomial.

Now the definition P^A is already weird; now we will get even more weird.

DEFINITION 4.11. Let $\mathscr{C}$ be a class of languages, such as NP or Σ_2 or whatever. Then define $\mathsf{P}^{\mathscr{C}} = \bigcup_{A \in \mathscr{C}} \mathsf{P}^A$. $\diamond$

Then note that if A is $\mathscr{C}$-complete then $\mathsf{P}^A = \mathsf{P}^{\mathscr{C}}$. So $\mathsf{P}^{\mathsf{P}} = \mathsf{P}$. So when the machine is deterministic there is no fun. The fun begins when we have non-determinism.

DEFINITION 4.12. NP^A is the class of languages accepted in a polynomial time by a non-deterministic machine with an A-oracle. $\diamond$

Well, if $A \in \mathsf{P}$ then $\mathsf{NP}^A = \mathsf{NP}$, but we will see that $\mathsf{NP}^{\mathsf{NP}} = \Sigma_2$. Just as we believe that $\mathsf{P} \neq \mathsf{NP}$, we believe that $\mathsf{NP} \neq \Sigma_2$.

Note that $\mathsf{NP}^A \cup \mathsf{NP}^B \subseteq \mathsf{NP}^{A,B}$, it is not clear that the two are equal. Lecture 6, 12 Feb

4.2.2. Σ_k via Oracles.

THEOREM 4.13. $\Sigma_2 = \mathsf{NP}^{\mathsf{CIRCUIT-SAT}} (= \mathsf{NP}^{\mathsf{NP}} = \mathsf{NP}^{\Sigma_1})$.

This will generalize to $\Sigma_k = \mathsf{NP}^{\Sigma_{k-1}}$, so this is another way to define the hierarchy.

PROOF. We first prove that $\Sigma_2 \subseteq \mathsf{NP}^{\mathsf{CIRCUIT-SAT}}$. We will see that we only need to ask one question. Take $A \in \Sigma_2$. Then

$$x \in A \iff \exists y_1 \forall y_2, V(x, y_1, y_2) = 1 \iff \exists y_1 \neg \exists y_2, V(x, y_1, y_2) = 0$$

which, since deterministic verifiers are equivalent to circuits, is the same as saying $\exists y_1 \neg \exists y_2 C(y_2) = 1$ where $C = \overline{V}(x, y_1, \cdot)$ where x, y_1 are "hardwired". So then this is equivalent to saying $\exists y_1, C \notin \mathsf{CIRCUIT-SAT}$. This gives a non-deterministic algorithm to check if $x \in A$: guess y_1, construct C and ask if $C \in \mathsf{CIRCUIT-SAT}$, which it can do since it has a $\mathsf{CIRCUIT-SAT}$-oracle, then accept if and only if the answer is No.

Now we will prove the reverse inclusion $\mathsf{NP}^{\mathsf{CIRCUIT-SAT}} \subseteq \Sigma_2$. This is an interesting proof. So take $B \in \mathsf{NP}^{\mathsf{CIRCUIT-SAT}}$. B is accepted by a polynomial time DTM M with a $\mathsf{CIRCUIT-SAT}$-oracle. In general, this computation may be very complicated, but without loss of generality, we can show that it ends up only asking one question. So we'll "simulate" M by M' that makes only one query to the $\mathsf{CIRCUIT-SAT}$, at the very end. The idea of M' is as follows: first guess the whole computation, then there are two kinds of queries (that are circuits): those on which the oracle say Yes and those on which the oracle say No. Then for all the circuits on which the oracle says Yes we could have in fact guessed a satisfying assignment. For the ones on which the oracle says No we can just combine all of them by taking a logical OR and then asking the oracle.

So M' will do the following on input x: it will first guess the computation of M: all the non-deterministic choices and all queries and their answers. Then if $C_1, \ldots, C_r$ are the queries whose answers are guessed to be Yes, guess satisfying assignments to these circuits. Now we verify that the entire computation is correct. There is still some uncertainty, as there were circuits that were guessed to be No. Then we will just make sure of that by bunching them up and asking the oracle. So let $C'_1, \ldots, C'_s$ be the circuits whose answers are guessed to be No, then take $C' = \bigvee_{i=1}^{s} C'_i$ and ask the oracle if $C' \in \mathsf{CIRCUIT-SAT}$. Accept if and only if the answer is No.

We conclude by observing that

$$x \in B \iff \exists y, C' \text{ such that } V(y, C') = 1 \text{ and } C' \notin \mathsf{CIRCUIT-SAT},$$

where y represents all of the non-deterministic choices made. Then this is equivalent to saying $\exists y, C' \forall a, V(y, C') = 1 \wedge C'(a) = 0$. This shows that $B \in \Sigma_2$. □

Exercise 4.14. Show that in general $\Sigma_{k+1} = \mathsf{NP}^{\Sigma_k} (= \mathsf{NP}^{\mathsf{CIRCUIT-SAT}_k})$. ◇

4.3. Theorems about the Polynomial Hierarchy and Oracles

We already strongly believe $\mathsf{P} \neq \mathsf{NP}$, so here is a theorem that gives further evidence.

Theorem 4.15. *If $\mathsf{P} = \mathsf{NP}$ then $\Sigma_k = \mathsf{P}$ for $k \geqslant 1$.*

Proof. We already have $\Sigma_1 = \mathsf{P}$. Then we have

$$\Sigma_2 = \mathsf{NP}^{\mathsf{NP}} = \mathsf{NP}^{\mathsf{P}} = \mathsf{NP} = \Sigma_1$$

Then $\Sigma_3 = \mathsf{NP}^{\Sigma_2} = \mathsf{NP}^{\Sigma_1} = \Sigma_2 = \ldots$. Then just continue in this manner. □

The standard way of stating this theorem is that "If $\mathsf{P} = \mathsf{NP}$ then the "polynomial hierarchy collapses" to the 0th level".

A similar theorem can be proved for another strongly believed conjecture, that $\mathsf{NP} \neq \mathsf{coNP}$:

Theorem 4.16. *If $\mathsf{NP} = \mathsf{coNP}$ then the "polynomial hierarchy collapses" to the 1st level.*

Proof. In the previous proof we just proved that $\Sigma_2 = \mathsf{NP}$ then everything came down. So it suffices to show that $\Sigma_2 = \mathsf{NP}$. Well for $A \in \Sigma_2$ we have $x \in A \iff \exists y_1 \forall y_2, V(x, y_1, y_2) = 1$. Think of x, y_1 as fixed, then this is equivalent to saying there is some circuit that is a tautology, but TAUT is coNP-complete. So using the assumption that $\mathsf{NP} = \mathsf{coNP}$, we get $x \in A \iff \exists y_1, y_3 C(y_3) = 1$. So $A \in \mathsf{NP}$. $\square$

Once we are familiar with computation with oracles, it is easy to see that if we want to show $\mathsf{P} \subsetneq \mathsf{NP}$, the technique of diagonalization as a tool is "insufficient" to separate P from NP.

Recall the theorem $\mathsf{DTIME}(n^2) \subsetneq \mathsf{DTIME}(n^3)$. We can see how the proof works and conclude that the same proof will prove the following:

THEOREM 4.17. $\mathsf{DTIME}^A(n^2) \subsetneq \mathsf{DTIME}^A(n^3)$ *for any language A.*

Now suppose that there is a proof, using diagonalization, gave $\mathsf{P} \subsetneq \mathsf{NP}$. Then the same proof ought to give $\mathsf{P}^A \subsetneq \mathsf{NP}^A$ for any language A, but this statement is false.

FACT 4.18. *There is a language A such that $\mathsf{P}^A = \mathsf{NP}^A$.*

A real-world analogy is the following: say that we have a weak person and a strong person, but we give both of them a very strong weapon, like a missile launcher. Then it is clear that any difference before no longer matters; they have the same power. So we can make A a very powerful language and then we are done.

So we have $\mathsf{P} \subseteq \mathsf{NP} \subseteq \Sigma_2 \subseteq \Sigma_3 \subseteq \ldots \subseteq \mathsf{PSPACE}$. We did not prove this yet, but we will. Using this, we can do the following: let A be PSPACE-complete, then $\mathsf{P}^A = \mathsf{NP}^A = \mathsf{PSPACE}$.

This is evidence that diagonalization is not good enough to separate P from NP.

We will now state some more interesting theorems that we will not prove.

THEOREM 4.19 (Baker-Gill-Solovay). *There is an oracle A such that $\mathsf{P}^A \subsetneq \mathsf{NP}^A$.*

There is a term for this "world with oracles", called a "relativized world". So in this "relativized world" P can be separated from NP.

What the proof does is that we can take A to be a random language constructed by, for every string x then $x \in A$ with probability $\frac{1}{2}$, then this theorem holds with probability 1.

THEOREM 4.20 (Ladner). *If $\mathsf{P} \subsetneq \mathsf{NP}$ there exists $A \in \mathsf{NP}$ such that $A \notin \mathsf{P}$ and A is not NP-complete.*

4.4. PSPACE Revisited

Recall that we defined $\mathsf{P} \subseteq \mathsf{NP} \subseteq \Sigma_2 \subseteq \Sigma_2 \subseteq \ldots \subseteq \Sigma_k \subseteq \ldots \subseteq \mathsf{PH} \subseteq \mathsf{PSPACE}$ with $\mathsf{PSPACE} = \bigcup_{c \in \mathbb{N}} \mathsf{DSPACE}(n^c)$.

We will present a problem that is PSPACE-complete. The inclusion $\mathsf{PH} \subseteq \mathsf{PSPACE}$ follows from that the representative problem of each Σ_k is a circuit satisfiability problem.

DEFINITION 4.21. A *quantified circuit* is a circuit C such that

$$Q_1 x_1 \ldots Q_n x_n C(x_1, x_2, \ldots, x_n), x_i \in \{0, 1\}$$

where $Q_i \in \{\exists, \forall\}$ $\diamond$

Definition 4.22. The language quantified circuit satisfiability is $\mathsf{QC-SAT} = \{Q_1x_1 \dots Q_nx_nC(x_1, \dots, x_n) \mid \text{this quantified circuit is "true"}\}$. ◇

There is a version of this problem using boolean formulas called TQBF. But for our purposes boolean formulas and circuits are the same.

Observation 4.23. $\mathsf{QC-SAT} \in \mathsf{PSPACE}$. ◇

This follows because we can just brute force over all assignments in polynomial space. Formally, write a recursive function $\mathrm{EVAL}(Q_1x_1 \dots Q_nx_n\ C(x_1, \dots, x_n))$ that sets

$$\begin{aligned} a &\leftarrow \mathrm{EVAL}(Q_2x_2 \dots Q_nx_nC(0, x_2, \dots, x_n)) \\ b &\leftarrow \mathrm{EVAL}(Q_2x_2 \dots Q_nx_n\ C(1, x_2, \dots, x_n)) \end{aligned}$$

If $Q_1 = \exists$, output $a \vee b$; if $Q_1 = \forall$, output $a \wedge b$.

The space is at most the recursion depth times the space for one "instantiation", which is at most $n\,\mathrm{poly}(n) = \mathrm{poly}(n)$. So this problem is in PSPACE.

Recall that $\mathsf{CIRCUIT-SAT}_k$ is the complete problem for Σ_k. It is clear from this construction that $\mathsf{QC-SAT}$ contains all the problems $\mathsf{CIRCUIT-SAT}_k$, since $\mathsf{CIRCUIT-SAT}_k$ is a special case of $\mathsf{QC-SAT}$ with $\leqslant k-1$ alterations.

The interesting theorem is the following:

Theorem 4.24. *$\mathsf{QC-SAT}$ is PSPACE-complete.*

Before we prove this theorem, let us make some comments about $\mathsf{QC-SAT}$. We can view this problem as a game between two players. Take an instance $I = \exists x_1 \forall x_2 \exists x_3 \dots C(x_1, x_2, \dots, x_n) = 1$. The two-player game is as follows: the first player chooses x_1, then the second player chooses x_2, then the first player chooses x_3, and so on. Then after choosing these run C on these; if $C(x_1, \dots, x_2) = 1$ player 1 wins; otherwise player 2 wins. Then $I \in \mathsf{QC-SAT}$ if and only if player 1 has a winning strategy: there is a value player 1 can choose such that for any value player 2 chooses, player 1 can choose a value, and so forth and still win. So the problem of deciding if a player has a winning strategy is PSPACE-complete.

In particular, we have the problem $\mathsf{GEOGRAPHY}$, a game played on a directed graph of n vertices. There is a starting vertex v_0, then at each step the player has to pick a vertex that is a neighbor of the previous vertex without choosing a vertex that has already been chosen. A player loses if there is no valid choice.

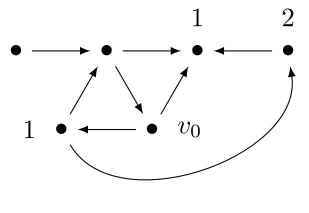


Figure 4.25. Example of an instance of a game of $\mathsf{GEOGRAPHY}$, in which Player 1 wins.

Lecture 7, 19 Feb

Proof of Theorem 4.24. To prove that $\mathsf{QC-SAT}$ is PSPACE-complete, we show that for all $A \in \mathsf{PSPACE}$, $A \leqslant_P \mathsf{QC-SAT}$. The proof resembles to that of the Cook-Levin Theorem.

So the computation of the machine runs in polynomial space, but potentially in exponential time. But we still want to capture it in a quantified circuit of polynomial size.

Start with $A \in \mathsf{PSPACE}$. Then by definition A is accepted by a deterministic TM M in space n^k for some constant k, and hence, without loss of generality, in time 2^{n^k}. So the machine can be captured in a sequence of configurations $C_0 = C_{\text{start}}, C_1, \ldots, C_i, C_{i+1}, \ldots, C_{2^{n^k}}$. Since the machine works in polynomial space, the length of each configuration is n^k. So really, the decision as to whether $x \in A$ is the same as saying whether we can start in C_{start} and end up at C_{accept}. That is, if $C_{2^{n^k}} = C_{\text{accept}}$. Equivalently, the question $x \in A$ is the same as the question REACH$(C_{\text{start}}, C_{\text{accept}}, m = 2^{n^k})$, which needs to be encoded as a $\mathsf{QC{-}SAT}$ formula.

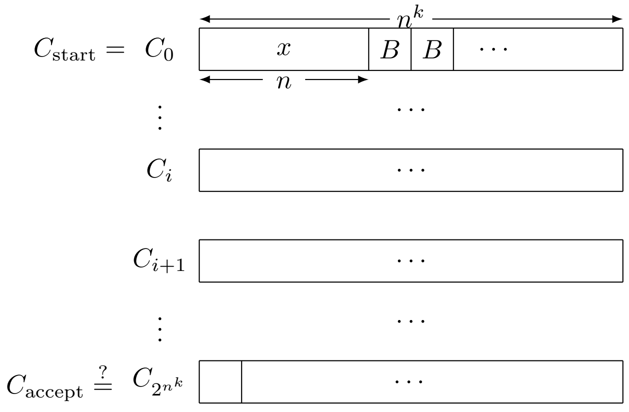

n^k
$C_{\text{start}} =$ C_0
x
B
B
$\cdots$
n
C_i
C_{i+1}
$C_{\text{accept}} \overset{?}{=}$ $C_{2^{n^k}}$

FIGURE 4.26. The sequence of configurations

Now observe that if we can reach from the start step to the accept step in m steps, then we can reach some intermediate step from the start step in $m/2$ steps, then the accept step from the intermediate step in $m/2$ steps. That is, REACH$(C_{\text{start}}, C_{\text{accept}}, m) = \exists w,$ REACH$(C_{\text{start}}, W, \frac{m}{2}) \wedge$ REACH$(W, C_{\text{accept}}, \frac{m}{2})$. Of course then we can recurse. Then we drop down to the step where we have predicates of the form REACH$(X, Y, 1)$. Well is a circuit of size poly(n), so we can write a big quantified circuit formula that captures the configurations.

Now the problem is that this is not in polynomial time, for it has m levels of recursion and so has size around 2^{n^k}. Furthermore, it only uses the existential quantifier. Now there is one little trick we can use. When we recurse we will capture both predicates in a single one. We write the predicate[1]

$$\text{REACH}(X, Y, m) =$$
$$\exists W \forall U, V (U = X \wedge V = W) \vee (U = W \wedge V = Y) \implies \text{REACH}(U, V, \frac{m}{2})$$

So again when we recurse we eventually we have something like REACH$(D, E, 1)$. Now a complaint may be that we have quantifiers interleaved with circuits. But we

[1]If we are unhappy with writing this as an implication we can simply rewrite it as the formula $\neg((U = X \wedge V = W) \vee (U = W \wedge V = Y)) \vee$ REACH$(U, V, \frac{m}{2})$.

can just stare at it and see that logically, it would not change anything if we moved all of the quantifiers to the beginning. □

We talked about how this can be viewed as a two-player game; there are many nice problems of this form.

CHAPTER 5

Randomized Complexity Classes

The main question is if there are problems that do not have a deterministic algorithm, but can be solved if we allow randomness. A priori, we want to say "why not?" just as we want to say $\mathsf{P} \neq \mathsf{NP}$. Over the last 15 years or so, researchers have made great progress and there is strong evidence that randomization does not help, that is, if there is a randomized algorithm for a problem there is a deterministic one that runs in time polynomial of the running time of the randomized algorithm.

Whether or not it helps or not in theory, it seems to help in practice: a problem may have a randomized algorithm that is simpler or easier to implement that the deterministic one.

But we still do not know for fact that every problem that has a randomized algorithm has a deterministic one. The evidence comes from different directions. For example, we have theorems such as "Under certain Complexity Theoretic assumptions, $\mathsf{P} = \mathsf{BPP}$". Furthermore 10-15 years ago, we did have a list of problems that had randomized algorithms but not deterministic algorithms, but over the years, people have killed examples off of the list and now it is very short; the only compelling example left is the problem Symbolic Determinant Zero-Testing.

An example of a problem that was on this list but is no longer is primality testing or PRIME. We know $\mathsf{PRIME} \in \mathsf{P}$.

The problem of Symbolic Determinant Zero-Testing is as follows: We are given an $n \times n$ matrix Q with entries $Q_{ij} \in \mathbb{Z}[x_1, x_2, \ldots, x_n]$ with bounded degree$\leqslant d$[1]. Now compute the determinant $\det(Q)$. We know that this is a polynomial in the same amount of variables, with degree $\leqslant dn$. The question is, is $\det(Q)$ as a polynomial identical to 0?

This is a problem for which we know a randomized algorithm, which we will describe. But we do not know if this problem has a deterministic polynomial-time algorithm.

5.1. An Example of a Randomized Algorithm

Loosely, the randomized algorithm does the following: suppose that the determinant was indeed 0, then clearly it is 0 if we assign some concrete integral values to the variables. If we assign concrete values to Q we can indeed compute the determinant very quickly (in polynomial time), such as by using Gaussian Elimination. We can convince ourselves very quickly that this Gaussian Elimination will not work in polynomials. In principle, a polynomial of degree nd in n variables is

[1]The issue with integers is that as soon as we start dealing with them we have to be careful about the size of the integers involved. If we want to avoid talking about bits and whatnot we can do this over a large enough prime field, so we can ignore this problem.

in fact too large and has too many coefficients, so we have no hope of writing it down in polynomial time.

Formally, we do the following: pick $x_i \in \{1, 2, \ldots, L = 100nd\}$ for $i = 1, \ldots, n$ independently. Let $Q^* = Q(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are concrete integer values. Now compute $\det(Q^*)$ in polynomial time. If $\det(Q^*) = 0$ then say "Yes", otherwise say "No".

Now the output of a randomized algorithm can depend on the random choices made. So the output has a certain probability of being correct and a certain probability of being incorrect, which adds to 1. A randomized algorithm is regarded as "correct" if it is correct with high probability.

So for our randomized algorithm for Symbolic Determinant Zero-Testing, if $\det(Q) = 0$ the output is "Yes" with probability 1. Now if $\det(Q) \neq 0$ the output should be "No" with "high probability".

Recall the following fact:

THEOREM 5.1. *Say $P(x)$ is a degree d polynomial in a single variable x. If x is given a value from $\{1, 2, \ldots, L\}$, then* $\Pr[P(x) = 0] \leqslant \frac{d}{L}$

The Schwartz-Lippel lemma states that this is true even in many variables, that is, the previous theorem holds for $x = (x_1, \ldots, x_n)$ and $x_1, \ldots, x_n$ given values from $\{1, 2, \ldots, L\}$ independently.

So if $\det(Q) \neq 0$ the probability that the output is "Yes" is at most $\frac{nd}{L}$ where we chose $L = 100nd$, so the probability is at most $\frac{1}{100}$[2].

This is the problem such that if we have a deterministic algorithm for it, we more or less (that is, with many caveats) have a deterministic algorithm for any problem with a randomized algorithm.

5.2. The class BPP

Now we can formally define classes of problems with randomized algorithms, which we call BPP.

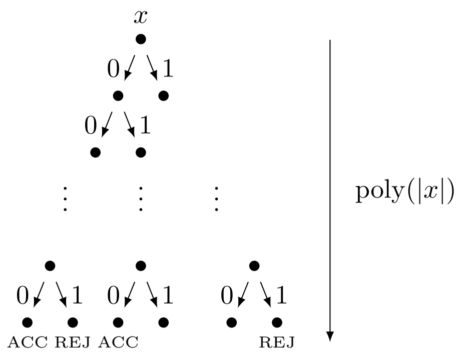


FIGURE 5.2. The probabilistic computation tree

[2]This is completely arbitrary as we could have chosen L to be as large as we want, to make this completely negligible.

Definition 5.3. A probabilistic polynomial time algorithm (PPT) is a polynomial time Turing Machine M which at each step chooses a random bit $\in \{0,1\}$ and its move may depend on this bit (see Figure 5.2). The probability that M accepts x is the number of accepting leaves over the number of leaves. $\diamond$

The idea is that instead of moving nondeterministically we move randomly.

In the non-deterministic model, the machine is supposed to accept if there is at least one leaf accepting. But here the machine is supposed to succeed with high probability, no matter what x is. So either the accept leaves are in high majority, or the reject leaves are in high majority. So clearly only special machines have this property, and those are the ones that qualify as randomized algorithms.

Definition 5.4. BPP is the class of languages L such that there is a PPT M such that

$$\begin{array}{ccccc} x \in L & \Longleftrightarrow & \Pr[M(x) = \text{accept}] & \geqslant & \frac{99}{100} \\ x \neq L & \Longleftrightarrow & \Pr[M(x) = \text{reject}] & \leqslant & \frac{1}{100} \end{array}$$

$\diamond$

BPP stands for Bounded Probabilistic Polynomial Time. What the word "Bounded" refers to is that if we look at the acceptance probabilities for if $x \in L$ and $x \neq L$, the numbers differ by a reasonable amount. It also refers to how it is used in practice: for example, we can run an algorithm 100 times and then take the majority of the answers, then the probability of making an error is incredibly low, so that it is just as good as a deterministic algorithm from a practical point of view.

Suppose we had an algorithm where $x \in L$ has $\Pr[\text{accept}] \geqslant \frac{1}{2} + \frac{1}{2^n}$ and $x \notin L$ has $\Pr[\text{reject}] \leqslant \frac{1}{2}$. This is useless, as to get a good difference we need to run the algorithm an exponential number of times.

Now if we look at the definition of BPP we can complain that the number $\frac{99}{100}$ is completely arbitrary. But this is okay:

Fact 5.5. *Suppose $L \in$ BPP. Then there is a PPT M such that if $x \in L$, $\Pr[M(x) = accept] \geqslant 1 - \frac{1}{2^n}$ and if $x \notin L$, $\Pr[M(x) = reject] \leqslant \frac{1}{2^n}$ where $n = |x|$.*

So as long as we we set the error to be reasonable we can still find a good algorithm.

Proof. The proof is simply that we can create a machine M':

```
$M'$ := "On input $x$
    Run $M$ $\ell$ times
    If $M$ outputs "Yes"$\geqslant \frac{\ell}{2}$ times, output "Yes"
    Otherwise, output "No"
```

Consider the case when $x \in L$ (the case where $x \notin L$ is symmetric). Then $\Pr[M' \text{ makes an error}] = \Pr[\leqslant \frac{\ell}{2} \text{ runs of } M \text{ accepted}] \leqslant 2^{-\Omega(\ell)}$. This bound is called a Chernoff bound. So we can always choose large $\ell = O(n)$. $\square$

This shows that we could just as easily write the definition of BPP with c instead of $\frac{99}{100}$ and s instead of $\frac{1}{100}$ such that $c - s \geqslant \frac{1}{\text{poly}(|x|)}$. Then we can boost our confidence to close to 1.

One last comment is that we know that every NTM defines a language. But not every proabilistic Turing Machine defines a language: we must have the property that it must always overwhelmingly accept or overwhelmingly reject. We do not know any way of testing this property for a probabilistic Turing Machine. In particular, we have no time hierarchy theorems or even know if BPP has a complete problem. Recall that it was very easy to create a complete problem for NP.

The class NP is called a syntactic complexity class. BPP is called a semantic complexity class. It seems that syntactic complexity classes easily lead to having complete problems.

Lecture 8, 24 Feb

In summary of the above, will now state a slightly different presentation of BPP, using non-determinism and verification.

Definition 5.6. $L \in \mathsf{BPP}$ if and only if there is a polynomial time DTM V such that $x \in L \implies \Pr_y[V(x,y)=1] \geqslant 1-e$ and $x \in L \implies \Pr_y[V(x,y)=1] \leqslant e$ where e is small, such as $e = \frac{1}{100}$ or $e = \frac{1}{2^n}$. ◇

5.3. BPP vs. Σ_2

We have shown the relation $\mathsf{P} \subseteq \mathsf{NP} \subseteq \Sigma_2$, we will now prove the relation $\mathsf{P} \subseteq \mathsf{BPP} \subseteq \Sigma_2$.

Theorem 5.7. *$\mathsf{BPP} \subseteq \Sigma_2$.*

The proof is very short, providing we prove a small lemma on the side:

Lemma 5.8. *Take $S = \{0,1\}^m$, Suppose there is a very large subspace $A \subseteq S$, say $|A| \geqslant (1-\frac{1}{2^n})|S|$ where $m = \text{poly}(n)$. Then there is a notion of translation, since we can think of S as a vector space, i.e. A translated by a vector $z \in \{0,1\}^m$ is $A \oplus z = \{a \oplus z \mid a \in A\}$. Then there is a small number of translations $z_1, z_2, \ldots, z_\ell$, $\ell = \text{poly}(n)$ of A that will cover the whole space, that is, $\bigcup_{i=1}^{\ell} A \oplus z_i = S$.*

Proof. Choose $z_1, z_2, \ldots, z_\ell$ randomly from $\{0,1\}^m$. Then for some fixed $w \in \{0,1\}^m$, $\Pr[w \notin \bigcup_{i=1}^{\ell} A \oplus z_i] \leqslant (\frac{1}{2^n})^\ell$. Then taking the union bound over w we have $\Pr[S \neq \bigcup_{i=1}^{\ell} A \oplus z_i] \leqslant 2^m (\frac{1}{2^n})^\ell \ll 1$. □

Proof of Theorem 5.7. Now take a language $L \in \mathsf{BPP}$, then we know that if $x \in L$ the verifier accepts with high probability $\geqslant 1 - \frac{1}{2^n}$; if $x \notin L$ then the verifier accepts with very low probability $\leqslant \frac{1}{2^n}$.

So take $S = \{0,1\}^m$ where $m = |y|$ and take $A_x = \{y \mid V(x,y) = 1\}$. If $x \in L$ then $|A_x| \geqslant (1 - \frac{1}{2^n})|S|$, and if $x \notin L$ then $|A_x| \leqslant \frac{1}{2^n}|S|$. So we know that if $x \in L$ there are a small number of translations so that A_x covers the space, and if $x \notin L$ then a small number of translations will be negligible. That is,

$$\begin{aligned} x \in L \quad &\iff \quad \exists z_1, \ldots, z_\ell,\ \ell = \text{poly}(n) \text{ such that } \textstyle\bigcup_{i=1}^{\ell} A_x \oplus z_i = S \\ &\iff \quad \exists z_1, \ldots, z_\ell \forall y \in S, A_x \oplus z_i = y \text{ for some } i \in \{1, \ldots, \ell\} \\ &\iff \quad \exists z_1, \ldots, z_\ell \forall y \in S, z_i \oplus y \in A_x \text{ for some } i \in \{1, \ldots, \ell\} \\ &\iff \quad \exists z_1, \ldots, z_\ell \forall y, V(x, z_i \oplus y) = 1 \text{ for some } i \in \{1, \ldots, y\} \end{aligned}$$

where V is a deterministic polynomial time verifier. So $L \in \Sigma_2$. □

Note that $\mathsf{BPP} \subseteq \Pi_2$ since BPP is closed under complements.

5.4. One-sided and zero-sided error: the classes RP and ZPP

Note that BPP allows two-sided error; most algorithms, however, only have one-sided error.

Definition 5.9. The class RP is such that $L \in \mathsf{RP}$ if there is a polynomial time deterministic verifier V such that $x \in L \implies \Pr_y[V(x,y)=1] \geqslant \frac{1}{2}$[3] and $x \notin L \implies \Pr_y[V(x,y)=1]=0$. ◇

RP stands for Randomized Polynomial Time.

Observation 5.10. $\mathsf{RP} \subseteq \mathsf{NP}$. ◇

coRP is defined similarly by switching the zero and the $\frac{1}{2}$.

We now define ZPP, or Zero-error Probabilistic Polynomial Time. There are two ways to define this.

Definition 5.11. $\mathsf{ZPP} = \mathsf{RP} \cap \mathsf{coRP}$. ◇

Equivalently,

Definition 5.12. ZPP is the class of languages that have a probabilistic algorithm that outputs an answer with probability $\frac{1}{2}$[4] and if so, the answer is guaranteed to be correct. ◇

Since we can just keep running the algorithm until it outputs, then we can think of ZPP as having expected polynomial runtime (one has to be careful about how to determine expectation).

5.5. BPP vs. P

As previously stated, many researchers now believe that $\mathsf{P} = \mathsf{BPP}$. In fact, there is a theorem that says this under certain assumptions. The assumption is that there is a sufficiently strong "circuit lower bound". We will talk about circuit lower bounds in the next chapter.[5] This is a strange result: that if some things are hard, then other things are easy.

A feature of this result is the following: suppose we prove that $\mathsf{P} = \mathsf{BPP}$. This means that for every problem with a randomized algorithm there exists a deterministic one for the problem; this process is called derandomization. It does not suffice to enumerate over all problems. The idea is to feed in a certain sequence of pseudo-random bits and show that the machine can not tell the difference between pseudo-random and truly random bits.

A later result showed that if $\mathsf{P} = \mathsf{BPP}$ then strong circuit lower bounds exist. This "strong" is not exactly the same as the "sufficiently strong" but they are similar. In fact this statement is even weaker: we only need to show that one problem in BPP is in P, this problem is Polynomial Identity Testing, which is very similar to the Symbolic Determinant Zero-Testing, and in fact uses the same algorithm of picking random values and testing.

Let us revisit the problem of Symbolic Determinant Zero-Testing. Recall that in this problem we have an $n \times n$ matrix whose entries are polynomials of degree d in $\mathbb{Z}_d[x_1, \ldots, x_n]$. Suppose we have a polynomial $P \in \mathbb{Z}_{nd}[x_1, \ldots, x_n]$, $P \neq 0$. Then

[3]In general we can take $\frac{1}{\mathrm{poly}(n)}$ and be okay.

[4]Again, we can take $\frac{1}{\mathrm{poly}(n)}$ and be okay.

[5]For those who already know what circuit lower bounds are, the assumption is as follows: if $L \in \mathsf{DTIME}(2^n)$ then we can compute this in a circuit of similar size. But perhaps we can create a much simpler circuit, but the required assumption is that L requires a circuit of similar size, such as $2^{0.001n}$. In the next chapter we will see that proving statements of this kind is basically hopeless.

$\exists w \in \{1, 2, \ldots, 100nd\}^n$ such that $P(w) \neq 0$. A deterministic algorithm could go over all possibilities of w, but this algorithm takes exponential time. So what is the next thing we could hope for? Maybe we don't need to go over all the points in the space. Maybe there is a polynomial-sized set of points such that the same holds. And this is a fact:

FACT 5.13. *$\exists S \subseteq [1, 2, \ldots, 100nd]^n$, $|S| \leqslant \mathrm{poly}(n, x)$ such that $w \in S \implies P(w) \neq 0$.*

Such a set S is called a pseudo-random set. So if we can find a way of constructing such a set then we are done, but if we prove that we cannot then things become hopeless.

This is closely related to cryptography.

CHAPTER 6

Circuit Complexity

We have seen that just as Turing Machines are a model of computation, circuits are as well. The annoying fact is that circuits are what are known as a "non-uniform" model of computation, whereas Turing Machines are a uniform model. We will see what this means, but what it amounts to is that Turing Machines can work on inputs of all lenghts, whereas a circuit can only work on inputs of fixed length, so on inputs of different lenghts we need different circuits. Then if we need different circuits for different lenghts, then where do they come from? Can they be constructed efficiently?

But we do know that the running time of a Turing Machine is closely related to circuit size, that is, a Turing Machine can be simulated by a circuit of similar size, but not vice versa.

Recall the definition of a circuit.

Definition 6.1. A *circuit* C consists of inputs $x_1, \ldots, x_n \in \{0,1\}^n$ with AND, OR, NOT gates that feed into each other with one final output gate, whose output is the output of the circuit. The *fan-in* is the number of inputs a gate can take; generally we use fan-in $\leqslant 2$. It is important what the fan-in is, as differences occur for larger fan-in. The *fan-out* is the number of gates the output of the gate feeds into, and this is unrestricted. Of course, there are no feedback loops. The circuit C defines a boolean function $C : \{0,1\}^n \to \{0,1\}$. $\diamond$

6.1. Circuit Complexity of a function

Now for a function f there are many circuits that can represent it.

Definition 6.2. Given $f : \{0,1\}^n \to \{0,1\}$, the *circuit complexity* $\mathscr{C}(f)$ is the minimum size (number of gates) of a circuit that computes f. $\diamond$

Fact 6.3. *If f is a function of n variables, $\mathscr{C}(f) \leqslant 2^{O(n)}$.*

Proof. We can always write a function f as

$$f(x_1, \ldots, x_n) = f'(x_1, \ldots, x_{n-1}, 0) \vee f''(x_1, \ldots, x_{n-1}, 1)$$

the OR of two functions of $n-1$ variables (see Figure 6.4). Then we can build recursively.

This roughly doubles for every input bit. $\square$

Another easy but important observation is that almost every function has circuit complexity of roughly 2^n.

Theorem 6.5. *Almost every $f : \{0,1\}^n \to \{0,1\}$ has $\mathscr{C}(f) \geqslant \Omega(\frac{2^n}{n})$.*

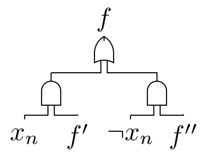


FIGURE 6.4. Recursive decomposition of f

PROOF. The proof is a counting proof. So whatever the number $\Omega(\frac{2^n}{n})$ is, call it s. We will show that the number of functions of size s is small. Well, the number of functions $f : \{0,1\}^n \to \{0,1\}$ is 2^{2^n}, but the number of circuits of size $\leqslant s$ is upper bounded by $(3+(s+n)^2)^s$: there are 3 types of gates, then for each gate two wires going in. Asymptotically, this is $s^{O(s)}$. When $s = 0.01\frac{2^n}{n}$ then $s^s \ll 2^{2^n}$. $\square$

So in this sense almost every function is very hard for circuits. In some sense the tragedy in complexity is that we do not know explicit examples of functions that are hard for circuits.

So the following is an open question:

QUESTION 6.6. Can we find an explicit function $f : \{0,1\}^n \to \{0,1\}$ that has "high" circuit complexity? $\diamond$

The best known result is a circuit of complexity about $4.5n$.

So if we come up with a good enough lower bound, we will prove $\mathsf{P} = \mathsf{BPP}$, so this is some motivation for studying circuits. Another motivation was in fact to try to separate P from NP:

P has polynomial size circuits, and the Karp-Lipton Theorem says that NP does not have polynomial size circuits unless the polynomial hierarchy collapses (so very unlikely). So one way to separate P from NP is to take our favorite problem in NP and show that it cannot be computed by a polynomial size circuit. It is very frustrating that we cannot prove this.

More or less equivalent is the following problem: we have a boolean formula, and we cannot show that it cannot be computed by a polynomial size circuit.

6.2. Circuit Complexity of a Language and $\mathsf{P}_{/\mathsf{poly}}$

Lecture 9, 26 Feb

DEFINITION 6.7. A language L has *circuit complexity* $\leqslant g(n)$ if for all n, any "slice" $\chi_{L,n} : \{0,1\}^n \to \{0,1\}$ with

$$\chi_{L,n}(x) = \begin{cases} 1 & x \in L \\ 0 & x \notin L \end{cases}$$

has $\mathscr{C}(\chi_{L,n}) \leqslant g(n)$. $\diamond$

DEFINITION 6.8. $\mathsf{P}_{/\mathsf{poly}}$ is the class of languages with poly(n) size circuits. $\diamond$

FACT 6.9. $\mathsf{P} \subseteq \mathsf{P}_{/\mathsf{poly}}$.

As an easy proof, take $L \in \mathsf{P}$, decided by a polynomial time DTM M. Then for all x, $|x| = n$, then $x \in L \iff M(x) = 1 \iff C_{M,n}(x) = 1$, where $C_{M,n}$ is a circuit of size $\leqslant$ poly(n). In fact, $C_{M,n}$ can be constructed by a Turing Machine in poly(n) time.

As far as the definition of $\mathsf{P}_{/\mathsf{poly}}$ is concerned, these circuits do not have to be computed efficiently. As a result, the class $\mathsf{P}_{/\mathsf{poly}}$ is very strange. Morally, $\mathsf{P}_{/\mathsf{poly}}$ should be the same as P but not technically, due to this feature:

FACT 6.10. $\mathsf{P}_{/\mathsf{poly}}$ *contains undecidable languages.*

PROOF. Let L be any undecidable language over $\{0,1\}$. The claim is that we can define a unary language $L' = \{0^n \mid \text{binary representation of } n \text{ is in } L\}$. L' is undecidable, but since it is a unary language it trivially has polynomial size circuits, as the slice

$$\chi_{L',n}(x) = \begin{cases} \text{accept} & x = 0^n \text{ and } x \in L' \\ \text{reject} & \text{otherwise} \end{cases}$$

□

This issue is what is known as "non-uniformity", as opposed to the Turing Machine, which is a uniform model.

FACT 6.11. *Suppose* $L \in \mathsf{P}_{/\mathsf{poly}}$, *decided by* $\{C_n\}_{n=1}^{\infty}$ *and* C_n *can be constructed in* $\text{poly}(n)$ *time. Then* $L \in \mathsf{P}$.

This is kind of a triviality.

This notation can be generalized:

DEFINITION 6.12. Let $\mathscr{C}$ be a complexity class defined by some computational model, then the model with advice $\mathscr{C}_{/f(n)}$ is $\mathscr{C}$ with some "advice" of size $f(n)$ fixed for each n and each L. ◇

So for $\mathsf{P}_{/\mathsf{poly}}$ the "advice" is the circuit C_n. Indeed if $L \in \mathsf{P}_{/\mathsf{poly}}$ then L is computed by a DTM in polynomial time with polynomial size advice.

THEOREM 6.13 (Karp-Lipton). *If* $\mathsf{NP} \subset \mathsf{P}_{/\mathsf{poly}}$ *then* $\mathsf{PH} = \Sigma_2$.

PROOF. To show this it suffices to show that $\Sigma_3 = \Sigma_2$.

To start the proof, it is important what language we pick. In this case the correct language is SAT. So assume SAT can be solved by polynomial size circuits. This means that there exist circuits $\{C_m\}_{m=1}^{\infty}$ such that for input the encoding of a SAT formula ϕ, the output is 1 if ϕ is satisfiable, and 0 if ϕ is unsatisfiable.

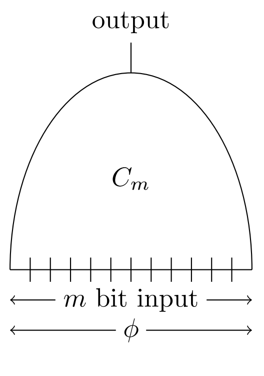


FIGURE 6.14. The circuit C_m

The reason we pick SAT is because it has the nice feature of being *downward self-reducible*. What this means is the following: Suppose we have the formula

$\phi(x_1, \ldots, x_n)$. Then deciding if ϕ is satisfiable reduces to deciding if $\phi|_{x_1=0}$ or $\phi|_{x_1=1}$ are satisfiable.[1]

The idea will be to take $L \in \Sigma_3$; there is a polynomial time deterministic verifier V such that

$$x \in L \iff \exists y \forall z \exists w, V(x, y, z, w) = 1 \iff \exists y \forall z, (\phi_{x,y,z}(w) \in \mathsf{SAT})$$

Note that we have this circuit family $\{C_m\}_{m=1}^{\infty}$ that decides SAT. Then a Σ_2 machine can "find" $\{C_i\}_{i=1}^{m}$ in poly(m) time: Look at polynomial size circuits $C_m, C_{m-1}, \ldots, C_1$; these circuits have the property that taking any ϕ, $|\phi| = i$, then $|\phi|_{x_1=0}| = j$, $|\phi|_{x_1=1}| = k$, $j, k < 1$. Then $C_i(\phi) = C_j(\phi|_{x_1=0}) \vee C_k(\phi|_{x_1=1})$. This holds for every ϕ of length $\leqslant n$, with the base condition that C_1 is correct, which is trivial. Then circuits that satisfy this property are also circuits deciding SAT. Then the machine has two quantifiers and will solve the rest on its own.

Formally,

$$\begin{aligned} x \in L \quad &\iff \quad \exists y \forall z (\phi_{x,y,z} \in \mathsf{SAT}) \\ &\iff \quad \exists y, C_m, \ldots, C_1, \forall z, \phi, |\phi| \leqslant m, \\ &\qquad\qquad C_m(\phi_{x,y,z}) = 1 \wedge (C_i(\phi) = C_j(\phi|_{x_1=0}) \vee C_k(\phi_{x_1=1})) \\ &\qquad\qquad\qquad \wedge C_1(T) = 1 \wedge C_1(F) = 0 \end{aligned}$$

So $\Sigma_3 \subseteq \Sigma_2$. □

This is why we believe that circuit complexity is a good approach to separating P vs NP.

6.3. Towards $\mathsf{P} \subsetneq \mathsf{NP}$

Fact 6.15. $\mathsf{CLIQUE} = \{G \mid |G| = n, G$ *has clique of size* $\lfloor n^{1/4} \rfloor\}$ *is* NP-*complete.*

One way to show that $\mathsf{P} \subsetneq \mathsf{NP}$ is to show that this problem does not have a polynomial size circuit. Note that a graph G corresponds to a bit string in $\{0,1\}^{\binom{n}{2}}$, where each bit can represent whether that edge exists or not. Then for C_n, $|C_n| \leqslant \text{poly}(n)$, then

$$C_n(G) = \begin{cases} 1 & G \text{ has } \lfloor n^{1/4} \rfloor\text{-clique} \\ 0 & \text{otherwise} \end{cases}$$

This computes a function $f : \{0,1\}^{\binom{n}{2}} \to \{0,1\}$.

The task is to show that this is not possible.

Definition 6.16. A function $f : \{0,1\}^n \to \{0,1\}$ is called *monotone* if $f(x) = 1 \implies f(y) = 1$ if y is obtained from x by switching a coordinate from 0 to 1. ◇

Fact 6.17. *If f is monotone then f is computed by a circuit with only* AND *and* OR *gates, and no* NOT *gates. These are called* monotone circuits.

This can be seen by viewing a monotone function as an AND of inclusions of min-terms.

Note that the CLIQUE function is a monotone function.

[1]There are other problems that are downward self-reducible, and any of these would also work for the proof.

DEFINITION 6.18. The *monotone complexity* $\mathscr{M}(f)$ is the minimum size of a monotone circuit computing f. ◇

THEOREM 6.19 (Razborov). $\mathscr{M}(\mathsf{CLIQUE}) \geqslant 2^{n^{1/8}}$

So it seemed pretty close that we would achieve the desired result if we can prove lower bounds on monotone functions, but Razborov then dashed the hopes.

THEOREM 6.20. $\mathscr{M}$(*Perfect Matching in Bipartite Graphs*) $\geqslant n^{\log n}$.

We see that Perfect Matching is a monotone function. The result is because Perfect Matching is in P.

6.4. Circuit Depth, the class NC, and Parallel Computing

The depth of a circuit is how many successive layers we need to compute it.

DEFINITION 6.21. For a circuit C, the layer 0 is the inputs, then layer 1 is all gates that only have inputs from layer 0, then layer 2 is all gates that have inputs from layer 0 or layer 1, and so on. Then if the output is at layer k then k is the *depth* of the circuit. ◇

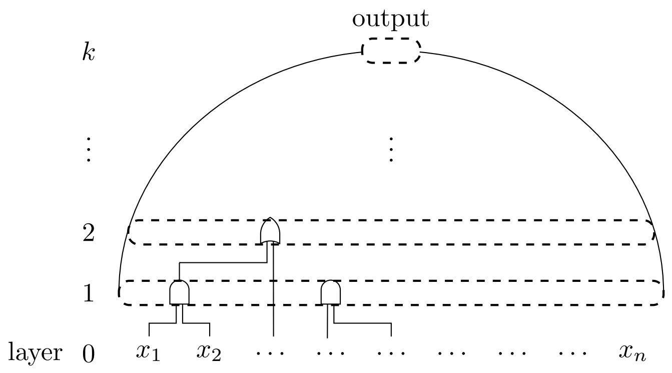


FIGURE 6.22. Layers of a circuit

The model of parallel computation is as follows: at every step the processors can perform a step, then continue to the next step depending on the result of the previous step. So we can think of a circuit of depth k as a parallel computation that completes in time k.

Now we said that a circuit is efficient if it has size poly(n), so when talking about depth, it is good to think about circuits of depth poly(log(n)).

DEFINITION 6.23. NC or Nick's Class is a family of classes $\mathsf{NC}^1 \subseteq \mathsf{NC}^2 \subseteq \ldots$, where NC^i is the class of languages L that are computed by circuits $\{C_n\}_{n=1}^{\infty}$ such that $|C_n| \leqslant \text{poly}(n)$, $\text{depth}(C_n) \leqslant O((\log n)^i)$, and the fan-in is $\leqslant 2$. ◇

Notice that the fan-in matters, for if we allow unbounded fan-in, many steps of computation in the bounded fan-in model would only take one step. It matters most that the fan-in is bounded.

This is strictly the definition of non-uniform NC^i.

Definition 6.24. Uniform NC^i is such that there is a $\log(n)$-space DTM that generates C_n. ◇

Here we mean that the DTM outputs a description of the circuit C_n. Lecture 10, 3 Mar

Definition 6.25. A *description of a circuit* are tuples $(\ell, \mathsf{TYPE}, j, k)$ where ℓ, j, k are indices of gates, $\mathsf{TYPE} \in \{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}\}$, and the input of ℓ are outputs of j, k. ◇

There is a related class AC.

Definition 6.26. $L \in \mathsf{AC}^i$ if there is a circuit C_n such that $|C_n| \leqslant \text{poly}(n)$, $\text{depth}(C_n) \leqslant O((\log n)^i)$, and gates can have unbounded fan-in. ◇

Note that $\mathsf{AC}^0 \subseteq \mathsf{NC}^1 \subseteq \mathsf{AC}^1 \subseteq \mathsf{NC}^2 \subseteq \mathsf{AC}^2 \subseteq \ldots$, since we can move unbounded fan-in to bounded fan-in by increasing the depth by a factor of $O(\log n)$.

Let us now look at some examples of interesting problems in these classes.

The following example separates AC^0 from NC^1.

Example 6.27. $\mathsf{PARITY} \in \mathsf{NC}^1$, but $\mathsf{PARITY} \notin \mathsf{AC}^0$. ◇

The reason $\mathsf{PARITY} \notin \mathsf{AC}^0$ can be seen by the following: fix some of the bits in an insance of PARITY to fixed values. The result is still PARITY. The problem does not simplify in some sense. But for a circuit of constant depth, the circuit does simplify.

Example 6.28. Boolean Matrix Multiplication is in NC^1. The problem is defined as follows: Given A, B $n \times n$ 0-1 matrices, $(AB)_{ij} = \bigvee_{k=1}^{n}(A_{ik} \wedge B_{kj})$. ◇

The idea is that all of these pairwise ANDs can be done in $\log n$ depth, and all of the ORs can be done in $\log n$ depth as well.

Example 6.29. $\mathsf{ST{-}CONN}$ is in NC^2. ◇

We already know that $\mathsf{ST{-}CONN}$ is NL-complete.

Proof. If $|G| = n$ then define the matrix M by

$$M_{ij} = \begin{cases} 1 & i = j \text{ or } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Then

$$M_{ij}^{\ell} = \begin{cases} 1 & \exists i \rightsquigarrow j \text{ path of length } \leqslant \ell \\ 0 & \text{otherwise} \end{cases}$$

Then $\langle G, s, t\rangle \in \mathsf{ST{-}CONN} \iff \exists s \rightsquigarrow t \text{ path of length } \leqslant n \iff M_{st}^{n} = 1$.

Well we can solve this using Boolean Matrix Multiplication in depth $\leqslant (\log n)^2$ via the sequence $M, M^2, M^4, \ldots$. □

Theorem 6.30. $\mathsf{NC} \subseteq \mathsf{P}$. *Furthermore,* $\mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{NC}^2$ *and more generally* $\mathsf{NC}^i \subseteq \mathsf{DSPACE}((\log n)^i)$.

Proof. To show that $\mathsf{NC} \subseteq \mathsf{P}$, for input x, $|x| = n$, we can just generate C_n for language L and run C_n on x.

We already know $\mathsf{NL} \subseteq \mathsf{NC}^2$.

Next, for $L \in \mathsf{NC}^i$, Figure 6.31 is a $(\log n)^i$-space algorithm for L:

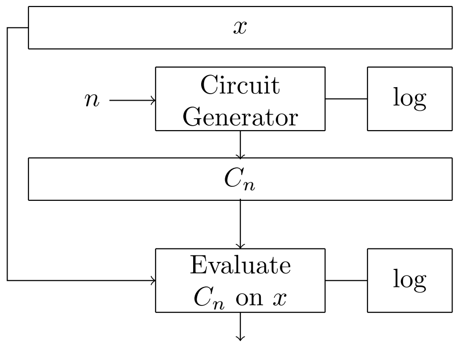


Figure 6.31. $(\log n)^i$-space algorithm for deciding a language in NC^i.

The problem is that when computing in space-bounded computation, that we cannot just compute the entire circuit, since the circuit takes polynomial space. So we have to be careful. The idea is that we only generate the gates that we need.

So we define a function COMPUTE so that COMPUTE(ℓ) generates a tuple $(\ell, \mathsf{TYPE}, j, k)$ (throwing out generated circuits if they are not the correct one), then set $a = \text{COMPUTE}(j)$ and $b = \text{COMPUTE}(k)$. Then depending on the output, output $a \wedge b$, $a \vee b$, or $\neg a$.

Now this is a recursive algorithm, so the space required is the recursion depth times the amount of data needed to store the information at each level. So the recursion depth is $(\log n)^i$, and at each level we need to store pointers j, k, ℓ. $O(\log n)$. So the space required is $\leqslant O((\log n)^{i+1})$, which is a problem. So we will use a little trick: if we unroll the entire recursion, what the algorithm is remembering is some path from the output down to an input. Now we observe that we do not really need to know the index of the gate; we only need to know if it is the left or right child of its parent to be able to backtrack through the path. Then we only need to remember one bit: left vs. right. So the space required is $O((\log n)^i)$. □

We know that $\mathsf{NC}^i \subseteq \mathsf{P}$. That $\mathsf{NC}^i \subsetneq \mathsf{P}$ would mean that there are problems in P that cannot be efficiently parallelized. Intuitively this makes sense, since there are problems that are inherently sequential. However, we do not know if this is true. The following is known, however:

Theorem 6.32 (Mulmuley). *In the restricted setting where an integer is a unit and not composed of bits, the max-flow problem is* P *but not* NC.

CHAPTER 7

Interactive Proofs

The basic idea is the following: we have a miserable, time-bounded verifier. But we have a guy who is very powerful but evil, say, perhaps the devil. Now the devil has something of convincing you of, and you want to verify something as well. In a more realistic setting, a software company wants to sell you some software, and you want to verify that it works. So in both settings we have some other party that is in some way malicious. So the idea is, can we interact with this malicious other party and still verify what we want to verify?

We need to ideas: completeness and soundess. Completeness says that if the software is good, then the company can convince you of it. Soundness says that if the software is faulty, there is no way that the company can convince you that it is good.

There are a few motivating factors:

- By definition a large chunk of what occurs in cryptography are interactive proofs.
- We can also prove statements of the following kind: Graph Isomorphism is unlikely to be NP-complete. So this is a candidate for a language in NP but not NP-complete.
- Interaction by itself is just NP, and Randomization by itself is just BPP, but Interaction + Randomization gives interesting results, which is exactly the class of Interactive Proofs.

7.1. A First Attempt: Deterministic Interactive Proofs

Consider a verifier V and a prover[1] P and a language L. The prover has incredible interest in convincing the verifier that $x \in L$, and the verifier is interested in verifying the truth. Now L may be some very difficult language so the verifier cannot verify on its own, so it can ask questions to the prover, which can answer honestly or it can cheat. At the end, the verifier should output "yes" or "no".

So formally, we have the following:

Say V asks a question $q_1 = f_1(x)$, then P responds with $a_1 = g_1(x, q_1)$. Then V asks $q_2 = f_2(x, q_1, a_1)$, and P responds with $a_2 = g_2(x, q_1, a_1, a_2)$, and so on. In the last round V asks q_k and P responds with a_k. In general

$$q_i = f_i(x, q_1, a_1, \ldots, q_{i-1}, a_{i-1})$$
$$a_i = g_i(x, q_1, a_1, \ldots, q_{i-1}, a_{i-1}, q_i)$$

Then V looks at the transcript of all of the interactions, and accepts if and only if some verification process $\tilde{V}(x, q_1, a_1, \ldots, q_k, a_k) = 1$.

[1]In our analogy, the prover was the devil.

This looks perfectly fine. We have to restrict the verifier to polynomial time, or else if the verifier were too powerful we would not require a prover. So we require $f_1, f_2, \ldots, f_k, \tilde{V}$ to be polynomial time computable, but the functions $g_1, g_2, \ldots, g_k$ are arbitrary[2]. Furthermore, the number of rounds $2k$ is $\leqslant \text{poly}(n)$.

Now we want to state what we expect from this protocol $f_1, f_2, \ldots, f_k, \tilde{V}$.

We have the ideas of *completeness* and *soundness*. Let $V^P(\cdot)$ be the result of interaction of V with P. Then completeness says that $x \in L \implies \exists P, V^P(x) = 1$, otherwise the protocol is useless. Soundness says that $x \notin L \implies \forall P, V^P(x) = 0$.

Then the question is, fixing a language L, does L have an "interactive proof"? That is, does it have a "protocol" conforming to these properties?

So formally the interactive proof is just the sequence $f_1, f_2, \ldots, f_k, \tilde{V}$. The prover P is the sequence $g_1, g_2, \ldots, g_k$.

FACT 7.1. *L has a deterministic interactive proof if and only if $L \in \mathsf{NP}$.*

PROOF. If $L \in \mathsf{NP}$ then on input x, then V can send a dummy question and P responds with a certificate that $x \in L$ (if not, P can send garbage). Then V just verifies the certificate. Then completeness and soundness follow trivially.

Now if L has a deterministic interactive proof, then given this protocol we can create a one-round protocol simulating it in which the prover only sends only one answer. Of course since P can be all-powerful, P knows what questions V will ask, and thus can just send the entire transcript. Then V just verifies that the transcript is correct corresponding to if we had actually run the original protocol. □

So we have not gone very far.

7.2. Interactive Proofs with Randomization

We saw that the set of languages with Deterministic Interactive Proofs was just NP. Now if we have randomness by itself, we get BPP. However if we combine them to get the class IP, we will eventually see that IP is exactly PSPACE.

Before we give the formal definition of IP, let us look at a protocol for Graph Isomorphism.

So let L be Graph Non-Isomorphism, that is $\mathsf{GNI} = \{\langle G, H\rangle \mid |G| = |H| = n, G \ncong H\}$. Now this is in coNP and is possibly not in NP.

THEOREM 7.2. *GNI has an Interactive Proof.*

PROOF. Randomly pick either G or H and randomly permute the graph. Then ask if and accept if and only if the graph is isomorphic to the graph chosen. Then if they are not isomorphic, then we accept with probability 1, otherwise, if they are isomorphic we accept with probability $\frac{1}{2}$. In this case $\frac{1}{2}$ is the error bound.

Formally, we pick a bit $b \in \{0, 1\}$ and a permutation $\pi : [n] \to [n]$ at random. The question is $F = \begin{cases} \pi(G) & b = 0 \\ \pi(H) & b = 1 \end{cases}$. Then P responds with a bit $b' \in \{0, 1\}$, and V accepts if and only if $b = b'$. Here b denotes if the graph is isomorphic to G or H, and b' denotes if the graph is isomorphic to G or H is P's opinion.

We need to verify completeness and soundness. For completeness, if G, H are not isomorphic, then there exists P such that $\Pr[V^P \text{ accepts}] = 1$. Now in this case

[2]This captures the notion that P is powerful. We can also study models where the prover is limited.

P is a map from n-vertex graphs to a bit in $\{0,1\}$. So the desired prover is the one mapping the graph to 0 if it is isomorphic to G, and 1 if it is isomorphic to H (in the case that neither is true the prover can just map to garbage). For soundness, if G, H are isomorphic then for all P, $\Pr[V^P \text{ accepts}] = \frac{1}{2}$. □

Now there are issues in which what happens when the prover returns an answer that is not of the type expected, but in such a case the verifier can just reject. Then this does not affect soundness since we are making the verifier more strict, and does not affect completeness since it does not change that there is one prover that responds correctly.

Note that in this case the randomness was private to the verifier, and that the prover cannot see it. This is called *private randomness* (or *private coins*). This is essential to the protocol: if the prover could see the coin then this does not work. Lecture 11, 5 Mar

Now notice that the error probability bound of $\frac{1}{2}$ can be reduced by taking sequential k-wise repetition and then the error probability becomes $\leqslant \frac{1}{2^k}$. Now this changes the number of rounds from 2 to $2k$, but we can get it back down to 2 using what is known as *parallel repetition*. Remember that the verifier generated (b, π) and sent a graph F and accepted if and only if the prover returns b' with $b' = b$. Now instead we can just generate $(b_1, \pi_1), \dots, (b_k, \pi_k)$ and send graphs $F_1, \dots, F_k$ and accept if and only if the prover returns $b'_1, \dots, b'_k$ with $b_1 = b'_1 \wedge \dots \wedge b_k = b'_k$.

It is not difficult to prove that this has error probability $\leqslant \frac{1}{2^k}$, but there is a technical point: now the prover is a function $g : (F_1, \dots, F_k) \to (b'_1, \dots, b'_k)$. This strategy could be rather complicated: it does not need to be a simple product of pairwise functions. But we can prove it using by conditioning.

We now formally define the class of interactive proofs with private randomness.

Definition 7.3. In an *interactive proof with private randomness* there is a verifier V and a prover P such that V picks private randomness r_1 and sends $q_1 = f_1(x, r_1)$ and P responds with $a_1 = g_1(x, q_1)$, then V picks new private randomness r_2 and sends $q_2 = f_2(x, r_1, q_1, a_1, r_2)$[3] and P responds with $a_2 = g_2(x, q_1, a_1, a_2)$, and so on for up to q_k and a_k. Then there is a verification process $\tilde{V}$ where we pick randomness r_{k+1} and accept if and only if $\tilde{V}(x, (r_i, q_i, a_i)_{i=1}^k, r_{k+1}) = 1$.

The verifier V, that is, $f_1, \dots, f_k, \tilde{V}$ should be polynomial time and deterministic, and the prover P, that is, $g_1, \dots, g_k$ are arbitrary. Of course $|q_i|, |a_i|, |r_i|, k$ are all $\leqslant \mathrm{poly}(n)$.

Furthermore, the following hold:

- Completeness: $x \in L \implies \exists P = (g_1, \dots, g_k) \Pr_{r_1, \dots, r_{k+1}}[V^P(x) = 1] \geqslant \frac{2}{3}$
- Soundness: $x \notin L \implies \forall P, \Pr_{r_1, \dots, r_{k+1}}[V^P(x) = 1] \leqslant \frac{1}{3}$

◇

Definition 7.4. IP is the class of languages L that have a private-coin interactive proof. ◇

There is a related class that allows public-coin proofs:

Definition 7.5. AM (Arthur-Merlin) is the class of languages L that have a public-coin interactive proof. Arthur is the verifier and Merlin is the prover. ◇

[3]Now note that some of these are not necessary, as q_1 depends on x and r_1 and generally r_2 does not depend on r_1.

There is an idea of a randomized prover, but we can show that the optimal prover is one that is deterministic.

It is useful to study interactive proofs where the number of rounds is constant:

Definition 7.6. $\mathsf{IP}[t]$ is the class of languages L with (private-coin) interactive proofs with t rounds, $t = O(1)$. $\diamond$

We can also study $\mathsf{IP}[\log]$ and $\mathsf{IP}[\text{poly}]$.

As it turns out, having private coins and public coins is basically the same, which is surprising. In particular, for the protocol for GNI we can do some massaging and have public randomness and it still works without increasing the number of rounds. That is, basically $\mathsf{IP} = \mathsf{AM}$, but before they were separate classes that took a lot of work to prove are identical.

Theorem 7.7. $\mathsf{IP} = \mathsf{AM}$

In fact, this preserves the number of rounds, as well as which side the error is on.

7.3. #SAT and Sum-Check Protocols

Let us give an explicit public-coin protocol, for the problem #SAT,, which is counting the number of satisfying assignments to a SAT formula. This is not a decision problem, but we can create one:

Definition 7.8. $L_{\#\mathsf{SAT}} = \{\langle \phi, k\rangle \mid \phi$ is a SAT formula with k satisfying assignments$\}$. $\diamond$

Clearly this language is at least as difficult as NP since NP asks if ϕ have at least one satisfying assignment. For the same reason it is also at least as difficult as coNP since coNP asks if all assignments satisfy ϕ. But in fact $L_{\#\mathsf{SAT}}$ can solve all of PH. Formally, the class is called $\mathsf{P}^{\#\mathsf{SAT}}$. Now it is easy to show that $\mathsf{P}^{\#\mathsf{SAT}} \subseteq \mathsf{PSPACE}$.

It is easy to show that $\mathsf{IP} \subseteq \mathsf{PSPACE}$ since in polynomial space we can construct the best strategy for the prover, and then figure out the acceptance probability of this best prover.

We will show that $\mathsf{P}^{\#\mathsf{SAT}} \subseteq \mathsf{IP} \subseteq \mathsf{AM}$, using what is known as a sum-check protocol. This technique will also be used to show that in fact $\mathsf{IP} = \mathsf{PSPACE}$. These protocols are also used in probabilistic checkable proofs.

The protocol for #SAT starts with the following observation:

Fact 7.9. *Take a* $3\mathsf{-SAT}$ *formula* $\phi = C_1 \wedge \ldots \wedge C_m$ *over variables* $x_1, \ldots, x_n$. *There is a polynomial* $g \in \mathbb{Z}_d[x_1, \ldots, x_n]$ *where the degree is* $d \leqslant 3m$, *such that* $\forall x \in \{0,1\}^n$,

$$g(x) = \begin{cases} 1 & x \textit{ satisfies } \phi \\ 0 & x \textit{ does not satisfy } \phi \end{cases}$$

So g *is equivalent to* ϕ *on binary values. But for* $\alpha_1, \ldots, \alpha_n \in \mathbb{Z}$, $g(\alpha_1, \ldots, \alpha_n)$ *can be computed in* $\text{poly}(n)$ *time.*

Proof. Take the product over the clauses, that is

$$g = \Pi_{i=1}^m (1 - (1 - x_{i_1}) x_{i_2} (1 - x_{i_3}))$$

. This is a polynomial of degree 3. $\square$

Now we proceed with the protocol for #SAT. We move from SAT to 3$-\mathsf{SAT}$ but this okay since we can move from one to the other.

Proof that #SAT has a protocol. The prover is trying to convince the verifier that given a 3$-\mathsf{SAT}$ formula ϕ, the number of satisfying assignments to ϕ is k. This is the same as saying the prover is trying to covince the verifier that $k = \sum_{x \in \{0,1\}^n} g(x)$ since $g|_{\{0,1\}^n} = \phi$. This is why the protocol is called the *sum-check protocol*. What the protocol will do is to evaluate this on unit hypercube and reduce the dimension iteratively until the dimension equals 0.

More formally, we have $k = \sum_{x_1 \in \{0,1\}} \sum_{x_2 \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} g(x_1, x_2, \ldots, x_n)$ which can be viewed as $\sum_{x_1 \in \{0,1\}} h_1(x_1)$ by viewing the values $x_2, \ldots, x_n$ as fixed. Now $g \in \mathbb{Z}_d[x_1, \ldots, x_n]$ and then h_1 is a degree d polynomial in x_1. But the verifier does not know what h_1 is.

So the verifier asks the prover to give $h_1(x_1)$. Now the verifier has to be on guard against the possibility that the prover could send garbage, so the verifier needs to check if $k = h_1(0) + h_1(1)$. Now this does not suffice, since the prover could send $h_1' \neq h_1$ such that $k = h_1'(0) + h_1'(1)$. The way to guard against this is that if two polynomials are different, they are different almost everywhere, so if we pick values over a large range, they should be different. So the verifier picks random $\alpha_1 \in [1, 2, \ldots, R]$ where R is something large like $n^{100}d^{100}$, and computes $T_1 = h_1(\alpha_1)$. Well T_1 is a function over a hypercube of one dimension less, that is, $T_1 = \sum_{x_2 \in \{0,1\}} \cdots \sum_{x_n \in \{0,1\}} g(\alpha_1, x_2, \ldots, x_n)$. We view g as a polynomial of degree d over a hypercube of dimension $n-1$.

We then continue in this manner, picking random $\alpha_2, \alpha_2, \ldots$. Eventually we will pick a random α_n and the prover has returned a function $h_n(x_n)$, and we plug in α_n and see that $h_n(\alpha_n)$ is supposed to be $g(\alpha_1, \alpha_2, \ldots, \alpha_n)$. So we need to verify this equality, which we can do since we know what g is. Then the verifier accepts if and only if $h_n(\alpha_n) = g(\alpha_1, \ldots, \alpha_n)$.

So suppose $k = \sum_{x \in \{0,1\}^n} g(x)$, an honest prover will give back $h_1, \ldots, h_n$ as we ask of it, and we accept with probability 1. This gives us completeness.

If $k \neq \sum_{x \in \{0,1\}^n} g(x)$, then the verifier accepts with probability $\leqslant O(\frac{dn}{R})$, which is negligible. The proof of soundness relies on the following: say that there is a liar: as soon as the liar tells a lie, the liar must continue to lie again and again, and eventually will get caught. Concretely, $k \neq \sum_{x_1 \in \{0,1\}} h_1(x_1)$, but in order to have the verifier continue, the prover must lie and give some $h_1^* \neq h_1$, but $h_1^*(\alpha) \neq h_1(\alpha_1)$ with probability $1 - O(\frac{d}{R})$. This continues over smaller and smaller subcubes, and at the end the liar must lie about h_n, that is the value of g itself. So the liar will get caught. Applying the Schwarz-Lippel lemma we get the bound $\leqslant O(\frac{dn}{R})$. $\square$

This protocol is a specific case of a more general statement:

Lecture 14, 24 Mar

Theorem 7.10. *Suppose* $g(x_1, \ldots, x_n) \in \mathbb{Z}_d[x_1, \ldots, x_n]$, *where* $d = \mathrm{poly}(n)$ *and* g *can be evaluated in polynomial time in* n. *The prover tries to convince the verifier* $k = \sum_{x \in \{0,1\}^n} g(x)$. *There is a sum-check protocol that in* $2n$ *rounds, if* $k = \sum_{x \in \{0,1\}^n} g(x)$, *then there is a prover such that* $\Pr[\textit{accept}] = 1$, *but if* $k \neq \sum_{x \in \{0,1\}^n} g(x)$, *then for any prover* $\Pr[\textit{accept}] \leqslant O(\frac{nd}{R})$ *where* R *is a verifier. Then the verifier only needs to compute the value* $g(\alpha_1, \ldots, \alpha_n)$ *for* α_i *randomly picked from* $[1, 2, \ldots, R]$.

7.4. AM vs. PSPACE

Recall that $\mathsf{IP} = \mathsf{PSPACE}$. We now prove the following:

THEOREM 7.11. $\mathsf{AM} = \mathsf{PSPACE}$.

Let us look at where we currently are in the hierarchy of complexity: $\mathsf{P} \subseteq \mathsf{NP} \subseteq \Sigma_2 \subseteq \ldots \subseteq \mathsf{PH} \subseteq \#\mathsf{P} \subseteq \ PSPACE$, where $\#\mathsf{P}$ is the class of $\#\mathsf{SAT}$. We already showed an interactive proof for $\#\mathsf{P}$. This is obviously contained in PSPACE since in polynomial space we can just count the number of satisfying assignments.

PROOF. The trivial direction is that $\mathsf{AM} \subseteq \mathsf{PSPACE}$. We only need two rounds, we ask a question with randomness r_1, then the prover returns a_1, and with randomness r_2 then accept if and only if $V(x, r_1, a_1, r_2) = 1$. Well in polynomial space, we can compute the probability for every possible answer, so we can compute the optimal answer to send: we can just compute the entire tree in polynomial space and search through it.

Now for $L \in \mathsf{PSPACE}$ we construct a public coin protocol for L. This protocol will use the sum-check protocol as a subroutine.

There is a DTM M that decides L in space $\ell = \text{poly}(n)$ and time 2^m where $m = \text{poly}(n)$, where $n = |x|$. Without loss of generality we can assume that it takes 2^m steps, enumerating through 2^m configurations, starting with $C_0 = C_{\text{start}}$. The question of $x \in L$ is the same as that $C_{2^m} = C_{\text{accept}}$.

Now the prover tries to convince the verifier that we can go from C_{start} to C_{accept} in 2^m steps, which we write $C_{\text{start}} \xrightarrow{2^m} C_{\text{accept}}$. So we encode this as boolean formulae, then comvert them to polynomials, and then we will run several rounds of sum-check protocols.

So let us see what these functions are.

For $j = 0, 1, 2, \ldots, m$, let $f_{2^j} : \{0,1\}^\ell \times \{0,1\}^\ell \to \{0,1\}$, where we identify ℓ-bit strings with configurations, and $f_{2^j}(X, Y) = \begin{cases} 1 & X \xrightarrow{2^j} Y \\ 0 & \text{otherwise} \end{cases}$. Now observe that $f_{2^{j+1}}(X, Y) = \bigvee_{Z \in \{0,1\}^\ell} (f_{2^j}(X, Z) \wedge f_{2^j}(Z, Y))$, that is, there is some intermediate configuration Z such that $X \xrightarrow{2^j} Z$ and $Z \xrightarrow{2^j} Y$. Furthermore, if such a Z, if it exists, is unique since the machine is deterministic.

So now the prover tries to convince the verifier that $1 = f_{2^m}(C_{\text{start}}, C_{\text{accept}})$.

For $j = 0$, $f_1(X, Y)$ is a boolean formula of polynomial size. Then there exists $g_1(X, Y) \in \mathbb{Z}_d[2^\ell$ variables$]$ of degree d in each variable that is equivalent to f_1 on boolean inputs, that is $g_1(X, Y) = f_1(X, Y)$ if $X, Y \in \{0,1\}^\ell$, but we can plug integral values into it, just as we did in the protocol for $\#\mathsf{SAT}$. Now eventually if we just explicitly write out the corresponding polynomials $g_{2^{j+1}}$ by mimicking the formula for $f_{2^{j+1}}$ we will get an exponential-sized formula, which we do not want. But we can write $g_{2^{j+1}}(X, Y) = \sum_{Z \in \{0,1\}^\ell} g_{2^j}(X, Z) g_{2^j}(Z, Y)$. Now in general this summation might not work for an exact conversion on boolean inputs, but in this case since Z is unique it does.

So now we can say that the prover tries to convince the verifier that $1 = g_{2^m}(C_{\text{start}}, C_{\text{accept}})$. Now we will show that $* \overset{?}{=} g_{2^m}(*, *)$ can be reduced to an identical question, but for a polynomial of lower order, $* \overset{?}{=} g_{2^{m-1}}(*, *)$, then in the end, to $* \overset{?}{=} g_1(*,*)$, where we can compute the value of $g_1(*,*)$ on its own in polynomial time.

So what we want to prove is: given K, Γ, Θ, $K \in \mathbb{Z}$, $\Gamma, \Theta \in \mathbb{Z}^\ell$, there is a protocol that computes K', Γ', Θ', $K' \in \mathbb{Z}$, $\Gamma', \Theta' \in \mathbb{Z}^\ell$, such that if $K = g_{2^{j+1}}(\Gamma, \Theta)$ then $K' = g_{2^j}(\Gamma', \Theta')$ and if $K \neq g_{2^{j+1}}(\Gamma, \Theta)$ then the protocol rejects, or $K' \neq g_{2^j}(\Gamma, \Theta)$ with high probability.

Now $K \overset{?}{=} g_{2^{j+1}}(\Gamma, \Theta) = \Sigma_{Z\in\{0,1\}^\ell} g_{2^j}(\Gamma, Z) g_{2^j}(Z, \Theta) = \sum_{Z\in\{0,1\}^\ell} h(Z)$, where $h(Z)$ has double the degree of g_{2^j}. Then we can run sum-check, and in the end, the verifier checks if some concrete value is equal to this polynomial at some concrete point, $* = h(\alpha)$. Now where does the verifier get the value of h at a certain input? Well, $h(\alpha)$ by definition is $g_{2^j}(\Gamma, \alpha) g_{2^j}(\alpha, \Theta)$. Now if the verifier had these two values, then it would be in good shape. So we can attempt to ask the prover for the value $K'' = g_{2^j}(\Gamma, \alpha)$ and $K''' = g_{2^j}(\alpha, \Theta)$, then check if $h(\alpha) = K''K'''$. Now we further need to be convinced that these are indeed true. But the issue is now we have two questions of the form we wanted, instead of just one. So we need to bring down the two questions to one question.

The trick is to use two values $g_{2^j}(\Gamma, \alpha)$ and $g_{2^j}(\alpha, \Theta)$. Now observe that the restriction of a global polynomial to a line through two points is a polynomial in one variable, the variable that parametrizes the line. So we just ask the prover for a function of degree d in one variable through this line that goe through (Γ, α) and (α, Θ). But the prover could cheat, so we could just pick a random point on the line, and verify some condition on the global function on that value. So we restrict to $\Psi(t) = g_{2^j}(t\Gamma + (1-t)\alpha, t\alpha + (1-t)\Theta)$ of degree $O(d)$. So $g_{2^j}(\Gamma, \alpha) = \Psi(1)$ and $g_{2^j}(\alpha, \Theta) = \Psi(0)$.

Now we want to make sure that the Ψ that the prover gives is the correct one, so we check this at some random point and use the Schwarz-Lippel lemma. So we pick random t^* and we want to verify $\Phi(t^*) = g_{2^j}(t^*\Gamma + (1-t^*)\alpha, t^*\alpha + (1-t^*)\Theta)$. Then $\Phi(t^*) = K'$ and then $t^*\Gamma + (1-t^*)\alpha = \Gamma'$ and $t^*\alpha + (1-t^*)\Theta = \Theta'$.

Then we just continue until we get down to $* = g_1(*,*)$ and verify.

This is a public coin protocol, but it is important to not reveal all the randomness ahead of time. We have to reveal the randomness as we go along. □

7.5. Some Concluding Remarks

DEFINITION 7.12. The classes $\mathsf{AM}[k]$ are AM where only k rounds are allowed. ◇

Lecture 15, 26 Mar

However, we have the following:

THEOREM 7.13. $\mathsf{AM}[k] = \mathsf{AM}[2]$, *$\forall k \geqslant 2$, $k = O(1)$.*

So all of these are the same: Arthur asks a question, Merlin responds, and Arthur runs a verification.

Now MA is the class where Merlin speaks first. So $\mathsf{NP} \subseteq \mathsf{MA}$ and $\mathsf{BPP} \subseteq \mathsf{MA}$. So MA amounts to Merlin provides a proof, and Arthur can verify with randomness instead of just deterministically. So just as under circuit complexity assumptions $\mathsf{P} = \mathsf{BPP}$, $\mathsf{NP} = \mathsf{MA}$. It is clear that $\mathsf{MA} \subseteq \mathsf{AM}[2]$, then $\mathsf{AM}[2] \subseteq \Pi_2$, since we can just say "for all questions Arthur asks, does there exist an answer Merlin can give?" except with some manipulation to make the verification deterministic.

THEOREM 7.14. *If* $\mathsf{coNP} \subseteq \mathsf{AM}[2]$ *then* $\mathsf{PH} = \Sigma_3$.

Notice that the protocol for $\#\mathsf{SAT}$ ran in n rounds. Now it is possible that problems in coNP can have protocols in a constant number of rounds. This theorem shows that this is unlikely.

COROLLARY 7.15. *If Graph Isomorphism is* NP*-complete, then* $\mathsf{PH} = \Sigma_3$.

So Graph Isomorphism is unlikely to be NP-complete. This is simply because if this were the case then Graph Non-Isomorphism would be coNP-complete, but has a 2-round protocol. So all of $\mathsf{coNP} \subseteq \mathsf{AM}[2]$, so $\mathsf{PH} = \Sigma_3$.

So the study of interactive proofs has interesting consequences.

CHAPTER 8

Complexity of Counting Problems

Counting problems are the problems where there is an underlying computational problem, and the real problem is to find the number of satisfying assignments to the underlying problem.

EXAMPLE 8.1. In #SAT, given ϕ we want to find the number of satisfying assignments to ϕ. $\diamond$

EXAMPLE 8.2. In #PM, given bipartite $G(V, W, E)$ we want to find the number of perfect matchings in G. $\diamond$

EXAMPLE 8.3. In #PATH, given directed G, s, t find the number of paths $s \rightsquigarrow t$. $\diamond$

So these are problems where the answer is a number. For any NP-complete problem we can always find the corresponding counting problem. So clearly this class contains many natural problems.

We now make a formal definition, and in particular these three examples will be counting problems under this definition.

DEFINITION 8.4. A counting problem is a function $f : \{0,1\}^* \to \mathbb{Z}^+ = \{0, 1, 2, \ldots\}$ $\diamond$

DEFINITION 8.5. The class #P is the class of all functions $f : \{0,1\}^* \to \mathbb{Z}^+$ such that there is a polynomial time NTM M such that $\forall x \in \{0,1\}^*$, $f(x)$ is the number of accepting computational paths of M on input x. That is, given the configuration tree of M $f(x)$ is the number of accepting leaves. $\diamond$

Now clearly it is the case that #SAT $\in$ #P, since a NTM can just guess an assignment and accept if and only if it is a satisfying assignment. The number of accepting paths is exactly the number of satisfying assignments. Similarly #PATH $\in$ #P and #PM $\in$ #P.

Now let us add one more example to our list.

EXAMPLE 8.6. In PERM, for $M_{n\times n}$ in $\mathbb{Z}^+$, $\mathrm{perm}(M) = \sum_{\sigma:[n]\to[n]} \prod_{i=1}^{n} M_{i\sigma(i)}$. $\diamond$

This is similar to the determinant, which also has a factor of (-1) before the product, depending on if the permutation is even or odd. One of the interesting things to study is the difference between permanent and determinant. Well determinant can be done in polynomial time, but we will see that the permanent is very hard, in fact it is #P complete, so it is harder than the entire polynomial hierarchy. This is very astonishing.

Now how does PERM fit into #P? Well the permanent is really the same as the perfect matching problem on bipartite multi-graphs, since if we write the adjacency matrix, the permanent of the matrix is exactly the number of perfect matchings.

So let us now study this class.

8.1. Completeness for $\#\mathsf{P}$

We will talk about what completeness means.

Now usually completeness is done using reductions. What this amounts to is if $A \leqslant_P B$, then we can reduce to an instance of B and solve it, in Karp reductions, or a polynomial number of instances of B, in Turing reductions. In most cases for NP, Karp reductions suffice.

In this case however, Turing reductions make more sense.

Definition 8.7. Let $f : \{0,1\}^* \to \mathbb{Z}^+$ be a counting problem, then P^f is the class of problems we can solve in polynomial time with oracle for f, that is, by solving a polynomial number of instances of f. In particular $\mathsf{P}^{\#\mathsf{P}} = \bigcup_{f \in \#\mathsf{P}} \mathsf{P}^f$. $\diamond$

Note that PERM for matrices over $\mathbb{Z}$ can be done by using two calls to PERM for matrices over $\mathbb{Z}^+$, viewing it as $f_1 - f_2$ for $f_1, f_2 \in \#\mathsf{P}$. So formally PERM for matrices over $\mathbb{Z}$ is not in $\#P$, but it is in $\mathsf{P}^{\#P}$.

Definition 8.8. Given $f : \{0,1\}^* \to \mathbb{Z}^+$, $0 \leqslant f(x) \leqslant 2^{n^{O(1)}}$, f is called $\#\mathsf{P}$-complete if $f \in \#\mathsf{P}$, and for any $g \in \#\mathsf{P}$, $g \in \mathsf{P}^f$. $\diamond$

This is an immediate result:

Theorem 8.9. *$\#\mathsf{SAT}$ is $\#\mathsf{P}$-complete.*

Proof. We already know that $\#\mathsf{SAT} \in \#\mathsf{P}$.

So take any function g. Then there is an underlying NTM M so that g is the number of accepting computational paths on M, which is the number of non-deterministic choices y such that $V(x,y) = 1$. Now this can be simulated perfectly by a circuit, so there is some boolean formula ϕ_x such that $V(x,y) = 1 \iff \phi_x(y) \in \mathsf{SAT}$. So the number of accepting computational paths is the same as the number of satisfying assignments to ϕ_x. So $g \in \mathsf{P}^{\#\mathsf{SAT}}$. $\square$

In general if we pick an NP-complete problem, its counting version will probably be $\#\mathsf{P}$-complete, since these reductions preserve the number of satisfying assignments.

Theorem 8.10 (Valiant). *$\#\mathsf{SAT}$, $\#\mathsf{PM}$, $\#\mathsf{PATH}$, PERM are all $\#\mathsf{P}$-complete.*

This is very surprising since $\mathsf{PM} \in \mathsf{P}$ but the counting version is $\#\mathsf{P}$-complete.

8.2. $\#\mathsf{P}$ and PH

So now we will focus on how powerful this class is.

We already know that $\mathsf{NP} \in \mathsf{P}^{\#\mathsf{P}}$ and $\mathsf{coNP} \in \mathsf{P}^{\#\mathsf{P}}$, and $\mathsf{P}^{\#\mathsf{P}} \in \mathsf{PSPACE}$. Now the surprising thing is that $\mathsf{PH} \in \mathsf{P}^{\#\mathsf{P}}$.

Theorem 8.11 (Toda). $\mathsf{PH} \subseteq \mathsf{P}^{\#\mathsf{P}}$.

The proof is long, so we will break it up into steps and do one step at a time. The goal will be to show that any language $L \in \mathsf{PH}$ Turing reduces to $\#\mathsf{SAT}$. The steps involved are: first, we define a problem called $\oplus\mathsf{SAT}$, or parity sat, which is just whether the number of satisfying assignments is even or odd. As it turns out this is more or less as difficult as $\#\mathsf{SAT}$. Then we will show that any language $L \in \mathsf{PH}$ in fact reduces to $\oplus\mathsf{SAT}$. But this reduction will be a randomized reduction, which is a reduction that works with high probability. The third step will be to get rid of the randomness in the reduction, so that L Turing reduces deterministically to $\#\mathsf{SAT}$.

The second step works as follows: the languages in PH can be computed by a family of circuits $\{C_n\}$ of a certain type, and $\oplus\mathsf{SAT}$ can be solved by a family of circuits $\{D_n\}$ of another type. These circuits have constant depth and exponential size. The difference is that $\{C_n\}$ can contain AND and OR gates, and $\{D_n\}$ can contain AND, OR, and XOR gates. The difficulty is with the fan-in of the gates allowed. For $\{C_n\}$, the gates have unbounded fan-in, but for $\{D_n\}$ only the XOR gate has unbounded fan-in and the AND and OR gates have polynomially-bounded fan-in.

Definition 8.12. $\oplus\mathsf{SAT} = \{\phi \mid \phi \text{ has an odd } \# \text{ of satisfying assignments}\}$. ◇

Definition 8.13. $\oplus\mathsf{P}$ is the class of languages L such that there is a polynomial time NTM M such that for all input x, $x \in L \iff M$ has an odd number of accepting computations on x. ◇

The following is clear:

Theorem 8.14. *$\oplus\mathsf{SAT}$ is complete for $\oplus\mathsf{P}$.*

Definition 8.15. Given A, B, A reduces to B via a randomized reduction, that is, $A \leqslant_{\text{rand}} B$ if there is a deterministic polynomial time machine M such that choosing randomness r, $x \in A \implies M(x,r) \in B$ with high probability, whereas on the other hand $x \notin A \implies M(x,r) \notin B$ with high probability. More precisely, for m the number of random bits, $x \in A \implies$ the number of random strings r such that $M(x,r) \in B \geqslant \frac{2}{3}2^m$, whereas $x \notin A \implies$ the number is $\leqslant \frac{1}{3}2^m$. ◇

The following lemmas are important to showing that $L \leqslant_{\text{rand}} \oplus\mathsf{SAT} \implies L \leqslant \#\mathsf{SAT}$.

Lemma 8.16. *Given ϕ_1, ϕ_2 boolean formulas, then one can construct formulas ψ, ψ' such that the number of satisfying assignments to ψ, $\#\psi = \#\psi_1 + \#\psi_2$, and $\#\psi' = \#\psi_1 \cdot \#\psi_2$.*

So in some sense we can simulate addition and multiplication using formulas.

Proof. ψ' is easy: imagine that we made the variables of ϕ_1 and ϕ_2 distinct, and then take $\psi'(X,Y) = \phi_1(X) \wedge \phi_2(Y)$.

Similarly $\psi(X,Y,w) = (\phi_1(X) \wedge Y \wedge w) \vee (\phi_2(Y) \wedge X \wedge \overline{w}$, where w is a new variable. □

Lemma 8.17. *Given $a \in \mathbb{Z}$, If $a \equiv -1 \pmod{2^{2^i}}$ then $4a^3 + 3a^4 \equiv -1 \pmod{2^{2^{i+1}}}$, and if $a \equiv 0 \pmod{2^{2^i}}$ then $4a^3 + 3a^4 \equiv 0 \pmod{2^{2^{i+1}}}$.*

So the way we use this is that if we have a formula ϕ whose number of satisfying assignments is a, then we can create another formula ψ whose satisfying number of assignments is $4a^3 + 3a^4$ until the modulus is pretty large. Then roughly speaking we know an integer exactly if we know its remainder modul some large number. So we go from knowing the parity to knowing the integer exactly. Furthermore, the size of this new formula is still $\text{poly}(n)$. Lecture 16, 31 Mar

Lemma 8.18. *Let $L \leqslant_{\text{rand}} \oplus\mathsf{SAT}$, then on input x and randomness r with $m = |r|$, the number of random bits, we have $(x,r) \rightsquigarrow \phi_{x,r}$ such that $x \in L \implies \phi_{x,r} \in \#r \oplus \mathsf{SAT} \geqslant \frac{99}{100}2^m$ and $x \notin L \implies \#r\phi_{x,r} \in \oplus\mathsf{SAT} \leqslant \frac{1}{100}2^m$.*

Proof. So we will now design a NTM M such that counting the number of accepting computations of M on input x enables us to decide if $x \in L$:

```
M := "On input x, guess r.
        Construct $\phi_{x,r}$, $\psi_{x,r}$
        Guess a satisfying assignment to $\psi_{x,r}$
        Accept if it is indeed a satisfying assignment.
```

So the number of accepting computations of M is

$$\sum_r \#\psi_{x,r} \equiv -(\#r\phi_{x,r} \equiv -1 \pmod 2) \pmod{2^{2^k}}$$
$$\equiv -(\#r\phi_{x,r} \in \oplus\mathsf{SAT}) \pmod{2^{2^k}}.$$

Hence knowing the number of accepting computations of M means knowing the number of r such that $\phi_{x,r} \in \oplus\mathsf{SAT}$, which amounts to knowing if $x \in L$ or not. □

That finishes the first step. We now show the second step.

Theorem 8.19. *$L \in \mathsf{PH}$ if and only if L is decided by $\{C_n\}_{n=1}^{\infty}$ such that $|C_n| \leqslant 2^{n^{O(1)}}$, $\mathrm{depth}(C_n) \leqslant O(1)$, with only AND and OR gates with unbounded fan-in, and $\{C_n\}$ is uniformly generated*[1].

We only need the forward direction, so we will only prove the forward direction.

Proof of $\implies$. Recall that

$$x \in L \iff \exists y_1 \forall y_2 \ldots \exists y_{k-1} \forall y_k, V(x, y_1, \ldots, y_k) = 1.$$

We can construct a circuit such that existential quantifiers become OR gates and univeral quantifiers become AND gates with the inputs the possibilities of each y_k. The k-th gate has fan-in 2^{n^c} since $|y_k| = n^c$. Then the leaf at the end of y_k can just be a circuit C_V agreeing with V. C_V can be constructed with constant depth using exponential fan-in.

Because of the regular structure of this circuit it is uniformly generated. □

Theorem 8.20. *$L \in \oplus\mathsf{P}$ if and only if L is decided by $\{D_n\}$ such that $|D_n| \leqslant 2^{n^{O(1)}}$, $\mathrm{depth}(D_n) = O(1)$, with only AND and OR gates with $n^{O(1)}$ fan-in and XOR gates with unbounded fan-in, and $\{D_n\}$ is uniformly generated.*

Then the idea will be that a language $L \in \mathsf{PH}$ is characterized by $\{C_n\}$ which can be reduced randomly to $\{D_n\}$, then by reverse characterization gives a language in $\oplus\mathsf{P}$.

Proof. In the forward direction, take the canonical problem $\oplus\mathsf{SAT}$, then compute the following circuit: on input ϕ with n inputs, at the top there is a big XOR gate with fan-in 2^n, one for each $a \in \{0,1\}^n$. Feeding in is a circuit evaluating ϕ on a. Then this is a circuit calculating the parity of the number of satisfying assignments with exponential size but constant depth and the AND and OR gates have polynomial fan-in, since without loss of generality we can think of ϕ as a $3{-}\mathsf{SAT}$ formula.

[1]Specifically there is a polynomial time algorithm that given $i, j \in \{1, \ldots, |C_n|\}$ the algorithm outputs the type of gate i and the index of the gate feeding into the j-th input of gate i

The reverse direction is tricky, since in the forward direction our circuit had depth 3. Now if L is decided with $\{D_n\}$ with these properties then we will construct a polynomial time NTM M such that on input x, $x \in L \iff D_n(x) = 1 \iff$ M has an odd number of accepting computations on input x.

The idea is as follows: M will keep generate a local view of D_n and every time it encounters a parity gate, it will choose one of its inputs non-deterministically. Then it will focus on the subset below it. Then assume as an induction hypothesis that we can simulate circuits of smaller and smaller size such that the output of the gate is the same as the number of accepting computations. Now for the AND gate, Now note that if an integer is a multiple of other integers, the first one is odd if and only if all of the rest are odd, and it can only have polynomial-size fan-in, so we can construct the circuit below it. If it hits an XOR gate then it will go non-deterministically. The OR case is similar. Formally:

```
M := "On input $x$,
       SIMULATE($D_n$)"
```

where

```
SIMULATE($C$) := "Generate the top gate $G$ of $C$
                  If $G = \oplus$, non-deterministically choose one of its subcircuits $C'$
                     SIMULATE($C'$)
                     Accept if this accepts
                  If $G = \wedge$ for $i = 1, 2, \dots, n^{O(1)}$
                     SIMULATE($C_i$) where $C_i$ is the $i$-th subcircuit.
                     Accept if all $i = 1, 2, \dots, n^{O(1)}$ accept
                  Output $T/F$ if $C$ is constant-size."
```

□

We will use the following lemma:

Lemma 8.21 (Valiant-Vazirani). *Let $S \subseteq \mathbb{F}_2^N$, $S \neq \emptyset$, and $H \subseteq \mathbb{F}_2^N$ be a random affine subspace of dimension i, $i \in_R \{0, 1, 2, \dots, N\}$. Then* $\Pr[|H \cap S|$ *odd*$] \geqslant \frac{1}{4N}$.

Now we come back to the issue of simulating $\{C_n\}$ by $\{D_n\}$. The simulation is gate by gate. Without loss of generality suppose we have an OR gate with fan-in 2^N where $N = n^{O(1)}$ and input $w_1, \dots, w_{2^N}$, then replace it with a circuit with AND, OR, and XOR gates that does the same thing using a randomized algorithm.

Lemma 8.22. $\{C_n\} \overset{randomized}{\rightsquigarrow} \{D_n\}$.

Proof. Naively, suppose we try to replace the OR gate by an XOR gate since it is the only one of huge fan-in and hope that it works. But it works in one of two cases: if $\bigvee_{i=1}^{2^N} w_i = 0$ then $\bigoplus_{i=1}^{2^N} w_i = 0$. But if $\bigvee_{i=1}^{2^N} w_i = 1$ then $\bigoplus_{i=1}^{2^N} w_i$ depends, so it does not quite work.

We now try a "mild" simulation (see Figure 8.23): let $s \in \mathbb{F}_2^N$, then in the new circuit with the XOR gate, take the s-th input and replace it with an AND gate with inputs w_s and the question "$s \in H$?" for a random affine subspace H with dimension from $\{0, 1, \dots, N\}$. If the OR is 0, then then the XOR is still 0. On the other hand if the OR is 1, then let $S = \{s \mid w_s = 1\}$, so $S \neq \emptyset$. So $w_s = 1$ will pass all the way up if and only if $s \in H$. So the output is simply the parity of $|H \cap S|$, which is 1 with probability $\geqslant \frac{1}{4N}$ by the Valiant-Vazirani Lemma.

Lecture 17, 2 Apr

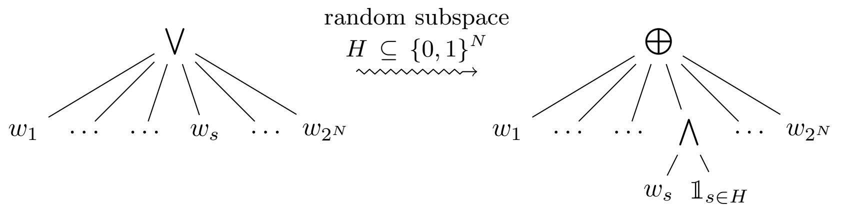

random subspace
$H \subseteq \{0,1\}^N$
$\bigvee$
w_1 … … w_s … w_{2^N}
$\bigoplus$
w_1 … … $\bigwedge$ … w_{2^N}
w_s $\mathbb{1}_{s\in H}$

FIGURE 8.23. The "mild simulation" of an OR gate.

Then the strong simulation is to just do this with an OR gate instead of an XOR gate, and we have this randomness for a polynomial number of gates, so that if the OR is 0, we still get 0, but if the OR is 1, then we get 1 again with as high probability as we like. Then by union bound, with high probability over the choice of randomness, $C_n(x) = D_n(x)$, $\forall |x| = n$. $\square$

So recall that we have $x \in L \iff C_n(x) = 1 \overset{\text{w.h.p.}}{\iff} D_n(x) = 1 \iff M(x)$ has odd number of accepting computations for some NTM M. So $L \leqslant_{\text{rand}} \oplus\mathsf{P}$.

This reduction is a bit more subtle than that, since we do not have C_n in hand. What we have is we have a way of going from an algorithm computing local views of C_n and randomness to an algorithm computing local views of D_n.

This completes the proof.

CHAPTER 9

Proof Complexity

Lecture 12, 10 Mar
Pratik Worah

One thing that Cook had in mind with P vs. NP was with proof systems. He wanted to show that there is no "super" proof system that had polynomial sized proofs for tautology boolean functions.

Cook wanted to try to start with the weakest proof systems and work up to powerful proof systems.

9.1. Proof Systems

Definition 9.1. For a language L, a *proof system* is an algorithm $P(x, \pi)$ where $x, \pi \in \Sigma^*$, x is the input and π is the proof, that has three characteristics:

- Completeness: $\forall x \in L, \exists \pi \quad P(x, \pi) = 1$.
- Soundness: $\forall x \notin L \forall \pi \quad P(x, \pi) = 0$.
- Efficiency: P runs in time $O(\text{poly}(|x|, |\pi|))$. ◇

We will work with what are called *propositional proof systems*, which are proof systems where $L = \mathsf{UNSAT}$.

Lemma 9.2. $\mathsf{NP} = \mathsf{coNP}$ *if and only if there exists a propositional proof system which has polynomial-sized proofs for all* $L \in \mathsf{UNSAT}$.

Proof. If $\mathsf{NP} = \mathsf{coNP}$ then by definition of NP we have a propositional proof system with polynomial-sized proofs for $L \in \mathsf{UNSAT}$

If there exists a propositional proof system with polynomial-sized proofs for $L \in \mathsf{UNSAT}$, we just use that UNSAT is coNP-complete. □

So the question is, does there exist a super proof system?

We start with what is called resolution. In resolution, we have clauses, literals, and variables. For example, we start with a k-CNF formula (called axioms) and we derive new clauses using the following rules, until we reach an empty clause (to refute the system):

- Cut Rule: From $C_1 \vee x$ and $C_2 \vee \overline{x}$ we can derive $C_1 \vee C_2$.
- Weakening: From C we can derive $C \vee x$.

Example 9.3. Suppose we have the pigeonhole principle where we put try two pigeons in one hole, we want to try to show we can and derive an empty clause. This is $\neg\text{PHP}_1^2$. The axioms are $\overline{x_1} \vee \overline{x_2}$ (injection), and x_1 and x_2 (everywhere defined), where x_1 means pigeon 1 is put into the hole, and x_2 means pigeon 2 is put into the hole. So every pigeon goes into a hole, but every hole can only take one pigeon. Then we want to prove that this does not work.

For pigeonhole principle for $n+1$ pigeons in n holes, $\neg\text{PHP}_n^{n+1}$, this requires size $2^{\Omega(n)}$. ◇

Resolution is a kind of sequential proof system, which we will now define.

Definition 9.4. In a *sequential proof system*, we have A_i boolean formulae, and the following derivation rules: from $A_{i_1}, \ldots, A_{i_k}$ we can derive B. ⋄

The *length* of a sequential proof system is as follows: If we draw the derivations as a directed acyclic graph, eg. in Figure 9.5, then we say the length of the proof system, $L_p(F \vdash 0, \pi)$ is the number of nodes in the DAG.

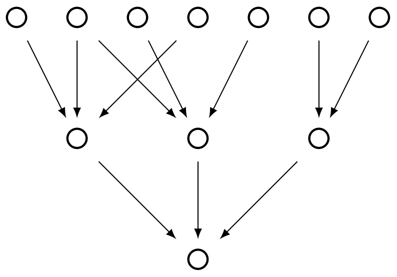


Figure 9.5. The derivations as a DAG

If we convert the DAG to a Tree, then the length $L_{TR}(F \vdash 0, \pi)$ is the number of nodes in the tree.

Here $F \models \emptyset$ means that we semantically show that F implies $\emptyset$, whereas here we have $F \vdash 0$, that we showed syntactically that F implies $\emptyset$.

Now why do we define sequential proof systems in general? Well just as we have resolution, we have what is called cutting planes, in which instead of clauses we have inequalities over $\mathbb{R}$, and the derivation rule is a rounding rule. The goal is to derive the "empty" inequality, for example $-1 \geqslant 0$. This is the origin of LP and SDP proof systems.

In the state-of-the-art we know how to prove lower bounds on the depth of the DAG, but in general we do not know how to show general size lower bounds.

Question 9.6. Prove exponential size lower bounds for random k-CNF in cutting planes? (We know depth lower bounds). ⋄

9.2. An lower bound for Resolution

We now prove a lower bound for resolution. We define what are called (m, γ)-expanders. So consider a bipartite graph $G = (L \cup R, E)$, and take a set $U \subseteq L$, then look at the neighbors of U, $N(U) \subseteq R$. G is a (m, γ)-*expander* if for all U, $|U| \leqslant m$ and $|N(U)| \geqslant (1 + \gamma)|U|$.

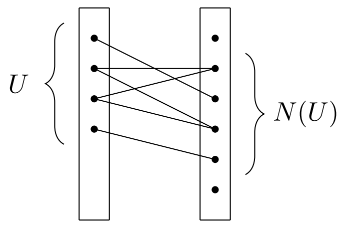


Figure 9.7. Example of a (m, γ)-expander graph

We have a Variable vs Constraint Graph, with N constraints and n variables, $N \gg n$. Each constraint is linked to a certain number of variables.

A k-CNF formula F is a (m, γ)-expander if its constraint bipartite graph is a (m, γ)-expander.

THEOREM 9.8. *Every (m, γ)-expanding k-CNF F requires $2^{\Omega(m\gamma)}$ sized proofs in Resolution.*

As it turns out most bipartite graphs are expander graphs, so this means that most formulas have exponential sized proofs.

The proof is due to Ben-Sasson and Wigderson in 1997.

FIRST PART OF THE PROOF. The *width* $W(F, \pi)$ of a proof π for a formula F is the maximum size of a clause in π. The width $W(F)$ of a formula F is $\min_\pi W(F, \pi)$. We first show the width-size trade-off: $W(F) \leqslant 3\sqrt{n \log S(F)} + k$ (using induction), then we show that there is a large width clause in every refutation of F (by the probabilistic method).

First, $W(F \vdash A) \leqslant d$ implies $W(F \vdash A \vee z) \leqslant d + 1$, where z is a literal. This is easy, because it is just weakening.

Next, $W(F|_{z=0} \vdash \emptyset) \leqslant d - 1$, $W(F|_{z=1} \vdash \emptyset) \leqslant d$ implies $W(F \vdash \emptyset) \leqslant \max\{k, d\}$. For tree-like proofs, $W_{TR} \leqslant k + \log S_{TR}(F)$, for general DAGs $W(F) \leqslant k + 3\sqrt{n \log S(F)}$. We will show the one for trees by induction.

On a proof of size one and one variable we always have that something of the form x and $\neg x$ leads to $\emptyset$. Then in general at the end we have the situation in Figure 9.9, with $|S_x| \leqslant |S_{\overline{x}}|$.

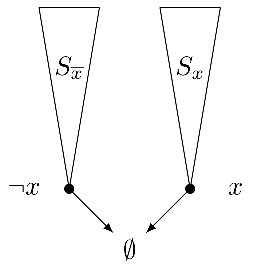


FIGURE 9.9. The general situation where S_x and $S_{\overline{x}}$ lead to $\emptyset$

So $W(F|_{x=0} \vdash 0) \leqslant k + \log S_x \leqslant k + \log_2 \frac{S}{2} = k + \log_2 S - 1$. Similarly $W(F|_{x=1} \vdash 0) \leqslant k + \log_2 S$. □

Now let $|F|$ be the number of clauses in F, and $|\text{Vars}(F)|$ be the number of variables of F. Lecture 13, 12 Mar Pratik Worah

To prove the second part, we need the following:

THEOREM 9.10 (Hall's Matching Theorem). *For a partite graph $(U \cup V, E)$ a perfect matching exists if and only if for all $U' \subseteq U$, $|N(U')| \geqslant |V|$.*

A formula F is *minimum unsatisfiable* if it is unsatisfiable but any $F' \subsetneq F$ is satisfiable.

LEMMA 9.11 (Tarsi). *If F is minimum unsastiable then $|F| > |\text{Vars}(F)|$.*

LEMMA 9.12. $F \underset{\min}{\models} C \implies |F| \geqslant |\text{Vars}(F)| - |C|$.

Now onto the second part of the proof.

SECOND PART OF THE PROOF OF THEOREM 9.8. Let μ be a map from clauses to non-negative integers, $\mu(C) = \min\{|F'| : F' \subseteq F, F' \models C\}$. So for a formula F of clauses $A_1, \ldots, A_N$ then $\mu(A_i) = 1$, $\mu(\emptyset) = N$, and $\mu(C) \leqslant \mu(D) + \mu(E)$ if D and E derive C.

When F is a (m, γ)-expander then $\mu(\emptyset) \geqslant m$ by Hall's Theorem. So from the beginning to the contradiction, μ is increasing from 1 to m. Fix a proof, then there exists c in the proof such that $\frac{m}{2} \leqslant \mu(C) \leqslant m$. By Lemma 9.12, $|C| \geqslant |\text{Vars}(F')| - |F'| \geqslant (1+\gamma)|F'| - |F'| = \gamma|F|$. So we are done. □

We now prove Lemma 9.12.

LEMMA 9.13. *Given a bipartite graph of the clauses and variables of a formula F, there exists clauses U such that $|N(U)| < |U|$.*

PROOF. Let F' be the minimal such clause, and $F'' = F \setminus F'$. Then there is a perfect matching from F'' to the variables of F'', since F' is minimal. So F'' will be satisfiable, by Hall's Theorem. □

PROOF OF LEMMA 9.12. First fix a clause C, and then take a restriction $\rho \in \{0,1\}^{\text{Vars}(C)}$ that does not satisfy C, then we claim $F|_\rho \underset{\min}{\models} \emptyset$. Now for $\alpha \in \{0,1\}^{\text{Vars}(F)}$, if F is not the minimal the there is $F' \subsetneq F$ such that if α agrees with ρ then $F'(\alpha) \models C(\alpha)$ and otherwise $F'(\alpha) \models 1$, so $F' \models C$, contradicting that F is the minimal formula that implies $\emptyset$.

Then by Tarsi's Lemma we have $|F| \geqslant |F|_\rho| > |\text{Vars}(F|_\rho)| = |\text{Vars}(F)| - |C|$. □

In some ways, Proof Complexity is worse off than Circuit Complexity in terms of progress.
[1]

[1]A good resource for this topic is the notes from Eli Ben-Sasson, on which these lectures were based.

CHAPTER 10

Circuit Lower Bounds

10.1. An Example of a Circuit Lower Bound

Lecture 17, 2 Apr (cont'd)

We prove the following result:

THEOREM 10.1. $\mathsf{PARITY} \notin \mathsf{AC}^0$.

Recall that AC^0 is the set of languages computed by poly-size, $O(1)$-depth circuits with unbounded fan-in, and NC^1 is the set of languages computed by poly-size, $O(\log n)$-depth circuits with fan-in 2. For both we only allow AND and OR gates. Of course $\mathsf{AC}^0 \subseteq \mathsf{NC}^1$.

Now obviously $\mathsf{PARITY} \in \mathsf{NC}^1$ since we can just go over all of the bits in a logarithmic depth tree.

We need to introduce some notation:

For the proof it will be more convenient to view bits as $\{+1, -1\}$ instead of $\{0, 1\}$. Here $+1$ corresponds to FALSE and -1 corresponds to TRUE. The correspondence to binary bits is $(-1)^b$ corresponds to b, where $b \in \{0, 1\}$. In the boolean world, we had $a \oplus b$, here we have the product $(-1)^a(-1)^b$. Our boolean functions become $f : \{-1, +1\}^n \to \{-1, +1\}$. In particular, $\mathsf{PARITY}(x_1, \ldots, x_n) = \prod_{i=1}^n x_i$.

Note that for an OR gate if all of the inputs are $+1$, then the output is $+1$, which is FALSE, but if any of the inputs is -1, then the output is -1, which is TRUE.

The reason we switch to this notation is that it allows us to do Fourier analysis, and the proof we give is Fourier analytic.

A more precise statement of Theorem 10.1 is as follows:

THEOREM 10.2. *Any depth-d AND, OR circuit that computes parity must have at least $2^{\Omega(n^{1/2d})}$ gates.*

The following measure of complexity is useful: every function $f : \{-1, +1\}^n \to \{-1, +1\}$ can be represented as a polynomial, then we can worry about the degree of the polynomial. Then the measure is just if the function can be computed by a polynomial of low degree, or if it requires a polynomial of high degree.

The idea is that we prove that if we have a circuit whose size is not too large, then the circuit represents a boolean function that can be computed with a low degree polynomial.

That is, the basic reasons why this theorem is true are the following: poly(n) size circuits are "well-approximated" by "low degree" polynomials where "low-degree" can be thought of as $\sqrt{n}$ and "well-approximated" means they agree on most, say 99%, of the inputs; and no polynomial of degree $\sqrt{n}$ can compute PARITY on 90% of the inputs.

The issue is when we talk about polynomials we are implicitly regarding -1 and $+1$ as real numbers and the polynomials are actually polynomials over reals.

It turns out this requires a few too many steps. But as it turns out it works out for any field larger than two elements. So the convention is that all computations are over $GF(3)$. So when we say $f : \{-1,+1\}^n \to \{-1,+1\}$, we are really thinking of $-1,+1 \in GF(3)$. Now $0 \in GF(3)$ but we do not care about inputs whose elements are 0.

FACT 10.3. *Every function $f : \{-1,+1\}^n \to \{-1,+1\}$ can be computed by a multilinear polynomial (over $GF(3)$).*

This is also true over $\mathbb{R}$, which is what leads to Fourier analysis.

PROOF. For $f(x_1, x_2, \ldots, x_n)$, we look at whether x_1 is set to -1 or $+1$. So $f(x_1, x_2, \ldots, x_n) = \frac{1+x_i}{-1} f(1, x_2, \ldots, x_n) + \frac{1-x_i}{-1} f(-1, x_2, \ldots, x_n)$. Then by induction we can continue. Then we can assume that it is multilinear, since say any term x_i^2 can be dropped to x_i or $-x_i$ since we are working over $\{-1,+1\}$. □

Let us now prove Theorem 10.2.

PROOF OF THEOREM 10.2. We first prove that no polynomial of degree $\sqrt{n}$ can compute parity on 90% on inputs.

Suppose on the contrary that there was a multilinear polynomial $p(x_1, \ldots, x_n)$ of $\sqrt{n}$-degree that agrees with $\mathsf{PARITY} = \prod_{i=1}^n x_i$ on set of inputs $D \subseteq \{-1,+1\}^n$, with $|D| \geqslant \frac{9}{10} 2^n$.

Then this immediately implies the following: any function on D is computed by a $\frac{n}{2} + \sqrt{n}$ degree multilinear polynomial. But $|D|$ is still at least $\frac{9}{10}2^n$.

For any $f^r : D \to GF(3)$ we have $f = \sum_{S \subseteq [n]} c_s \prod_{i \in S} x_i$, for arbitrary coefficients c_s. Then we can split this to

$$\begin{aligned} f &= \textstyle\sum_{|S| \leqslant \frac{n}{2}} c_s \prod_{i \in S} x_i + \sum_{|S| > \frac{n}{2}} c_s \prod_{i \in S} x_i \\ &= \textstyle\sum_{|S| \leqslant \frac{n}{2}} c_s \prod_{i \in S} x_i + \sum_{|S| > \frac{n}{2}} c_s \prod_{i \in \overline{S}} x_i \prod_{i \in [n]} x_i \end{aligned}$$

Then on D we get $\sum_{|S| \leqslant \frac{n}{2}} c_s(\deg \leqslant \frac{n}{2}) + (\deg \leqslant \frac{n}{2})(\deg \leqslant \sqrt{n})$. But we know that $\dim(\text{space of functions on } D) = |D| \geqslant \frac{9}{10}2^n$, whereas

$$\dim\left(\text{polynomials of } \deg \leqslant \frac{n}{2} + \sqrt{n}\right) = \sum_{j=0}^{\frac{n}{2}+\sqrt{n}} \binom{n}{j} = \left(\frac{1}{2} + k\right) 2^n$$

so there is a contradiction.

Now we show that $O(1)$ depth $\text{poly}(n)$ size circuits are "well-approximated" by "low-degree" polynomials.

Formally, take a circuit C whose depth is d and M gates, then there is a polynomial $p(x_1, \ldots, x_n)$ of $\deg \leqslant (2\ell)^d$ which agrees with C on $1 - \frac{M}{2^\ell}$ fraction of inputs.

So we simulate every little part of the circuit by a small polynomial and get a weak simulation, then amplify to a strong simulation. This is similar to the proof of Toda's theorem except instead of simulating by circuits we simulate by polynomials.

So we claim that there is a randomized construction of a degree 2ℓ polynomial g such that $g(x_1, \ldots, x_n) = \bigvee_{i=1}^n x_i$ with probability $\geqslant 1 - \frac{2}{\ell}$ for an arbitrary fixed input.

First we will do the weak simulation. Choose independent $c_1, \ldots, c_n$ randomly from $GF(3)$, and then let $g(x_1, \ldots, x_n) = 1 + (\sum_{i=1}^n c_i(1 - x_i))^2$. So suppose $\bigvee_{i=1}^n x_i = +1$, then $\forall i, x_i = +1$ so $g(\cdot) = +1$. On the other hand suppose that

$\bigvee_{i=1}^n x_i = -1$, so for some i, $x_i = -1$. Then the vector $(1-x_1, 1-x_2, \ldots, 1-x_n) \neq 0$. Then when we take its inner product with a random vector, we get a scalar not equal to 0 with probability $\frac{2}{3}$. Then we square it, add 1, and then we get $g = -1$. This succeeds with probability $\frac{2}{3}$.

The strong simulation is as follows: let $G(x_1, \ldots, x_n) = -1 - \prod_{j=1}^{\ell} \frac{1+g_j}{2}$ where $g_1, \ldots, g_\ell$ are independently chosen copies of the weak simulation. So $\deg G \leqslant 2\ell$. Then $\bigvee_{i=1}^n x_i = +1$ then $G(\cdot) = +1$. On the other hand if $\bigvee_{i=1}^n x_i = -1$ then with probability $1 - (\frac{1}{3})^\ell$ at least one $g_j(\cdot) = -1$ and $G(\cdot) = -1$. The simulation succeeds with probability $1 - (\frac{1}{3})^\ell$.

So all together we can get a polynomial of degree $(2\ell)^d$ that agrees with the circuit on $1 - \frac{M}{2^\ell}$ fraction of inputs. Lecture 18, 7 Apr

Now we put these two results together. Let $(2^\ell)^d = \sqrt{n}$ so that $\ell = \frac{1}{2} n^{\frac{1}{2d}}$ and $\frac{M}{2^\ell} = \frac{1}{10}$ so $M = \frac{1}{10} 2^{\frac{1}{2} n^{\frac{1}{2d}}}$. So to compute parity a depth d circuit must use $\geqslant \frac{1}{10} 2^{\frac{1}{2} n^{\frac{1}{2d}}}$. □

CHAPTER 11

Quantum Computing

We will discuss a few results in Quantum Computing, namely Grover's Search Algorithm. So say that there are n boxes, and one box has a hidden ball, and we want to find the box that contains the ball. Now any typical algorithm will need to open all the boxes. Even a randomized algorithm will need to open most of the boxes to succeed with high probability.

In the quantum world, we first need to define what it means to open a box quantumly, for which there is a definition, and with that definition, it is possible to do this with only opening $\sqrt{n}$ boxes.

We will have comparisons to our previous model of computing, which the literature refers to as the "classical model".

11.1. The Quantum Model of Computation

In classical computing, the basic unit is the bit, and so we have n-bit strings, so we have n is the input size, and we want algorithms that run in time poly(n). The analog in the quantum world is the quantum bit, or *qubit*, so we have n qubits.

So in the concrete world we can have, for example, a memory register or just a box, that contain either a 0 or 1, so we can look and see a concrete sequence of 0s and 1s. In the quantum case, there really are not n separate boxes. Rather, it is like one big box that contains something, which we cannot see as such. Intuitively, what it contains is a probability distribution over 2^n strings in $\{0,1\}^n$ as we know them in the classical sense; roughly this is called superposition. This is known as the *state* of the quantum register.

What we can then do is press a button "measure", then the box spits out a string with respect to the probability distribution. So whereas classical bits are disjoint, qubits are *entangled*. So we must deal with all 2^n strings at the same time, not any one of them individually. So notationally it is convenient to write $N = 2^n$.

The state of a n-qubit quantum register is a vector in $\mathbb{C}^N$ with norm 1, which generalizes the probability distribution. So henceforth all vectors will be in $\mathbb{C}^N$.

We will need some tools from Linear Algebra.

We have the inner product $u \cdot v = \sum_{i=1}^{N} u_i v_i^*$. For the length of a vector, we have the length squared is $\|u\|^2 = u \cdot u = \sum_{i=1}^{N} |u_i|^2$.

A *unitary* $N \times N$ *matrix* U satisfies $UU^* = I$, where U^* is the conjugate transpose of U. Over real numbers this amounts to $UU^T = I$, then U is called *orthogonal*. Now the row vectors of U are unit vectors (that is, they have norm 1) and they are orthogonal to each other. To see this we just write out the product and see this holds. So the rows (and columns) of unitary U are an orthonormal set.

We should really think of a matrix U as a transformation $U : \mathbb{C}^N \to \mathbb{C}^N$ sending $u \mapsto Uu$. If U is unitary then $u \cdot v = Uu \cdot Uv$. In particular, $\|u\| = \|Uu\|$.

Now we will formally define what a quantum state is. Choose an orthonormal basis for $\mathbb{C}^N$ and denote its vectors as $|x\rangle$.

Definition 11.1. A *state of an n-qubit register* (or a quantum computer) is a vector $\sum_{x\in\{0,1\}^n} \alpha_x |x\rangle$ where $\alpha_x \in \mathbb{C}$ are complex numbers such that $\sum_{x\in\{0,1\}^n}$. $\diamond$

Now we can think of $|\alpha_x|^2$ as the probability distribution. But there is more information than just the probability distribution, and that is important for the model.

Note that a quantum state is a superposition over all 2^n classical states.

Now we will see what computation is.

So classical computation was the Turing Machine model, with the tape or the register, which had the contents of the machine. Initially it has a 0-1 input string x of length n, and then the machine does local computations on a little window, then it continues for ℓ steps, where ℓ counts as the running time. In the end, we output 0 or 1, for example, we look at the first cell and accept if it outputs 1, otherwise we reject.

So quantum computation can also be viewed in this manner, but we will first look at a definition that is completely different. It takes some time to realize that in some sense the definition is trying to mimic this view of classic computing. The definition will be built in two steps.

The quantum computer is allowed to perform three things. The first is *initialization*, which sets the state to $|x_{\text{input}}\rangle$. So we can assume the machine starts with this state. Then just as in classic computing where we apply these local changes, we *apply a sequence of computations*, so noting that the state of the machine must always be a unit vector, the only things we can do are apply unitary transforms, to get $U_\ell \dots U_2 U_1 |x_{\text{input}}\rangle$. So one step of a computation is the application of a unitary operator, and the running time is ℓ. Then we can *measure*, where if the final state is $\sum_{x\in\{0,1\}^n} \beta_x |x\rangle$, the machine gives a state $|x\rangle$ with probability $|\beta_x|^2$. Then we can conclude what we want to conclude; without loss of generality we can just look at the first bit of the string and accept or reject.

Now there is a serious doubt: these unitary matrices are $N \times N$ matrices, so how do we represent this? There is an even more serious doubt, which is that there is no point in having ℓ steps of applying unitary matrices, since we can just combine them into a unitary matrix and only take one step.

So to make this definition nontrivial we need to add some constraints. What is missing right now from the quantum model is some notion of locality. In the classic Turing Machine it is crucial that the machine can only make local changes. So we define a locality on a quantum operator (that is, a unitary matrix) and only allow these local operators.

So the further requirement is as follows: each U_i must be *k-local*, where $k = O(1)$ is small (to parallel a gate with bounded fan-in), and a unitary matrix is *k-local* if it acts only on k qubits. This phrase needs clarification. So here is an example:

Example 11.2. Imagine $k = 2$, $n = 7$, so that $N = 128$. Now we are interested in a matrix U which is 128×128. Now we want it to be 2-local, so what does this mean? We can label each row or column by a bit-string of length 7, then suppose we label the first four rows by $00z$, $01z$, $10z$, and $11z$, where z is a fixed 5-bit string, and we do the same for the first four columns. Then varying z we split the matrix into 32×32 blocks of size 4×4. U is 2-local if it is a block-diagonal matrix with the same 4×4 block matrix on the diagonal, and 0 everywhere else. So the split is just the number of bits, so 3-local would be 16×16 blocks of size 8×8. ◇

There is an alternate view that captures the view "acts only on k qubits" more precisely, and then we will see how the block matrix view is equivalent to this.

So if we have an operator U, all of its entries are indexed by bit-strings of length n, so call the entries $u_{x,y}$. Then on $x \in \{0,1\}^n$, $U|x\rangle = \sum_{y \in 0,1^n} U_{x,y}|y\rangle$. To completely define a linear operator, it suffices to define its action on every basis vector.

Definition 11.3. U is 2-local if there exists a 4×4 matrix $A = \alpha_{ij}$ such that, for example, writing $x = x_1 x_2 z$ where z is a 5-bit string, $U|x\rangle = U|x_1 x_2 z\rangle = \alpha_{00}|00z\rangle + \alpha_{01}|01z\rangle + \alpha_{10}|10z\rangle + \alpha_{11}|11z\rangle$. There is a notational abuse that is often used, to write this as $(A|x_1 x_2\rangle)|z\rangle$. This operates on the first two qubits; in general we can do this on any two bits. More generally U is k-local if we do this for any k qubits, $k = O(1)$. ◇

But just as without loss of generality we can assume that gates of constant fan-in can just have gates of fan-in 2, here for k-local we can assume that it is 3-local.

The only thing left is how to decide which operators to apply. So at every step there is a standard Turing Machine algorithm that comes up with this 4×4 matrix, or otherwise there are issues of uniformity. So we can view this as there is a polynomial time DTM that given an input n gives a sequence of 4×4 matrices that define the operators.

Lecture 19, 9 Apr

Definition 11.4. $L \in \mathsf{BQP}$ (Bounded-error Quantum Polynomial-Time) if there is a poly(n) time quantum algorithm such that $x \in L \implies \Pr[\text{accept}] \geqslant \frac{2}{3}$ and $x \notin L \implies \Pr[\text{accept}] \leqslant \frac{1}{3}$. ◇

As we will see this includes classical computing and also BPP as a special case. We will look at a specific quantum algorithm, called Grover's Search Algorithm.

11.2. Building Blocks of Quantum Algorithms

In most quantum algorithms, the basic building block is the Fourier Transform, on the boolean unit hypercube, on $\mathbb{Z}_p$, or even other structures. We will show that the operation of Fourier Transform in the boolean hypercube setting can be done efficiently.

Recall that given a function $f : \{0,1\}^n \to \{0,1\}$, we can write the function as $f = \sum_{S \subseteq \{1,\ldots,n\}} \hat{f}(s) \cdot (-1)^{\sum_{i \in S} x_i}$, where $\hat{f}(s)$ is the *Fourier coefficient*.

Definition 11.5 (Fourier Transform[1]). A *Fourier transform* is a unitary operator H where for $x \in \{0,1\}^n$, $H|x\rangle = \frac{1}{\sqrt{N}} \sum_{y \in \{0,1\}^n} (-1)^{\langle x,y \rangle}|y\rangle$ where $\langle x,y \rangle = \sum_{i=1}^n x_i y_i$. ◇

[1]Also known as Hadamard Transform or Walsh Transform

Note the particular case where $n = 1$, where we just have one bit. Then $N = 2$, and we have two basis states, $|0\rangle$ and $|1\rangle$. Then $H|0\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ and $H|1\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}$. So we can write $H|b\rangle = \frac{1}{\sqrt{2}}|0\rangle + (-1)^b\frac{1}{\sqrt{2}}|1\rangle$.

FACT 11.6. *The Fourier Transform on n qubits is computed as a 1-qubit Fourier Transform applied on bits 1, 2, 3, ... successively.*

So the Fourier Transform is the same as applying a sequence of n 1-local operators.

Note that the application of the 1-qubit Fourier Tranform on the i-th qubit, $H_i|x_1x_2\dots x_ix_{i+1}\dots x_n\rangle = \frac{1}{\sqrt{2}}|x_1x_2\dots 0x_{i+1}\dots x_n\rangle + \frac{1}{\sqrt{2}}|x_1x_2\dots 1x_{i+1}\dots x_n\rangle$.

PROOF OF FACT 11.6. We prove by induction that for the partial sequence $H_j\cdots H_2H_1|k\rangle = \frac{1}{2^j}\sum_{y_1,\dots,y_j\in\{0,1\}}(-1)^{\sum_{i=1}^j x_iy_i}|y_1\dots y_jx_{j+1}\dots x_n\rangle$. What we want is to plug in $j = n$.

So assume this is true for j, now we prove it for $j+1$: we will take the sequence

$$H_{j+1}H_j\dots H_2H_1|x\rangle = \frac{1}{\sqrt{2^j}}\sum_{y_1,\dots,y_j\in\{0,1\}}(-1)^{\sum_{i=1}^j x_iy_i}H_{j+1}|y_1\dots y_jx_{j+1}\dots x_n\rangle$$

We know that

$$H_{j+1}|y_1\dots y_jx_{j+1}\dots x_n\rangle = \frac{1}{\sqrt{2}}(|y_1\dots y_j0\dots x_n\rangle + (-1)^{x_{j+1}}|y_1\dots y_j1\dots x_n\rangle)$$

Plugging that in, we get

$$H_{j+1}H_j\dots H_2H_1|x\rangle = \frac{1}{2^{j+1}}\sum_{y_1,\dots,y_{j+1}\in\{0,1\}}(-1)^{\sum_{i=1}^{j+1} x_iy_i}|y_1\dots y_jy_{j+1}x_{j+2}\dots x_n\rangle$$

□

Now note that $H|0\rangle = \frac{1}{\sqrt{N}}\sum_{y\in\{0,1\}^n}|y\rangle$. This is a uniform super-position. Furthermore note that $H^2 = \text{Id}$, since

$$\begin{aligned} H^2|x\rangle &= \tfrac{1}{\sqrt{N}}\textstyle\sum_{y\in\{0,1\}^n}(-1)^{\langle x,y\rangle}\tfrac{1}{\sqrt{N}}\sum_{z\in\{0,1\}^n}(-1)^{\langle y,z\rangle}|z\rangle \\ &= \tfrac{1}{N}\textstyle\sum_{z\in\{0,1\}^n}\left(\sum_{y\in\{0,1\}^n}(-1)^{\langle x+z,y\rangle}\right)|z\rangle \\ &= |x\rangle \end{aligned}$$

In particular, $H(\frac{1}{\sqrt{N}}\sum_{y\in\{0,1\}^n}|y\rangle = |0\rangle$.

PROPOSITION 11.7. *Every classical logical gate can be simulated by a quantum gate.*

PROOF. Take the AND gates that has fan-in 2, and given input x and y outputs $x\wedge y$. The quantum version has fan-in 3 and given input x and y with z as a dummy bit, outputs x, y, and $(x\wedge y)\oplus z$. This is an invertible operation that simply acts by permuting.

So the simulation uses z as a work bit, so $U|xy\rangle|0\rangle = |xy\rangle|x\wedge y\rangle$. □

FACT 11.8. $\mathsf{P} \subseteq \mathsf{BQP}$.

FACT 11.9. $\mathsf{BPP} \subseteq \mathsf{BQP}$.

Proof. Suppose we take a randomized algorithm with input size n and r random bits. So we have $|x_{\text{input}}\rangle\,|0\dots0\rangle$, then after a series of operations we generate the randomness $|x_{\text{input}}\rangle\,(\frac{1}{\sqrt{2^r}}\sum_{y\in\{0,1\}^r}|y\rangle)$, so then we can simulate the deterministic algorithm by a quantum algorithm. □

Another block is the reflection operator.

Definition 11.10. The *Reflection operator* R sends $R\,|0\dots0\rangle = |0\dots0\rangle$, and $R\,|x\rangle = -\,|x\rangle$ for $x \neq 0\dots0$. ◇

This is reflection in $|0\rangle$.

Definition 11.11. Denote $\phi_{\text{uniform}} = \frac{1}{\sqrt{N}}\sum_{y\in\{0,1\}^n}|y\rangle$. ◇

Proposition 11.12. *The operator HRH is reflection in $\phi_{uniform}$.*

Proof. Given any state ϕ, write $\phi = \alpha\phi_{\text{uniform}} + \beta\phi_\perp$, where $|\alpha|^2+|\beta|^2 = 1$ and $\langle\phi_{\text{uniform}}|\phi_\perp\rangle = 0$, that is, they are orthogonal. Then $H\phi = \alpha\,|0\rangle + \beta H\phi_\perp$. So now $\langle 0|H\phi_\perp\rangle = 0$. So now if we apply R, $RH\phi = \alpha\,|0\rangle - \beta H\phi_\perp$. Then when we apply H again, then $HRH\phi = \alpha\phi_{\text{uniform}} - \beta\phi_\perp$. □

11.3. Grover's Search Algorithm

Suppose we have boxes $0,\dots,N$, where $N = 2^n$. All boxes contain 0, except for one, which contains 1. The goal is to find the box containing 1. Classically, we basically have to open every box to open every box, which takes N queries. After defining what query means in the quantum world, we will see that we only need $O(\sqrt{N})$ queries.

Now note that each of these boxes is indexed by a bit-string of length n, then we have a function $f : \{0,1\}^n \to \{0,1\}$ with the promise that there is a unique input x_0 such that $f(x_0) = 1$, and the goal is to find x_0.

Now the *quantum query model* is as follows: we have a unitary operator Q such that $Q\,|x\rangle = |x\rangle$ for $x \neq x_0$, and $Q\,|x_0\rangle = -\,|x_0\rangle$. So in general,

$$Q\left(\sum_{x\in\{0,1\}^n}\alpha_x\,|x\rangle\right) = \sum_{x\in\{0,1\}^n}\alpha_x(-1)^{f(x)}\,|x\rangle$$

So a *quantum query algorithm* starts in an arbitrary state, and then performs a sequence of unitary operators, then ask a query, then another sequence of unitary operators, ask a query, and so on for some number of times, and then output a string.

Grover's Algorithm is as follows: start with ϕ_{uniform} and perform $(HRHQ)$ $\left\lfloor\frac{\pi}{4}\sqrt{N}\right\rfloor$ times, and then measure. Then the output will be x_0 with high probability.

Now ϕ_{uniform} is a vector on the two-dimensional plane spanned by $|x_0\rangle$ and $\psi = \frac{1}{\sqrt{N-1}}\sum_{x\neq x_0}|x\rangle$. Let the angle between ϕ_{uniform} and ψ be denoted by θ, and we see that θ is very small. As we will see, the operation $HRHQ$ simply rotates the vector v bit by bit in the direction of the $|x_0\rangle$ vector, and then we find $|x_0\rangle$. Lecture 20, 14 Apr

Fact 11.13. *Q is a reflection in ψ on this two-dimensional plane.*

Proof. $Q(\alpha\psi + \beta\,|x_0\rangle) = \alpha\psi - \beta\,|x_0\rangle$, since $f(\psi) = 0$. □

So let v denote the current state, which starts with $v = \phi_{\text{uniform}}$, so that the angle between v and ψ is $\alpha + \theta$. Then Qv is a vector below ψ with angle $\alpha + \theta$, and then HRH rotates Qv across ϕ_{uniform}, which gives $HRHQv$ as a vector with angle $\alpha + 2\theta$ from ψ. So $HRHQ$ rotates v by angle 2θ.

Now note that $\cos\theta = \langle \phi_{\text{uniform}} | \psi \rangle \approx 1 - \frac{1}{2N}$, so $\theta \approx \frac{1}{\sqrt{N}}$. Then since we need to rotate the vector by a total angle of $\frac{\pi}{2}$, the number of steps is $\frac{\pi/2}{2\theta} = \frac{\pi}{4}\sqrt{N}$. So the final state is $0.999\,|x_0\rangle + 0.001\psi$, so it outputs x_0 with probability 0.999.

This generalizes to when there is more than one vector x on which $f(x) = 1$.

In fact, it is not clear that $\frac{\pi}{4}\sqrt{N}$ is the right number, since it is not an integer and we had some approximations in our analysis. However, it is possible to do something clever and rotate by the correct angle instead of 2θ in order to reach $|x_0\rangle$ with probability 1.

11.4. Shor's Algorithm

Let us look at a problem known as the *hidden subspace problem*. This problem is similar to the search problem, but not quite. We are given query access to a function $f : \{0,1\}^n \to \mathcal{L}$, such that there is a subspace $H \subseteq \{0,1\}^n$ such that f is constant on each coset of H and it is distinct on distinct cosets.

So for example, we can imagine that H is a hyperplane of dimension $n-1$, then f could be a coloring on the whole space, red for on the hyperplane and blue otherwise.

The goal is to find H, that is, the basis that defines it.

The *quantum query model* Q now needs to be more sophisticated than the one used for Grover's Algorithm. Even before, we could have defined Q as $Q\,|x\rangle\,|b\rangle = |x\rangle\,|b \oplus f(x)\rangle$, where b is a single dummy bit. Now the number of states is doubled, but what we really care about is when $b = 0$ since then $Q\,|x\rangle\,|0\rangle = |x\rangle\,|f(x)\rangle$. These two definitions are equivalent. Now we no longer restict the output of f to be a single bit, so we just have b have as many bits as needed.

The algorithm is as follows: we start with $v = \frac{1}{\sqrt{N}}\sum_{x\in\{0,1\}^n} |x\rangle\,|0\rangle$. We can think of this as starting with two registers, both all zero, and then applying the Fourier transform to the first register. If we apply Q then we get $Qv = \frac{1}{\sqrt{N}}\sum_{x\in\{0,1\}^n} |x\rangle\,|f(x)\rangle$, and then we measure the second register[2]. Normally when we measure, we collapse the entire state, but here we collapse the state to something that is only in the first register.

So the state will collapse to something that is only in the coset corresponding to the color, that is, $v = \frac{1}{\sqrt{|H|}}\sum_{x\in y_0+H} |x\rangle$ for a random coset $y_0 + H$.

Now apply to Fourier transform, this gives

$$\begin{aligned} v &= \frac{1}{\sqrt{|H|}}\sum_{x\in y_0+H}\left(\frac{1}{\sqrt{N}}\sum_{z\in\{0,1\}^n}(-1)^{x\cdot z}\,|z\rangle\right) \\ &= \frac{1}{\sqrt{|H|N}}\sum_{z\in\{0,1\}^n}\left(\sum_{x\in y_0+H}(-1)^{x\cdot z}\right)|z\rangle \\ &= \frac{1}{\sqrt{|H|N}}\sum_{z\in H^\perp}|H|(-1)^{y_0\cdot z}\,|z\rangle \\ &= \frac{1}{\sqrt{|H^\perp|}}\sum_{z\in H^\perp}(-1)^{y_0\cdot z}\,|z\rangle \end{aligned}$$

Then taking a measurement gives output $z \in H^\perp$ uniform randomly.

[2]There is some technicality here, since the first and second registers are entangled.

Once we do this enough times, we get a full basis for $H^{\perp}$, and thus a basis for H itself. The number of times is $O(n)$.

This can be generalized to the problem of hidden subgroup over abelian groups. The big open problem is to do this over non-abelian groups.

The following is known:

(1) If we can solve this for S_n then we have an efficient quantum algorithm for graph isomorphism.

(2) If we can solve this for the dihedral group then we have an efficient quantum algorithm to $n^{O(1)}$-approximate computation of the *Shortest Vector Problem in Lattices*. There are cryptographic systems that rely on the assumption that this problem is difficult to solve.

(3) In particular, we can take the abelian group $\mathbb{Z}$, and then *Shor's Factoring Algorithm* is more or less solving the Hidden Subspace Problem. There are caveats: the main problem is that $\mathbb{Z}$ is infinite, so we need to truncate, but then it ceases to be a group.

CHAPTER 12

Communication Complexity

Lecture 21, 16 Apr

We start with an example, the problem of *equality.*

Alice and Bob both have inputs, Alice with $x \in \{0,1\}^n$ and Bob with $y \in \{0,1\}^n$. Alice and Bob want to determine if $x = y$, and they can only communicate over a channel by exchanging bits. The goal is to decide if $x = y$, and to minimize the amount of communication.

We already saw communication in Interactive Proofs, but the setting here is completely different. Here both parties are completely honest and want to solve a particular problem, and both must know the answer. Secondly, there is no notion of computation time, as such. Alice and Bob can each decide what they want based on the input and the communication. So only the amount of communication is the important measure.

First observe that $n + 1$ bits of communication is enough, because Alice can just send her entire input to Bob, and Bob knows both x and y, and then can just communicate the answer back to Alice.

We will see shortly that this is necessary:

THEOREM 12.1. *Any (deterministic) communication protocol for equality must exchange at $\geqslant n + 1$ bits.*

THEOREM 12.2. *There is a randomized $O(\log n)$ communication complexity protocol for equality.*

PROOF. This is based on the following simple fact: Alice and Bob can think of their strings as integers between 0 and 2^n. Now if two numbers are equal, then they are equal modulo a prime.

So the protocol is Alice picks a random prime p with $\ell = O(\log n)$ bits, and sends p and $t \leftarrow x \mod p$ to Bob. Bob checks if indeed $t = y \mod p$.

This is a correct protocol insofar as if $x = y$ then Alice and Bob will accept with probability 1; on the other hand, they will reject with high probability. We just need to show that if $x \neq y$ then $x \mod p \neq y \mod p$ for 99% of primes with $\leqslant \ell$ bits. This simply follows from that the primes are dense enough: if x and y are equal to modulo more than 1% of the primes, then they are equal to their product, and then we can show that x and y must exceed 2^n. □

Note that we made no analysis about the efficiency of generating the primes or computing modulo, but these are irrelevant from the perspective of Communication Complexity.

12.1. Yao's Deterministic Model

In *Yao's Model of (deterministic) Communication Complexity*, Alice and Bob have inputs $x \in \{0,1\}^n$ and $y \in \{0,1\}^n$, respectively, and given a function f :

$\{0,1\}^n \times \{0,1\}^n \to \{0,1\}$, they want to compute $f(x,y)$. They can exchange bits, and at the end both should declare the correct value of the function. A *protocol* is simply a specification as to, in each round, whether Alice sends a bit or Bob sends a bit, and how that bit is computed. Without loss of generality, we may assume one bit gets sent in each round and they alternate with Alice sending first, so the protocol is $b_1 = h(x)$, $b_2 = g(y, b_1)$, $b_3 = h'(x, b_1, b_2)$, …, and at the end they output $f(x,y)$.

DEFINITION 12.3. The *deterministic communication complexity* of f, $D(f)$ is the amount of communication needed to compute f deterministically. $\diamond$

FACT 12.4. $D(f) \leqslant n+1$.

This is because there is the trivial protocol where Alice sends the entire input to Bob and Bob computes. We will see that in some cases this is nearly optimal.

Definitely in this business given f we want to find more and more efficient methods to compute f, but as it turns out, much of the research activity focused on proving lower bounds. We will see later why this is so. The main reason is that the main application of communication complexity is to prove lower bounds in other models of computation.

For example, we will show eventually the following: in the Turing machine model, consider the problem of checking if a string is a palindrome. We will show using communication complexity that any algorithm using the Turing machine model must use $\Omega(n^2)$ steps.

Now let M_f be the 0-1 matrix of size $2^n \times 2^n$ such that the x, y entry is $f(x,y)$.

Suppose there is a k-bit protocol that computes f, and look at the matrix for the first bit. We get a partition of this matrix into the rows where Alice sends 0, and the rows where Alice sends 1. At the end of this phase of communication, Alice and Bob have restricted their input space to half of the matrix, without loss of generality the upper rectangle or the lower rectangle. Suppose without loss of generality that we know that they are in the upper rectangle. Then Bob sends a bit, which partitions the rectangle into the left or the right rectangle.

So the observation is that if we have this k-bit protocol, we have this partition of the space into an area that is constant on the communication. So a k-bit deterministic protocol implies that M_f can be partitioned into at most 2^k "monochromatic rectangles".

DEFINITION 12.5. A *rectangle* is $S \times T$, $S \subseteq \{0,1\}^n$, $T \subseteq \{0,1\}^n$. $\diamond$

DEFINITION 12.6. A rectangle $S \times T$ is *monochromatic* if $f(x,y) = 0$ for all $(x,y) \in S \times T$ or $f(x,y) = 1$ for all $(x,y) \in S \times T$. $\diamond$

This makes it easy to prove these lower bounds. A priori, there is no way to prove a result like Theorem 12.1, since we have no control over the number of protocols, but once we have these observations, it becomes easy.

PROOF OF THEOREM 12.1. We just use the observation that if M_f cannot be partitioned into $\leqslant 2^k$ monochromatic rectangles then we need at least k bits.

The matrix $M_{\text{EQ}} = I$. Now it is clear that no rectangle can include two of the 1s on the diagonal, since we would have to include the 0s as well. So the rectangles that cover the 1 entries must be the trivial 1×1 rectangles. Then we need at least one more rectangle to cover the zeroes, so the number of monochromatic rectangles is $\geqslant 2^n + 1$, then $D(\text{EQ}) \geqslant \lceil \log_2 2^n + 1 \rceil = n + 1$. $\square$

Using the same method, we can prove a few more lower bounds:

Definition 12.7. Let the inner product function be defined as $\mathrm{IP}(x,y) = \sum_{i=1}^{n} x_i y_i \pmod 2$. $\diamond$

Theorem 12.8. *$D(\mathrm{IP}) \geqslant n-1$.*

Proof. We will focus on the 0 entries. One way to show the result will be to show that the matrix contains many 0 entries, but each of the rectangles are small, so we will need many rectangles.

So for the first part, we claim that the number of 0 entries in this matrix is $\geqslant 2^n 2^{n-1}$, since if we pick two vectors at random their inner product is 0 or 1 with almost equal probability, with 0 slightly higher.

Now we claim that every monochromatic 0-rectangle has size $\leqslant 2^n$. Then we are done, since then the number of rectangles needed is at least $\frac{2^n 2^{n-1}}{2^n}$, then we just take the log of that.

Now take a monochromatic rectangle $S \times T$, such that $x \cdot y = 0$ for $x \in S$, $y \in T$. So without loss of generality we may assume S and T to be subspaces of the vector space $\{0,1\}^n$, since if two vectors are in one of the sets, then their sum is in the set as well. So S and T are orthogonal subspaces, and it is a basic fact from Linear Algebra that $\dim(S) + \dim(T) \leqslant n$. Then $2^{\dim(S)} 2^{\dim(T)} \leqslant 2^n$, so $|S||T| \leqslant 2^n$. $\square$

Definition 12.9. The *disjointness function* is $f(x,y) = \begin{cases} 1 & x \cap y = \emptyset \\ 0 & \text{otherwise} \end{cases}$, viewing x and y as characteristic strings of inclusion in a set. So the sets are disjoint if there is no i such that x_i and y_i are both 1. $\diamond$

Theorem 12.10. *$D(\text{DISJOINTNESS}) \geqslant n$.*

Proof. The method used is known as a fooling set method.

Consider $\Phi = \{(x, \overline{x}) \mid x \in \{0,1\}^n\}$. Clearly $f = 1$ on all pairs in Φ.

The claim is that we cannot have two elements of Φ in the same monochromatic rectangle.

Suppose there is a monochromatic rectangle that contains $(x, \overline{x})$ and $(z, \overline{z})$. This rectangle also contains $(x, \overline{z})$ and $(z, \overline{x})$. Now one of the two must be 0, since $x \neq z$, contradicting that the rectangle is monochromatic. $\square$

There is another technique, which looks at the rank of characteristic matrices.

Observation 12.11. If the deterministic communication complexity is $\leqslant k$, then $\mathrm{rank}(M_f) \leqslant 2^k$ over any field. $\diamond$

Proof. We can write $M_f = A_1 + \ldots + A_{2^k}$ corresponding to the partitions: A_i is 1 on some monochromatic rectangle and 0 everywhere else. Then A_i is of rank at most 1, so $\mathrm{rank}(M_f) \leqslant \sum_{i=1}^{2^k} \mathrm{rank}(A_i) \leqslant 2^k$. $\square$

Fact 12.12. *$D(f) \geqslant \Omega(\log(\mathrm{rank}(M_f)))$.*

A very famous open conjecture in Communication Complexity is as follows:

Conjecture 12.13. *$D(f) \leqslant O(\log(\mathrm{rank}(M_f))^c)$ for some constant c and for all f.*

Note that for the three proofs we gave, we can give alternate proofs using the rank.

12.2. Randomized Communication Complexity

For now, let us say that the randomness is private, then we want to design a protocol such that matter what the input, we output $f(x, y)$ with high probability.

So say we fix the error parameter ε with $0 < \varepsilon < \frac{1}{3}$, and Alice and Bob hold inputs x and y respectively, and they want to compute $f(x, y)$. They have private randomness r_A and r_B. Now they exchange bits, except that the bits can depend on the choice of randomness: $b_1 = h(x, r_A)$, $b_2 = g(y, r_B, b_1)$, $b_3 = h'(x, r_A, b_1, b_2)$, etc. Since the randomness is private it does not matter if the randomness is fixed at the beginning or refreshed at every round. At the end they want to be correct with probability $\geqslant 1 - \varepsilon$.

Definition 12.14. The *private randomized communication complexity* $R_\varepsilon(f)$ is the minimum communication required to compute $f(x, y)$ correctly with probability $\geqslant 1 - \varepsilon$ via a private coin protocol on every input pair x, y. ◇

Similarly we can have the notion of a public coin protocol, where there is a random string r that both of them can see.

Definition 12.15. The *public randomized communication complexity* $R_\varepsilon^{\text{pub}}(f)$ is the minimum communication required to compute $f(x, y)$ correctly with probability $\geqslant 1 - \varepsilon$ via a public coin protocol on every input pair x, y. ◇

As we already showed, for equality, a private $O(\log n)$ communication is enough.

On the other hand, with a public coin protocol, only $O(\log(1/\varepsilon))$ bits are needed: take the public randomness r to be an n-bit string, and then Alice and Bob just take the inner product of their strings with r, and compare the result.

So the question is what is the difference between private and public coin protocols. As it turns out this is more or less the worst possible gap:

Fact 12.16. *$R_\varepsilon^{pub}(f) \leqslant R_\varepsilon(f) \leqslant R_{\varepsilon/2}^{pub}(f) + O(\log n + \log(1/\varepsilon))$ for all f.*

Proof. The first part is a triviality.

Suppose there is a k-bit public coin protocol. Fix the inputs x, y, then over the choice of r the protocol computes $f(x, y)$ with probability $\leqslant 1 - \varepsilon$. Now we claim that we can fix a small number of random strings, $\tilde{S} = \{r_1, r_2, \ldots, r_{\ell=\text{poly}(n,1/\varepsilon)}$ such that we still get $f(x, y)$ on $\geqslant 1 - 2\varepsilon$ fraction of the sample with probability $1 - 2^{-\ell}$. Then we can take a union bound over all choices of x, y. Then Alice and Bob can just agree on this set to begin with, and then Alice can pick one at random, tell Bob, and then they can act as if they had a public coin protocol. This part takes $O(\log \ell)$ bits. □

Lecture 22, 21 Apr

We saw several techniques to prove lower bounds for deterministic communication complexity. But since here we allow errors, we cannot exactly use monochromatic rectangles. The corresponding object of interest are rectangles that are approximately monochromatic.

One way to prove a lower bound is to upper bound the discrepancy of a function.

Definition 12.17. Let $R = S \times T$ be a rectangle of M_f, then the *measure* $\mu(R) = \frac{|S||T|}{2^{2n}}$, the size of R relative to the size of M_f. The *discrepancy* $\text{disc}(R, f) = \mu(R)|1 - 2\delta|$, where δ is the fraction of 0s in R. Equivalently, we may define $\text{disc}(R, f) = |\Pr_{(x,y)}[(x, y) \in R \wedge f(x, y) = 0] - \Pr_{(x,y)}[(x, y) \in R \wedge f(x, y) = 1]|$. The discrepancy of a function $\text{disc}(f) = \max_R \text{disc}(R, f)$. ◇

So having a low discrepancy amounts to every rectangle has low discrepancy, so either every large rectangle is nearly balanced.

THEOREM 12.18. $R_\varepsilon(f) \geqslant \Omega(\frac{1}{\log(\operatorname{disc}(f))})$.

That is, if a function has low discrepancy, it requires a large communication complexity. Recall that a c-bit deterministic protocol partitions M_f into 2^c monochromatic rectangles. So low discrepancy means that every monochromatic rectangle is small, so we need high communication complexity. We massage this to make it robust for randomization.

PROOF. Suppose we have a c-bit randomized protocol with ε-error on every fixed input (x, y). It follows that there is a deterministic protocol that succeeds on $1-\varepsilon$ fraction of the inputs: we fix the randomization to r_0 such that $\Pr_r \Pr_{(x,y)}[P(x,y) = f(x,y)] \geqslant 1-\varepsilon$. So call this protocol P_{r_0}. Now we have a matrix A represeneting P_{r_0}.

Now since we have a c-bit protocol, this partitions A into 2^c monochromatic rectangles. Now A does not necessarily equal M_f, but A agrees with M_f on $1-\varepsilon$ fraction of inputs. Now suppose we impose the same partition from A onto M_f. Look at one of these rectangles.

Suppose $\mu(R)$ is large, say $\mu(R) \geqslant 50 \cdot \operatorname{disc}(f) \geqslant 50 \cdot \operatorname{disc}(f) = 50\mu(R)|1-2\delta|$, so $\delta \in [\frac{1}{2} \pm \frac{1}{100}]$. Then since we require the overall error to be small, the number of rectangles that are small must be at least $\frac{\frac{1}{2}}{50\operatorname{disc}(f)}$, so c must be at least log of this. ☐

THEOREM 12.19. *The randomized communication complexity of the inner product function* $R_\varepsilon(\mathrm{IP}) \geqslant \Omega(n)$.

PROOF. We show that $\operatorname{disc}(\mathrm{IP}) \leqslant 2^{-n/2}$.

Let H be the $2^n \times 2^n$ matrix of IP, used in Fourier transforms.

We will need three facts:

First, $H^2 = 2^n I$: $H^2_{x,y} = \sum_z H_{x,z}H_{z,y} = \sum_z (-1)^{x\cdot z}(-1)^{z\cdot y} = \sum_z (-1)^{(x+y)\cdot z}$. Then we just use the fact that if $x \neq y$ then we get $+1$ and -1 with equal probability over z, so the sum vanishes, but if $x = y$ then we get 1.

Next, the eigenvalues of H are all equal to $\pm 2^{n/2}$. Since H is symmetric, we can write $H = ODO^T$ where O is orthogonal and D is a diagonal matrix of its eigenvalues. Then $H^2 = ODO^TODO^T = OD^2O^T$. Then since we know that the entries of D^2 are all 2^n, the result follows.

Third, define the norm of a matrix $\|A\| = \max_t \neq 0\frac{\|At\|}{\|t\|}$, that is, it is the maximum ratio by which it can "stretch" the length of a vector t. Then $\|A\| = |\lambda_0|$ where λ_0 is the largest eigenvalue of A. So $\|H\| = 2^{n/2}$.

Now suppose $S \times T$ is a rectangle. Now we know that

$$\begin{aligned}\operatorname{disc}(S\times T, \mathrm{IP}) = |&\Pr_{x,y}[(x,y) \in S \times T \wedge (-1)^{x\cdot y} = +1] \\ &- \Pr_{x,y}[(x,y) \in S \times T \wedge (-1)^{x\cdot y} = -1]|.\end{aligned}$$

We can rewrite this as $\operatorname{disc}(S\times T, \mathrm{IP}) = |\frac{1}{2^{2n}}\sum_{x,y} \mathbb{1}_S(x)\mathbb{1}_T(y)H(x,y)|$, where $\mathbb{1}_S$ is the indicator row vector, which is 1 for inputs corresponding elements in S, and 0 otherwise. So more compactly, $\operatorname{disc}(S \times T, \mathrm{IP}) = \frac{1}{2^{2n}}|\mathbb{1}_S H \mathbb{1}_T^T|$. But this is at most $\frac{1}{2^{2n}}\|\mathbb{1}_S\|\,\|H\mathbb{1}_T^T\|$, which is upper bounded by $\frac{1}{2^{2n}} \cdot 2^{n/2} \cdot 2^{n/2} \cdot 2^{n/2} = 2^{-n/2}$. ☐

A lower bound for set disjointness is much more difficult. The technique is the same, but the measure $\mu(R)$ is different.

12.3. Applications of Communication Complexity Lower Bounds

We show some nice complexity theoretic lower bounds that come from communication complexity lower bounds.

12.3.1. Time-Space Tradeoff for Turing Machines.

THEOREM 12.20. *Take the Turing machine model with a read-only input tape of length n, and $O(1)$ work tapes, each of size $S(n)$, then $S(n)$ is the space requirement. Let $T(n)$ be the time requirement. For deciding if the input is a palindrome, $T(n) \cdot S(n) \geqslant \Omega(n^2)$.*

Note that this theorem is tight, in that it is possible to have $T(n) = S(n) = O(n)$, by copying the input in reverse and then comparing character by character, so we get a product of $O(n^2)$. Alternatively, we can have $T(n) = O(n^2 \log n)$ and $S(n) = O(\log n)$, by just shuttling back and forth on the input tape but using the work tape as a position record, then ignoring log factors we again get $O(n^2)$.

PROOF. We will show that any algorithm to decide palindromes with space requirement $S(n)$ and time requirement $T(n)$ can be used to design a deterministic communication protocol to decide equality, with communication $O\left(\frac{T(n)}{n} S(n)\right)$, which we know from before must be $\geqslant \Omega(n)$.

Now suppose Alice has x and Bob has y, then $x = y$ if and only if the following holds: construct the string of length $3n$ $x0^ny^r$ where y^r of y, then this string is a palindrome.

So now Alice and Bob are now trying to decide if $x0^ny^r$ is a palindrome. So Alice can just simulate the algorithm for palindrome on this string until she needs a value from y^r. Then Alice will stop, then send the contents of the work tape to Bob, and then Bob will continue the simulation until he needs a value from x. Then Bob will stop, then send the contents of the work tape to Alice, and Alice will continue. Then they can just continue in this manner until the machine stops, and then they know the answer.

So clearly this is a correct protocol to decide equality, and the amount of communication needed is $\leqslant O(S(n))\frac{T(n)}{n}$, where $\frac{T(n)}{n}$ is the total number of transfers: the machine has at most $T(n)$ steps, and each transfer is separated by at least n steps, since the machine must cross 0^n before requiring a transfer. $\square$

12.3.2. Data Streaming Algorithms. The *data streaming model* is a model of computation that came about when the amount of data used in computers became too large to store in memory. In the model, the data comes as a stream of items $a_1, a_2, \ldots, a_N$ from $\mathbb{Z}$ with bit-size at most $O(\log N)$, and we are very limited in that we only have poly$(\log(N))$ memory. The question is what can we compute, possibly with randomization or approximation?

There are certainly some trivial things we can compute, like $\sum_{i=1}^N a_i$, since we only ever need to maintain the partial sum. Similarly, we can compute $\max_i a_i$.

What is surprising is that we can compute the following: the number of distinct integers/items, with high accuracy.

Lecture 23, 23 Apr

Now let f_j be the frequency of integer j, for $j \in \{1, \ldots, R\}$. Then let the frequency moment $F_k = \sum_{j=1}^{R} f_j^k$ for $k = 0, 1, 2, \ldots$. By convention we let $0^0 = 0$, so that F_0 is the number of distinct elements in the sequence. It turns out that the moments $F_0, F_1, \ldots$ can be computed with $1 \pm \varepsilon$ multiplicative approximation with high probability in $\mathrm{poly}(\log N)$ space.

THEOREM 12.21. *To compute F_k for $k \geqslant 3$ within a factor of 10, we require space $\tilde{\Omega}(N^{1-\frac{2}{k}})$.*

We need a *t-party communication model*, where instead of Alice and Bob we have Player 1 with input x_1, Player 2 with input x_2, …, Player t with x_t, and they wish to compute $f(x_1, x_2, \ldots, x_t)$ while minimizing communication. There are multiple models: the general model where everyone takes turns speaking; the simultaneous model where each player speaks only once, individually and independently, and then a referee looks at all of these and decides the output; and the 1-way model. We will only use the general model.

Even at the general model, there are two cases: the number-in-hand model, in which each player can only see their own number; and the number-on-forehead model, in which each player can see everyone else's number, but not their own. We will stick to the number-in-hand model.

We now prove a lower bound for set-disjointness in the t-party number-in-hand model. The input is a sequence of sets $S_1, S_2, \ldots, S_n \subseteq [n]$. The promise is that either all the sets are all pairwise disjoint (YES case), or they all intersect on one point: $\exists i_0 \in [n], S_i \cap S_j = \{i_0\} \forall i \neq j$ (NO case).

THEOREM 12.22. *Any random communication protocol for t-parity number-in-hand set-disjointness needs $\Omega(\frac{n}{t \log t})$ bits of communication.*

Note that there is a $O(\frac{n}{t} \log n)$-bit protocol in the simultaneous model: on average, each player has a set of size $\frac{n}{t}$, and then say they pick a set of size $\frac{n}{t^2}$ and communicates it. This takes $\frac{n}{t^2} \log n \times t = \frac{n}{t} \log n$ bits. We accept if and only if all of these sets are pairwise disjoint. In the YES case, then obviously this will be true. In the NO case, the probability that i_0 is in each set is $\frac{1}{t}$, so with constant probability two players will both have i_0 in their sets.

PROPOSITION 12.23. *Theorem 12.22 $\Longrightarrow$ Theorem 12.21.*

PROOF. To see this we will show that a data streaming algorithm to compute F_k in space $S(n)$ gives a $S(n) \cdot t$-bit communication protocol for set-disjointness in the t-parity number-in-hand model.

We construct a data sequence where all the elements in S_1 arrive first, in any order, then all elements in S_2 arrive, and so on. Then player 1 can start running the algorithm on all inputs from S_1, then transfer the state of the machine to player 2, who continues the algorithm on all inputs from S_2, then transfers the state of the machine to player 3, and so on. Then player t computes F_k up to $1 \pm \varepsilon$ approximation.

Now note that in the YES case, $F_k \leqslant n$, since $f_j \leqslant 1$ for all $j \in [n]$. However on the NO, then $F_k \geqslant t^k$, so if we choose, for example, $t = 10n^{1/k}$ then there is a large separation between the YES case and the NO case.

So player t outputs the correct YES/NO answer.

But we know that the amount of communication is $\leqslant S(n) \times t$, but we showed that the amount is $\geqslant \Omega(\frac{n}{t \log t})$. Then since we chose $t = 10n^{1/k}$, we get $S(n) \geqslant \tilde{\Omega}(n^{1-\frac{2}{k}})$. □

CHAPTER 13

Probabilistically Checkable Proofs and the Hardness of Approximation

At first sight, these look like two ideas that are completely different and have nothing to do with each other, but in fact they are so closely related to be actually the same.

For the rest of this chapter we will assume that $\mathsf{P} \neq \mathsf{NP}$.

13.1. Approximation Algorithms

We need the definition of an *approximation algorithm*.

Consider the *Minimum Vertex Cover* problem where given a graph G, $\mathrm{OPT}(G)$ is the minimum size of a vertex cover. We know that this problem is NP-hard, but as it turns out we can easily find a vertex cover of at most $2\,\mathrm{OPT}(G)$. This factor of 2 is guaranteed. So this is a 2-approximation for Vertex Cover.

The algorithm is simple: we find a maximal matching M, which can be done in polynomial time. Now let T be the set of $2|M|$ vertices consisting of the both endpoints of each edge in M. Clearly T is a vertex cover. Now observe that $\mathrm{OPT}(G) \geqslant |M|$, since every vertex cover must take at least one endpoint from each edge in M. So $|T| \leqslant 2\,\mathrm{OPT}(G)$. Surprisingly, no one knows if it is possible to find a vertex cover of size $(2-\varepsilon)\,\mathrm{OPT}(G)$. There are now results that make it seem likely that we cannot do better.

Another problem is the *Minimum Set Cover* problem, where we have a universe U, with $|U| = n$, and we have sets $S_1, \ldots, S_m \subseteq U$ that cover U, then the goal is to find the minimal number of sets $S_{i_1} \cup S_{i_2} \cup \ldots S_{i_k} = U$. A $\ln n$-approximation algorithm for this is known. The way this works is to continually take the set that contains the largest number of uncovered items.

There is the *Minimum Bin Packing* problem, where we have $x_1, \ldots, x_n \in [0,1]$, and we need to pack them into size-1 bins. The goal is to minimize the number of bins. This problem has a more-or-less a $(1+\varepsilon)$-approximation[1] for any constant $\varepsilon > 0$. This is called a *polynomial-time approximation scheme (PTAS)*.

As it turns out, many packing and scheduling problems have a PTAS. Furthermore, Euclidean TSP has a PTAS.

Consider the *Maximum* $\mathsf{3}{-}\mathsf{SAT}$ problem: given $x_1, \ldots, x_n \in \{0,1\}$, and clauses $c_1, \ldots, c_m$ where each clause contains at most 3 literals. We want to maximize the number of clauses satisfied. A trivial known algorithm is a $\frac{7}{8}$-approximation. If we just assign the variables at random, then in expectation we satisfy $\frac{7}{8}$ of the clauses. We can derandomize and still get $\frac{7}{8}$, and clearly $\mathrm{OPT} \leqslant m$, so we are done.

[1]Actually, it is $(1+\varepsilon)\,\mathrm{OPT}+1$, so there are some caveats that are annoying.

The problem of *Maximum Clique* has a trivial algorithm where we just output one vertex, then this is a $\frac{1}{n}$-approximation[2].

This was more or less the state of knowledge until around 1990, mostly from the algorithmic side. People did not know if we could or could not do better. People wondered if we could get a PTAS for every NP-hard problem.

13.2. Hardness of Approximation

Then a series of papers that proved what is known as the PCP Theorem showed that for Maximum $3{-}\mathsf{SAT}$, Maximum Set Cover, and Maximum Clique there is a constant cutoff such that approximating better than this cutoff is NP-hard. So if we could compute such an approximation we may as well compute it exactly.

Now we know, for example, that for Maximum Clique we cannot do better than $n^{1-\varepsilon}$ for any $\varepsilon > 0$. For Maximum $3{-}\mathsf{SAT}$ we cannot do better than $\frac{7}{8} + \varepsilon$ for any $\varepsilon > 0$. For Maximum Set Cover we cannot do better than $(1 - \varepsilon) \ln n$.

These are known as *hardness of approximation results*, where if we assume $\mathsf{P} \neq \mathsf{NP}$ we cannot do any better. This gives us a starting position.

For Minimum Vertex Cover it is known that we cannot do better than 1.36, and there is evidence that we cannot do better than 2.

Another very good example is the *Maximum Cut* problem, for which there is a Semi-definite Programming algorithm that gives a 0.878-approximation algorithm, where 0.878 is the value of θ, $0 \leqslant \theta\pi$ that minimizes $\frac{\theta/\pi}{(1-\cos\theta)/2}$. There is evidence that this is the best possible.

So there is a list of problems for which we know what is the best possible. However, there are problems for which we know nothing at all. For example, the *Metric Traveling Salesman Problem* has a trivial $\frac{3}{2}$-approximation, but we have no harness result.

Lecture 24, 28 Apr

THEOREM 13.1. *The Traveling Salesman Problem*[3] *is* NP*-hard to approximate within a factor of 10.*

We give a reduction from an NP-hard problem to an instance of TSP, which is known as a *gap instance*, which is an instance where there is a promise that every YES instance reduces to an instance with a tour of length $\leqslant \ell$, and every NO instance reduces to an instance with a tour of length $\geqslant 10\ell$. So clearly if we can approximate within a factor of 10, we can distinguish YES from NO in polynomial time. Such reductions are called *gap reductions*

PROOF OF THEOREM 13.1. We reduce from the Hamiltonian Cycle problem, where we take a graph $G = (V, E) \rightsquigarrow H(V, wt)$ so that in the YES case, that is, G is Hamiltonian, then $\mathrm{OPT}(H) \leqslant n$, and in the NO case, that is, G is not Hamlitonian, then $\mathrm{OPT}(H) \geqslant 10n$.

The reduction is actually a triviality. So take G, and then H will be as follows: for every edge in G, this edge appears in H with weight 1. For every edge not in G, the edge appears in H with weight $10n$. Hence if G is Hamiltonian, we can have a tour of length at most n, and if G is not Hamiltonian, then any tour must contain an edge of weight $10n$.

So in fact the 10 is arbitrary, we could have picked anything we want. □

[2]In literature this is often written as a n-approximation, as a matter of convention.

[3]Here TSP is in its most general form.

Here is an example that is less trivial.

In the *edge-disjoint paths* problem, we are given a directed graph $G = (V, E)$ and pairs $(s_1, t_1), \dots, (s_\ell, t_\ell)$. The goal is to find paths $s_i \rightsquigarrow t_i$ such that all of these paths are edge disjoint, and to maximize the number of source-sink pairs that have edge-disjoint paths.

It is known that there is an $O(\sqrt{|E|})$-approximation, which is kind of terrible. We will show that this is in fact optimal.

The problem of $2-\mathsf{EDP}$ is a special case of Edge-Disjoint Paths where there are only two source-sink pairs. This problem is known to be NP-complete, although the proof might be tricky. So we will take this fact for granted.

THEOREM 13.2. *For every $\varepsilon > 0$, it is NP-hard to approximate EDP within a factor $O(m^{\frac{1}{2}-\varepsilon})$ where m is the number of edges.*

PROOF. Let H be an instance of $2-\mathsf{EDP}$, we will create a corresponding instance G of EDP with n source-sink pairs $(s_1, t_1), \dots, (s_n, t_n)$ with the property that if $H \in 2-\mathsf{EDP}$, then $\mathrm{OPT}(G) = n$, that is, it is possible to connect all source-sink pairs via edge-disjoint paths; otherwise if $H \notin 2-\mathsf{EDP}$, then $\mathrm{OPT}(G) = 1$, that is, it is NP-hard to find more than one path. When we stare at the construction, we will see that $n = \sqrt{|E|}$, so that is the gap.

Construct the graph G as in Figure 13.3, where at every intersection we have a copy of H as shown.

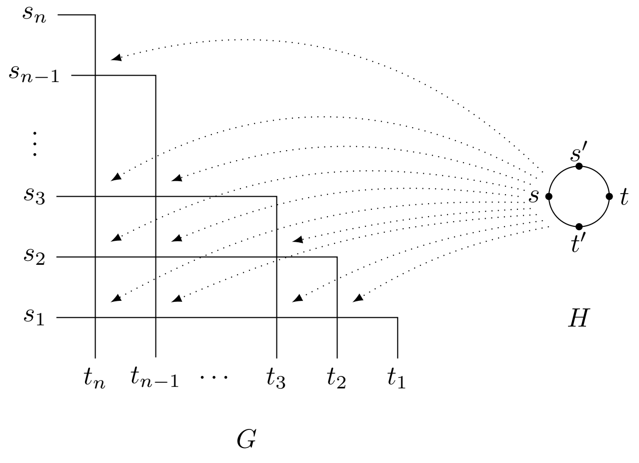


FIGURE 13.3. The graph G, where every intersection is a copy of H

So in the YES case, it is possible to find paths $s \rightsquigarrow t$ and $s' \rightsquigarrow t'$, and this gives natural edge-disjoint paths for each (s_i, t_i) pairs, and $\mathrm{OPT}(G) = n$.

The NO case takes a bit more work, but suppose we take $s_i \rightsquigarrow t_i$ and $s_j \rightsquigarrow t_j$, then we must go through an instance of H at their intersections. But in the NO case, this is impossible.

Now we just want to express n as a factor of the number of edges. There are n^2 intersection points, and at each intersection there is a copy of H. So the number of edges is $m = n^2 h$, where h is the number of edges in H. If n is large, say $n \gg h$, we get $m = n^{2+\varepsilon}$, so $n = m^{\frac{1}{2}-\varepsilon}$. $\square$

So perhaps it is possible that there are simple proofs like these for Vertex Cover, 3-Satisfiability, Clique, Metric TSP, etc. But it was open until the early 90s.

However, Papadimitriou and Yannakakis showed that all of these problems are equivalent as far as having a PTAS is concerned; they defined a class of these problems called $\mathsf{MAX-SNP}$. That is, either all of them do, or none of them do.

Then the PCP theorem came along, proved by a long sequence of papers, starting with Interactive Proofs and the Sum-Check Protocol. This showed that none of these have a PTAS; in particular, for each of them there is a constant such that beyond this it is NP-hard to approximate.

Theorem 13.4 (PCP Theorem (Version 1)). *There is a polynomial-time reduction from* $\mathsf{3-SAT}$[4] *to* $\mathsf{3-SAT}$ *such that the following holds:* $\phi \rightsquigarrow \psi$ *where if* $\phi \in \mathsf{3-SAT}$*, then* $\mathrm{OPT} = 1$ *and if* $\phi \in \mathsf{3-SAT}$*, then* $\mathrm{OPT} \leqslant c$ *where* $c < 1$ *is a constant.*

It is clear from this that $\mathsf{3-SAT}$ does not have a PTAS.

The proof of this theorem is very very long.

To get a flavor of how hardness of approximation results are proved using this, we use the PCP Theorem to go from $\mathsf{3-SAT}$ to a gap instance of $\mathsf{3-SAT}$, then we amplify the gap using the Parallel Repetition Cover to a Label Cover Problem, then on top of that, build Fourier Analysis and other things. This gives the final result.

One of the nice things is that until the Label Cover step, we do not need to know the proof of the PCP or Parallel Repetition theorems; we can just take them as a black box.

Let us illustrate a concrete example of using the PCP Theorem.

Theorem 13.5. *The Independent Set problem is* NP*-hard to approximate within some factor* $c > 1$*.*

Proof. Take an instance ψ of $\mathsf{Gap3-SAT}$, then we map $\psi \rightsquigarrow G$ such that if ψ has a satisfying assignment, that is $\mathrm{OPT}(\psi) = 1$, then G has an independent set of size m, and otherwise if $\mathrm{OPT}(\psi) < c$, then an independent set of C has size $\leqslant cm$, where $1/c$ is the number of clauses in ψ.

For every clause, create a triangle with the literals as the vertices, then create an edge between every pair of vertices x_i and $\overline{x_i}$.

In the YES case we know that the ψ has a satisfying assignment, so within every clause, at least one literal is true, then in every triangle we take at least one vertex. Since we never take a variable and its negated copy, this is an independent set. □

So once we know that we cannot approximate within a factor of say 0.001, then we can amplify the gap using *graph products*. Given a graph $G = (V, E)$, define the *graph product* of G as $G^{\otimes 2} = (V \times V, \{((u, v), (u', v')) \mid (u, u') \in E \text{ or } (v, v') \in E\})$.

Exercise 13.6. $\mathrm{OPT}(G^{\otimes 2}) = \mathrm{OPT}(G)^2$, where OPT denotes the maximum size of the independent set. ◇

Now once we have the graph G, we can create k-wise tensor products of G in polynomial time. Then if $\mathrm{OPT}(G) \geqslant m$, $\mathrm{OPT}(G^{\otimes k} \geqslant m^k$, and if $\mathrm{OPT}(G) \leqslant cm$, then $\mathrm{OPT}(G^{\otimes k}) \leqslant c^k m^k$. Thus we get the following:

[4]or any NP-complete language

Theorem 13.7. *It is* NP*-hard to approximate Independent Set within factor* $n^{0.01}$*, where* n *is the number of vertices.*

What we do is take $k = \log n$, but then the size is too big, but we can take an induced subgraph of polynomial size.

So Independent Set is very hard to approximate.

In general, it is not always easy to see when an NP-hard problem is easy or hard to approximate. For example, even problems that are similar in terms of exact computation, such as Vertex Cover and Independent Set, are very different from the point of view of approximation.

Now $3-\mathsf{SAT}$ is an instance of what are called *Constraint Satisfaction Problems*, where it is known that there is a class of Semi-Definite Programming relaxations that solve all CSP problems.

13.3. Probabilistically Checkable Proofs

We have stated the PCP Theorem as a reduction. However, originally the theorem was proved in a different language, a language known as Probabilistically Checkable Proofs. The fact that people were thinking in that language was helpful for showing this result.

We know very well that NP is the class of languages with short proofs.

So what happens is that we have a polynomial-time verifier that given input x, and a supposed proof Π of length $|x|^{O(1)}$, the verifier has full access to all of x and Π, then outputs a decision, either Accept or Reject.

Depending on whether the language is in the language or out of the language, we have concepts of Completeness and Soundness, where Completeness says that $x \in L \implies \exists \Pi, V(x, \Pi) = 1$, and Soundness says that $x \notin L \implies \forall \Pi, V(x, \Pi) = 0$.

Now we propose another model for how the proof needs to be checked. The verifier is probabilistic, and cannot read the entire proof; the number of bits it can read is given as a parameter.

Definition 13.8. A $(r(n), q(n))$*-restricted verifier* uses $r(n)$ random bits, and reads $q(n)$ bits from the proof, where $n = |x|$. The verifier reads the input x and random string τ, then computes indices $i_1, \ldots, i_{q(n)}$, $1 \leqslant i_1, \ldots, i_{q(n)} \leqslant |\Pi|$ based on τ. This gives a bit string of length $q(n)$. The verifier accepts or rejects based on a predicate $C_\tau : \{0,1\}^{q(n)} \to \{0,1\}$. The verifier accepts if and only if $C(\Pi[i_1], \ldots, \Pi[i_{q(n)}]) = 1$.

For completeness and soundness, $x \in L \implies \exists \Pi, \Pr_\tau[V(x, \Pi) = 1] = 1$, and $x \notin L \implies \forall \Pi, \Pr_\tau[V(x, \Pi) = 1] \leqslant \frac{1}{2}$. $\diamond$

Theorem 13.9 (PCP Theorem (Version 2)[5]). *Every language in* NP *has a* $(O(\log n), O(1))$*-restricted verifier.*

That $r(n) = O(\log n)$ really just ensures that the proof has polynomial size, and concretely for $O(1)$, in fact 5 bits is good enough.

This seems rather mysterious, and it is difficult to see a priori that thus a thing could be true. So we will demonstrate a non-trivial language where we can even just query 1 bit. Lecture 25, 30 Apr

Theorem 13.10. GNI *has a* $(O(n \log n), 1)$*-restricted verifier.*

[5]Though in reality this is the original version.

We previously saw an Interactive Proof for this. One way to turn any Interactive Proof into a Probabilistically Checkable Proof is to just have the prover write down the answers to all possible questions based on randomness.

Proof. Given $\langle G_0, G_1\rangle$, the verifier picks $|\tau| = O(n\log n)$ random bits to represent $n!$ permutations of a graph. In the supposed proof Π, at each location indexed by n-vertex graph H is a bit, which is set to 0 if $H \equiv G_0$ and 1 if $H \equiv G_1$.

So the verifier picks $b \in \{0,1\}$ at random, then picks H_τ and accepts if and only if $\Pi[H_\tau] \equiv G_b$.

If $\langle G_0, G_1\rangle \in \mathsf{GNI}$ then $\exists \Pi, \Pr[\text{Accept}] = 1$, and if $\langle G_0, G_1\rangle \notin \mathsf{GNI}$ then $\forall \Pi, \Pr[\text{Accept}] = \frac{1}{2}$. □

Now note that the number of bits is $n\log n$ instead of $\log n$, but this is not in contradiction with the statement of the theorem.

One thing to emphasize whenever we talk about proofs is the format or specification of the proof and what the proof means. But this is always up to the prover.

Proposition 13.11. *Theorem 13.4 $\implies$ Theorem 13.9.*

Proof. Let $L \in NP$ is NP-hard, where $x \rightsquigarrow \phi$ via a magic reduction.

Now pick a random clause C_i with three clauses. Read the locations of the three literals in Π. Accept if and only if these satisfy C_i.

If $x \in L$, $\text{OPT}(\phi) = 1$ so $\exists \Pi, \Pr[\text{Accept}] = 1$, and if $x \notin L$, $\text{OPT}(\phi) \leqslant c$ so $\forall \Pi, \Pr[\text{Accept}] \leqslant c$.

By picking enough clauses we can drag c down to $\frac{1}{2}$ as required. □

Proposition 13.12. *Theorem 13.9 $\implies$ Theorem 13.4.*

Proof. For a language $L \in \mathsf{NP}$ we have a $(O(\log n), O(1))$-verifier. We have input x, $C_\tau : \{0,1\}^{q(n)} \to \{0,1\}$, and locations $i_1, \dots, i_{q(n)}$.

So think of Π as a sequence of unknown variables $y_1, y_2, \dots, y_{|\Pi|}$. Write down the $2^r = \text{poly}(n)$ constraints $\tau : C_\tau(y_{i_1}, \dots, y_{i_q})$, this is a Constraint Satisfaction Problem instance Ψ.

If $x \in L$ then $\exists \Pi, \Pr[\text{Accept}] = 1$, so $\text{OPT}(\Psi) = 1$, and if $x \notin L$ then $\forall \Pi, \Pr[\text{accept}] \leqslant \frac{1}{2}$, so $\text{OPT}(\Psi) \leqslant \frac{1}{2}$.

Fact 13.13. *Any constraint $C(y_1, \dots, y_q)$ is equivalent to a system of $\leqslant q2^q$ $3-\mathsf{SAT}$ clauses with possibly auxiliary variables.*

One way to see this is that we can write the constraint as a circuit and then simulate the circuit using $3-\mathsf{SAT}$ clauses. So we can take our system of constraints Φ and send it to a formula Φ, so that $\text{OPT}(\Psi) = 1 \implies \text{OPT}(\Phi) = 1$, and $\text{OPT}(\Psi) \leqslant \frac{1}{2} \implies \text{OPT}(\Phi) \leqslant 1 - \frac{1}{2}\frac{1}{q2^q}$. □

13.4. Example: 3-Linearity

Definition 13.14. The 3-Linearity problem $3-\mathsf{LIN}$ is as follows: given boolean variables $x_1 \dots, x_n$, we have a system ϕ of m equations over $GF(2)$ each of the form $x_i + x_j + x_k = c_\ell$, $\ell \in \{0,1\}$. Let $\text{OPT}(\phi)$ be the maximum fraction of equations that can be satisfied. ◇

Fact 13.15. $\text{OPT}(\phi) \geqslant \frac{1}{2}$.

This can be done by assigning each variable randomly, and then we get this in expectation and derandomize. But in fact an easier way to do this is to look at the right hand side and assign all variables to 0 or to 1 accordingly.

FACT 13.16. *If* $\mathrm{OPT}(\phi) = 1$ *such an assignment can be found easily.*

This is done by using Linear Algebra.

THEOREM 13.17 (Håstad). *For every constant* $\varepsilon > 0$*, it is* NP*-hard to distinguish if* $\mathrm{OPT}(\phi) \geqslant 1 - \varepsilon$ *or* $\mathrm{OPT}(\phi) \leqslant \frac{1}{2} + \varepsilon$ *for 3-Linearity.*

COROLLARY 13.18. *For every constant* $\varepsilon > 0$*, it is* NP*-hard to distinguish given* $3-\mathsf{SAT}$ *instance* ψ *whether* $\mathrm{OPT}(\psi) \geqslant 1 - \varepsilon$ *or* $\mathrm{OPT}(\psi) \leqslant \frac{1}{2} + \varepsilon$*.*

PROOF. For an equation $x+y+z = 0$, we can write down four clauses $x \vee y \vee \overline{z}$, $x \vee \overline{y} \vee \overline{z}$, $\overline{x} \vee y \vee \overline{z}$, and $\overline{x} \vee \overline{y} \vee \overline{z}$.

Now suppose we fail on half of the linear equations, then that means that we fail on at least one of the four clauses. So we must fail on at least $\frac{1}{8}$. $\square$

Let us now prove Håstad's theorem.

We claim that it suffices to construct a PCP (for a NP-complete language L) that reads 3 bits from the proof x, y, z, and then the acceptance predicate is $GF(2)$-linear.

Then we have "imperfect completeness": $x \in L \implies \exists \Pi, \Pr[\text{accept}] \geqslant 1 - \varepsilon$, and $x \notin L \implies \forall \Pi, \Pr[\text{accept}] \leqslant \frac{1}{2} + \varepsilon$.

Then it is immediate that we can move to the point of view of number of satisfied equations.

Suppose we take the reduction from 3-Satisfiability to Independent Set. There is the notion of a "gadget" that, say for a clause $x_1 \vee x_2 \vee \overline{x_3}$, is a triangle whose vertices are $x_1, x_2, \overline{x_3}$, and then the reduction just makes several copies of this gadget, and then connect them up.

In this reduction $L \rightsquigarrow 3-\mathsf{LIN}$ the gadgets will be sophisticated, and proving their correctness will involve essentially proving Fourier analytic theorems. In particular, this gadget can be viewed as a property testing problem.

It is helpful to switch from $0, 1$ values to $\mathbb{F}_2 = \{\pm 1\}$, then the equations are $x_i x_j x_k = \pm 1$.

We will need an idea known as *linearity testing*.

For each property, there is a class of combinatorial objects satisfying that property, but sometimes the input is too large, so we can test locally and make queries. However, in this way we can never distinguish between objects with a property and objects very close to it, eg. graphs that are bipartite and graphs that are bipartite with an extra edge, so we can only distinguish between bipartite graphs and graphs that are "far" from being bipartite. Lecture 26, 5 May

So we are interested in a boolean function $A : \{-1, 1\}^n \to \{-1, 1\}$. A is called *linear* if $A(xy) = A(x)A(y)$ for all inputs $xy \in \{\pm 1\}^n$.

FACT 13.19. *The class of linear functions is precisely as follows: for every subset* $\alpha \subseteq \{1, 2, \ldots, n\}$*, there is the character functions* $\chi_\alpha : \{+1, -1\}^n \to \{+1, -1\}$ *where* $\chi_\alpha(x) = \prod_{i \in \alpha} x_i$ *for* $x \in \{+1, -1\}^n$*.*

Now there are 2^n linear functions, all indexed. We need a notion of distance.

DEFINITION 13.20. Let $\Delta(A, B) = \frac{1}{2}[\{x \in \{+1, -1\}^n \mid A(x) \neq B(x)]$. Then $\Delta(A, LIN) = \min_{\alpha \subseteq [n]} \Delta(A, \chi_\alpha)$. $\diamond$

THEOREM 13.21. *There is a 3-query linearity test such that for a function $A : \{+1,-1\}^n \to \{+1,-1\}$, if $A = \chi_\alpha$ for some $\alpha \in [n]$ then $\Pr[accept] = 1$, otherwise if $\Delta(A, LIN) \geqslant \delta$ then $\Pr[accept] \leqslant 1-\delta$.*

The test is just pick $x, y \in \{-1,+1\}^n$ at random and accept if and only if $A(xy) = A(x)A(y)$.

Completeness is obvious, so we need to prove soundness. To do this, we take a detour into Fourier analysis:

Let the inner product $\langle\cdot,\cdot\rangle : \{-1,+1\}^n \to \mathbb{R}$ be

$$\langle A, B\rangle = \frac{1}{2^n} \sum_{x\in\{-1,+1\}^n} A(x)B(x) = \mathbb{E}_x[A(x)B(x)].$$

FACT 13.22. *$\{\chi_\alpha \mid \alpha \subseteq [n]\}$ forms an orthonormal basis of all $\mathbb{R}$-valued functions on $\{+1,-1\}^n$.*

So

$$\langle\chi_\alpha, \chi_\alpha\rangle = E[\chi_\alpha(x)\chi_\alpha(x)] = +1$$

and

$$\langle\chi_\alpha, \chi_\beta\rangle = \mathbb{E}_{\alpha,\beta}[\chi_\alpha(x)\chi_\beta(x)] = \mathbb{E}_{\alpha,\beta}[\chi_{\alpha\Delta\beta}(x)] = E_{\alpha,\beta}[\Pi_{i\in\alpha\Delta\beta}x_i] = \prod_{i\in\alpha\Delta\beta} \mathbb{E}_i[x_i].$$

FACT 13.23. *Any $A : \{-1,+1\}^n \to \mathbb{R}$ can be uniquely written as*

$$A(x) = \sum_{\alpha\subseteq[n]} \hat{A}_\alpha \chi_\alpha(x)$$

for $x \in \{\pm1\}^n$, where $\hat{A}_\alpha$ are real coefficients, called the Fourier coefficients.

FACT 13.24. $\mathbb{E}[A(x)^2] = \sum_{\alpha\subseteq[n]} \hat{A}_\alpha^2 = 1$ *if A is boolean.*

FACT 13.25. $\Delta(A, \chi_\alpha) = \frac{1}{2} - \frac{1}{2}\hat{A}_\alpha$.

PROOF.

$$\begin{aligned}\hat{A}_\alpha &= \langle A, \chi_\alpha\rangle \\ &= \mathbb{E}[A(x)\chi_\alpha(x)] \\ &= \Pr[A(x) = \chi_\alpha(x)] - \Pr[A(x) \neq \chi_\alpha(x)] \\ &= 1 - 2\Pr[A(x) \neq \chi_\alpha(x)] \\ &= 1 - 2\Delta(A, \chi_\alpha)\end{aligned}$$

□

So let us rewrite Theorem 13.21 in terms of Fourier coefficients:

THEOREM 13.26. *If $\Pr[accept] \geqslant 1-\delta$, then there exists some linear function χ_{α_0} such that $\Delta(A, \chi_{\alpha_0}) \leqslant \delta$, then α_0 such that $\hat{A}_{\alpha_0} \geqslant 1-2\delta$.*

PROOF. Well, consider $\frac{1+A(x)A(y)A(xy)}{2}$, which is 1 if the test accepts, and 0 if the test rejects. Then

$$\begin{aligned}
\Pr[accept] &= \mathbb{E}_{x,y}\left[\frac{1+A(x)A(y)A(xy)}{2}\right] \\
&= \frac{1}{2}+\frac{1}{2}\mathbb{E}_{x,y}[A(x)+A(y)A(xy)] \\
&= \frac{1}{2}+\frac{1}{2}\mathbb{E}_{\alpha,\beta,\gamma}\left[\left(\sum_\alpha \hat{A}_\alpha\chi_\alpha(x)\right)\left(\sum_\beta \hat{A}_\beta\chi_\beta(y)\right)\left(\sum_\gamma \hat{A}_\gamma\chi_\gamma(xy)\right)\right] \\
&= \frac{1}{2}+\frac{1}{2}\sum_{\alpha,\beta,\gamma}\hat{A}_\alpha\hat{A}_\beta\hat{A}_\gamma\mathbb{E}_{\alpha,\beta,\gamma}\left[\chi_\alpha(x)\chi_\beta(y)\chi_\gamma(x)\chi_\gamma(y)\right] \\
&= \frac{1}{2}+\frac{1}{2}\sum_{\alpha,\beta,\gamma}\hat{A}_\alpha\hat{A}_\beta\hat{A}_\gamma\mathbb{E}_{\alpha,\gamma}\left[\chi_\alpha(x)\chi_\gamma(x)\right]\mathbb{E}_{\beta,\gamma}\left[\chi_\beta(y)\chi_\gamma(y)\right]
\end{aligned}$$

which is 0 unless $\alpha=\beta=\gamma$.

When they are all equal, we get $\Pr[\text{accept}] = \frac{1}{2}+\frac{1}{2}\sum_\alpha \hat{A}_\alpha^3$.

Then if $\Pr[\text{accept}] \geqslant 1-\delta$ then $\sum_\alpha \hat{A}_\alpha^3 \geqslant 1-2\delta$, but we know that $\sum_\alpha \hat{A}_\alpha^2 = 1$. Then one of the A_α must be very large and the rest must be very small. In other words, $\exists\alpha_0$ such that $\hat{A}_{\alpha_0} \geqslant 1-2\delta$. □

Lecture 27, 7 May

We will now "cheat" a bit in that we will not prove Håstad's theorem fully, but instead we will prove it assuming what is known as the Unique Games Conjecture. The reason is twofold: to introduce the Unique Games Conjecture, and assuming the Unique Games Conjecture, the proof is much cleaner, although it is possible to prove the theorem without it.

In a *Unique Games* instance, given a graph $G(V,E)$, we want to assign a labeling $\alpha(u) \subseteq \{1,\dots,n\}$ for n constant[6] to each vertex $u \in V$, so as to maximize the number of edges that are "properly labeled" or satisfied: for every edge (u,v), the edge is satisfied if $\alpha(u)\Delta\alpha(v) = \gamma(u,v)$, where $\gamma(u,v)$ is a fixed set depending on the edge.

CONJECTURE 13.27 (Unique Games Conjecture). *For every constant $\delta > 0$, there exists a sufficiently large constant n such that it is* NP*-hard to distinguish for a Unique Game instance G with labels from $\{1,\dots,n\}$ if $\mathrm{OPT}(G) \geqslant 1-\delta$ or $\mathrm{OPT}(G) \leqslant \delta$.*

What we will show is that this conjecture implies Håstad's Theorem. This is done via a reduction from Unique Games to 3−LIN taking $G \rightsquigarrow L$ such that for every ε, there is some δ such that $\mathrm{OPT}(G) \geqslant 1-\delta \implies \mathrm{OPT}(L) \geqslant 1-\varepsilon$ and $\mathrm{OPT}(G) \leqslant \delta \implies \mathrm{OPT}(L) \leqslant \frac{1}{2}+\varepsilon$.

Equivalently, we can create a 3-query PCP with a linear predicate with completeness $\geqslant 1-\varepsilon$ and soundness $\leqslant \frac{1}{2}+\varepsilon$.

So given a Unique Games instance $G(V,E)$, we construct an instance of 3−LIN as follows: we replace each vertex u with a block of 2^n variables, where n is the size of the set from which labels are drawn. This block can be interpreted as a function $A : \{-1,+1\}^n \to \{-1,+1\}$. For another vertex v we have a block that represents

[6] n here does not denote the size of the input.

a function $B : \{-1, +1\}^n \to \{-1, +1\}$. What we will do is, if we intend to assign $\alpha \subseteq [n]$ to u, then we assign χ_α to the block for u.

Now for every edge (u, v) and choice of inputs x, y, the equations will look like this: $A(xy)\chi_\gamma(xy) = B(x)B(y)$. This is what we would do if we were to do this as a reduction.

If were were to think of this as a PCP Verifier, the verifier would proceed as follows: it picks edge (u, v) at random, then the prover has written down for each vertex some truth table. So let A, B be supposed truth tables for u and v, respectively, and we have $\gamma(u, v)$. Accept if and only if $A(xy)\chi_\gamma(xy) = B(x)B(y)$. Clearly this is a 3-query PCP with linear predicate.

Now we have to complete the completeness and soundess properties. It should be clear that the reduction is designed exactly so that this happens.

For completeness, suppose $\mathrm{OPT}(G) \geqslant 1 - \delta$, then for $(1 - \delta)$ fraction of edges (u, v) there are assignments $u \to \alpha$ and $v \to \beta$ such that $\alpha\Delta\beta = \gamma$. Then we take this assignment, write down the truth tables of χ_α for u and χ_β for v. We observe that all equations are satisfied for that edge, that is,

$$\chi_\alpha(xy)\chi_\gamma(xy) = \chi_\beta(x)\chi_\beta(y).$$

Well we know that

$$\chi_\alpha(xy)\chi_\gamma(xy) = \chi_{\alpha\Delta\gamma}(xy) = \chi_\beta(xy) = \chi_\beta(x)\chi_\beta(y)$$

since $\alpha\Delta\gamma = \beta$. So if $\mathrm{OPT}(G) \geqslant 1 - \delta$, then in fact $\mathrm{OPT}(L) \geqslant 1 - \delta$ and $\delta \leqslant \varepsilon$.

For soundness, we will do the contrapositive. Suppose we have an assignment for L that satisfies at least $\frac{1}{2} + \varepsilon$ fraction of equations, then we will construct an assignment that satisfies at least δ fraction of edges. We will do this probabilistically, and then observe that there is a corresponding deterministic assignment.

We start by assuming that there is an instance of $3{-}\mathsf{LIN}$ such that

$$\frac{1}{2} + \varepsilon \leqslant \Pr[\text{accept}].$$

Then we write down

$$\Pr[\text{accept}] = \mathbb{E}_{(u,v),x,y}\left[\frac{1 + A(xy)\chi_\gamma(xy)B(x)B(y)}{2}\right],$$

then we get

$$2\varepsilon \leqslant \mathbb{E}_{(u,v),x,y}[A(xy)\chi_\gamma(xy)B(x)B(y)].$$

We know what the right hand side expression is in terms of its Fourier coefficients:

$$\mathbb{E}_{(u,v)}\left[\sum_\alpha \hat{A}_{\alpha\Delta\gamma}\hat{B}_\alpha^2\right].$$

Now we can just apply Cauchy-Schwartz and $\sum_\alpha \hat{B}_\alpha^2 = 1$:

$$2\varepsilon \leqslant \mathbb{E}_{(u,v)}\left[\sqrt{\sum_\alpha \hat{A}^2_{\alpha\Delta\gamma}\hat{B}^2_\alpha}\sqrt{\sum_\alpha \hat{B}^2_\alpha}\right] = \sqrt{\mathbb{E}_{(u,v)}\left[\sum_\alpha \hat{A}^2_{\alpha\Delta\gamma}\hat{B}^2_\alpha\right]}.$$

Then we can square both sides, and get

$$4\varepsilon^2 \leqslant E_{(u,v)}\left[\sum_\alpha \hat{A}^2_{\alpha\Delta\gamma}\hat{\beta}^2_\alpha\right].$$

Now for every v, let B be the boolean function written down for it. Then pick α with probability $\hat{\beta}_{\alpha}^{2}$, and assign $v \leftarrow \alpha$. Then the right hand side is just the expected fraction of edges satisfied. So we can just choose $\delta < 4\varepsilon^2$ and then we are done.

To do this without the Unique Games Conjecture, we can use a slightly more general version of the Unique Game which is known to be NP-hard to distinguish between satisfiable and highly unsatisfiable. There is a reduction from that, but there are more technical issues. For example, the linearity testing does not work as such: we need to modify it to what is known as dictatorship testing.

Index