# Lectures in Computational Complexity

Jin-Yi Cai
Department of Computer Sciences
University of Wisconsin
Madison, WI 53706
*Email: jyc@cs.wisc.edu*

December 12, 2003

# Contents

# Chapter 1

# Genesis

**Chapter Outline:** Structural versus Computational Mathematics. Historical perspectives. Brief overview of complexity theory. Hilbert's Tenth Problem. Turing Machines. Undecidability. Cantor's method of diagonalization. Undecidability of the halting problem. Time and space bounded Turing machines. Hierarchy Theorems. Complexity Classes (L, NL, P, NP, PSPACE, E, EXP).

## 1.1 Structural versus Computational

There have always been two major strands of mathematical thought since the time of antiquity: Structural Theory and Computational Methods. For example, Euclid's *Elements* is a synthesis of much that was known in geometry up till that time, it is also largely structural in that it emphasizes (and establishes) theorems and deductive proofs; by contrast, the writings of Diophantus, as in *Arithmetica*, were primarily algorithmic, where some of the methods probably go back to Babylonian mathematics 2000 years before that. Of course these strands of mathematical thought are not in opposition to, but rather, complement each other. Even in Euclid's *Elements*, one finds algorithmic gems such as The Euclidean Algorithm which finds the greatest common divisor of two positive integers. It is a shining example of an early triumph in algorithm design, whose correctness and efficiency demands proof in a purely structural sense. Outside of the Greek tradition, other ancient civilizations also had various emphasis on either the *Structural Theory* which prizes the framing and proof of general theorems by deductive reasoning, or *Computation* which seeks efficient computational method to solve problems. For example, Chinese mathematicians of antiquity seem to concern themselves primarily with computation.

To a great majority of the classical masters throughout history, starting with Archimedes, Newton, Leibniz, Euler, Lagrange, Gauss, . . ., computation is an inseparable aspect of mathematics as a whole. It is a rather recent phenomenon that the computation is somehow delegated as secondary to the Big Math edifice, perhaps helped along by the influential schools such as Bourbaki. But in reality, much of the motivation for the big structural discoveries were computational originally. For example, Calculus was invented so as to facilitate computation of orbits of heavenly bodies as well as measuring surface areas and volumes; Galois Theory and finite group theory was a discovery to investigate the solvability of equations by radicals; the Prime Number Theorem was first conjectured by Gauss after much computational experiments. It is also true that much of the advances made in structural mathematics had also greatly influenced the advances in computational mathematics.

While it can be said that the subject of algorithms is as old as mathematics itself, the serious mathematical study of algorithms as such, rather than the use of them, is a relatively new development.

Perhaps one could trace this beginning to *Set Theory*, that most structural of all subjects. In his study of Fourier series (surely one of the most computational subjects in origin), Cantor gave birth to a set of ideas that we now call (naive) set theory. Cantor's ideas are revolutionary in many aspects. In its basic framework it is highly non-constructive. For example, Cantor gave a conceptually crisp and simple proof of the existence of transcendental numbers, whereby inventing his famous diagonalization method. This proof is remarkable in many ways:

Firstly, it is much simpler than the monumental achievement of Hermite and Lindemann on the transcendence of $e$ and $\pi$ respectively. Perhaps one can still make the case that the "real" transcendental number theory is more along the lines of Hermite, Lindemann and Liouville, and not the mere existence proof by the magic of diagonalization. But even the most dedicated practitioners of "hard analysis" today will not dismiss the elegance and efficiency of Cantor's method. On the other hand, today many interesting computational problems, such as basis reductions for lattices, simultaneous Diophantine approximations, and volume estimations of convex bodies, form very active research areas which can be traced directly to the work such as Dirichlet, Liouville, Hermite and Minkowski.

Secondly, as Kronecker was quick to point out, Cantor's method is inherently non-constructive, and in his view, borders on the "philosophical". In particular it did not conform to the strictly finitistic and constructive approach that Kronecker had been advocating. To the end of his day, Kronecker never accepted Cantor's idea. The finitists distrust it on philosophical ground, which is ironic because the finitists are particularly concerned with the soundness of mathematical foundation, which is to be demonstrated in coming years to be closely related to computational undecidability, in which Cantor's diagonalization method is a forerunner.

Thirdly, the diagonalization method was to find its great application in Turing's undecidability proof of the Halting Problem. It subsequently became one of the basic mathematical tools in recurcsion theory, and in the founding of complexity theory with the proof of the time and space hierarchy theorems.

Because of its fundamental importance we will review the diagonalization proof by Cantor.

An algebraic number is a root of a polynomial with integral coefficients. A non-algebraic number is called a transcendental number. A set is countable if it can be put into one-to-one correspondence with the integers. It is clear that the set of all algebraic numbers is countable, since we can count all integral polynomials, and each polynomial of degree $n$ has at most $n$ roots.

**Exercise:** Show that the following is a *pairing function* which gives a one-to-one correspondence between non-negative integers $\mathbf{Z}_+ = \{0, 1, 2, \ldots\}$ with the cartesian product $\mathbf{Z}_+ \times \mathbf{Z}_+$.

$$\langle i, j \rangle = \binom{i+j-1}{2} + j.$$

**Exercise:** Show that the set of rational numbers, and the set of algebraic numbers are countable.

Theorem 1.1 *The set of real numbers is uncountable; in particular, there are non-algebraic real numbers.*

A curious historical note: In order not to offend Kronecker, who was powerful and somewhat petty at the same time and might block the publication of this work, Cantor had to phrase his main result strictly on the existence of non-algebraic numbers, and not mention anything of his emerging cardinality theory of the infinite.

*Proof.* Consider all binary infinite sequences $B = \{\beta\}$, where

$$\beta = b_1 b_2 \ldots b_n \ldots,$$

and $b_i \in \{0, 1\}$. We know that the real numbers in $[0, 1]$ can be put in 1-1 correspondence with this set $B$, via binary expansion.[1]

**Claim:** $B$ is uncountable.

Suppose otherwise. Let $\beta_1, \beta_2, \ldots, \beta_n, \ldots,$ be an enumeration of all $B$, where each $\beta_i = b_{i,1} b_{i,2} \ldots b_{i,n} \ldots$. Write these sequences by the rows, and we obtain an infinite table as follows.

[1] There is a technical problem of some real numbers in $[0, 1]$ having two different infinite binary expansions. But it is easy to see that these real numbers are precisely those with a finite terminating binary expansion, and thus are rational numbers, and clearly countable. Therefore the claim of 1-1 correspondence with $B$ is still valid.

$$\begin{array}{c|ccccccc}
 & 1 & 2 & 3 & 4 & \cdots & i & \cdots \\
\hline
\beta_1 & \boxed{b_{1,1}} & b_{1,2} & b_{1,3} & b_{1,4} & \cdots & \cdots & \cdots \\
\beta_2 & b_{2,1} & \boxed{b_{2,2}} & b_{2,3} & b_{2,4} & \cdots & \cdots & \cdots \\
\beta_3 & b_{3,1} & b_{3,2} & \boxed{b_{3,3}} & b_{3,4} & \cdots & \cdots & \cdots \\
\beta_4 & b_{4,1} & b_{4,2} & b_{4,3} & \boxed{b_{4,4}} & \cdots & \cdots & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & & \\
\beta_i & b_{i,1} & b_{i,2} & b_{i,3} & b_{i,4} & \cdots & \boxed{b_{i,i}} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & & & \ddots
\end{array}$$

Now we define a particular $\beta = b_1 b_2 \ldots b_n \ldots \in B$, which by its definition can not be on this list: We go down the diagonal of this infinite table; for the $n$ th entry, if it is 0 we let our new $b_n = 1$, and if it is 1 we let $b_n = 0$. Formally, $\forall n \geq 1$,

$$b_n = \begin{cases} 0 & \text{if } b_{n,n} = 1 \\ 1 & \text{otherwise.} \end{cases}$$

Thus we "constructed" our $\beta \in B$ to be "disagreeable": It disagrees with the $n$ th item on the list in the $n$ th place. Hence this $\beta$ cannot be among those listed. (If it were the $n$ th sequence, what should its $n$ th entry be?) ♣

It is self evident why Cantor's method is called the diagonalization method.

Kronecker's objections not withstanding, Cantor's set theory opened up a mathematical "paradise", from which, Hilbert was said to have remarked that mankind will never be driven out again. Nevertheless, it did bring troubles with it. In particular, at the turn of the 20th century, a number of set theoretic paradoxes were found that pertain to the foundation of mathematics.

Here is the famous Russell's paradox.

"Surely" pure logic dictates that a set $A$ either satisfies $A \in A$ or $A \notin A$. Let's form $\mathcal{A} = \{A \mid A \notin A\}$. In Cantor's naive set theory this seems perfectly legitimate. *If*, Russell says, we can form this set $\mathcal{A}$, then is $\mathcal{A} \in \mathcal{A}$? If $\mathcal{A} \in \mathcal{A}$, then by its definition, $\mathcal{A} \notin \mathcal{A}$. However, if $\mathcal{A} \notin \mathcal{A}$, then again by definition, $\mathcal{A} \in \mathcal{A}$.

## 1.2 Turing Machines and Undecidability

At the turn of the last century, paradoxes such as this one stirred up a lot of uneasiness with the foundation of mathematics, which was to be formulated increasingly based on Cantor's naive set theory. This created the strong desire to re-examine the foundation of mathematics, and try to formalize all mathematics on an axiomatic basis without contradiction. (In the modern axiomatic set theory, such paradoxes are avoided by being more careful as to what is admissible as a set construction.)

Hilbert was a leading advocate of the formalist school. In his famous address to the International Mathematicians Congress in 1900, he listed 23 problems which he thought to be most likely to excite the imagination of mathematicians worldwide in the new century. A number of Hilbert's problems are concerned with foundational issues, such as the Continuum Hypothesis. The 10th problem on the list asks the following: Find a systematic procedure to determine if a polynomial with integral coefficients in several variables has an integral solution. More broadly then, Hilbert initiated the study of *Decision Problems* where the aim is to find an algorithm to decide for each instance of a problem. The research in the next 40 years showed that the study of computability is intimately related to the foundation of mathematics. Several other Hilbert problems also had a profound impact on the future development of the foundation of mathematics and computability theory, such as the Continuum Hypothesis (#1) and Consistency Problem (#2), but it was the Decision Problem, a.k.a. *Entscheidungsproblem*, that was the primary impetus to Turing's seminal work which established the foundation of computability theory.

It turns out that the notion of a systematic procedure to compute is intimately related to the notion of a formal proof in axiomatic system. The work of Turing, Gödel, Church, Post, and others, established that the original program envisioned by Hilbert cannot be complete. While Hilbert only asked for an algorithm in the 10th problem, the possibility of the non-existence of such an algorithm probably never crossed his mind. This turned out to be the case in the end, as it was finally proved by Matiyasevich in 1970. But, it was Hilbert who raised the question, and focus the attention on the very notion of what constitutes an algorithm, what is computation. In answering Hilbert, computability theory was born.

It is to address Hilbert's Entscheidungsproblem, Turing defined his model of a general purpose computer—the Turing Machine. A Turing Machine (TM) has a finite state control and an infinite input/output tape with a reading/writing head. A deterministic Turing Machine moves in each step in a unique way determined by its current state, and the symbol it is currently scanning. A move consists of a possible change of state and scanned symbol, and moving the head left or right. A non-deterministic Turing Machine (NTM) may have several legal moves, which are still determined by its current state and the scanned symbol. (NTMs are not important for computability theory, but important for complexity theory later.) An input $x$ is accepted by a deterministic TM if the computation starting with initial configuration ends in an accepting state. For a NTM acceptance is defined by the existence of some sequence of legal moves that ends in an accepting state. Again for complexity theory purposes, we also define multitape Turing Machines. For space bounded Turing Machines with sublinear space bound we allow a separate read-only input tape, and a work tape (or several tapes [2]), and the space bound is counted only on the work tape(s). We assume the

[2] For space complexity it turns out a single work tape suffices.

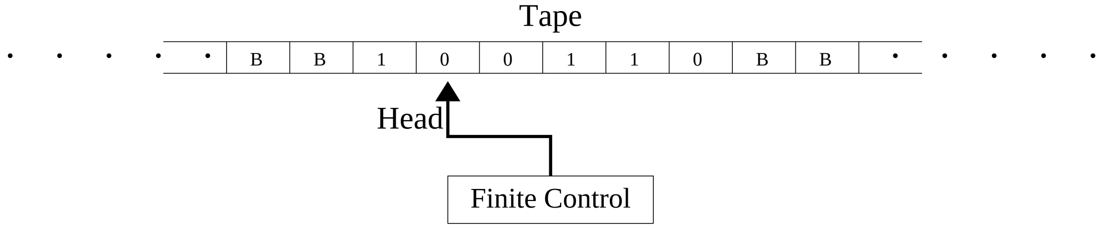


Figure 1.1: Turing Machine

readers are already familiar with these notions; for detailed definitions, please see [3, 4, 2].

TMs are not the only model of computation formulated. In the 1930s a number of different models were formulated; however every one of the general models were shown to be equivalent to TMs in due course. The universality of TMs is captured by The Church-Turing thesis which states that whatever can be computed can be computed by a Turing Machine.

Cantor's diagonalization method was technically the opening salvo by Alan Turing. Adapting Cantor's method, Turing showed that there are problems which can not be computed by any Turing Machine, and thus, by the Church-Turing thesis, uncomputable by any algorithm whatsoever. Such problems are called undecidable problems. In particular, the so-called Halting Problem for Turing Machines is one such problem. Furthermore, a reduciton theory is developed whereby a host of problems can be shown undecidable.

The proof of the undecidability of the Halting Problem goes as follows.

List all the Turing Machines $M_1, M_2, \ldots$ row by row, and index the columns by the inputs of all finite strings, which we will identify with integers $j = 1, 2, \ldots$. Mark the $(i, j)$ entry of this table by $M_i(j)$, the outcome of machine $M_i$ on input $j$. We will only consider the outcome of either "halt" or "not halt". We are not claiming this outcome can be computationally determined in general, and in fact the purpose of this proof is precisely to show that it is impossible to determine this outcome by TMs, and thus by the Church-Turing thesis, undecidable by any algorithm.

Thus we obtain an infinite table, much like that in Cantor's proof.

Now suppose there is a decision procedure in the form of a Turing Machine $M$ that, for any $\langle i, j\rangle$, can decide whether $M_i(j)$ halts or not in a finite number of steps. Let's say $M(\langle i, j\rangle) = 1$ if $M_i(j)$ halts and $M(\langle i, j\rangle) = 0$ otherwise. Then one can design another Turing Machine $M'$ as follows: On any $i$, $M'(i)$ simulates $M(\langle i, i\rangle)$. If $M(\langle i, i\rangle) = 1$, then $M'(i)$ enters a loop and never halts. If $M(\langle i, i\rangle) = 0$, then $M'(i)$ halts.

Now since all Turing Machines have been enumerated, there exists a $k$, such that our $M'$ is $M_k$. But what happens to $M_k(k)$?

$$\begin{array}{c|cccccc}
 & 1 & 2 & 3 & \cdots & i & \cdots \\
\hline
M_1 & \boxed{M_1(1)} & M_1(2) & M_1(3) & \cdots & \cdots & \cdots \\
M_2 & M_2(1) & \boxed{M_2(2)} & M_2(3) & \cdots & \cdots & \cdots \\
M_3 & M_3(1) & M_3(2) & \boxed{M_3(3)} & \cdots & \cdots & \cdots \\
\vdots & \vdots & \vdots & & \ddots & & \\
M_i & M_i(1) & M_i(2) & M_i(3) & \cdots & \boxed{M_i(i)} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & & \ddots
\end{array}$$

If $M_k(k)$ eventually halts, then by the assumption of $M$, $M(\langle k, k\rangle) = 1$, and thus $M'(k)$, which is $M_k(k)$, never halts. If $M_k(k)$ does not halt, then again by assumption, $M(\langle k, k\rangle) = 0$, and $M'(k)$ halts. Either way we have a contradiction. ♣

Next we will define time and space bounded computation. This is the domain of computational complexity theory. We will see that diagonalization method reappears, to establish that more time or more space provably can compute more.

## 1.3 Time, Space and Non-determinism

The following notions are basic, and can be found in more details in any reference books [3, 4, 2, 1].

A DTM is formally $(Q, \Sigma, q_0, \delta, F)$, where $Q$ is a finite set of states. $\Sigma$ is a finite set of alphabet symbols, excluding a special symbol $\sqcup$. $q_0 \in Q$ is the starting state. $F \subseteq Q$ is a subset of accepting states. $\delta$ is a partial transition function that maps $(Q - F) \times (\Sigma \cup \{\sqcup\})$ to $Q \times \Sigma \times \{L, R\}$. The interpretation is as follows: when TM $M$ is in state $q \in Q - F$, and reading symbol $A \in \Sigma \cup \{\sqcup\}$, if $\delta$ is defined, say $\delta(q, A) = (q', A', \Delta)$, where $\Delta = L$ or $R$, then in one step the partial transition function $\delta$ provides the next state $q' \in Q$, and overwrite the symbol by $A' \in \Sigma$, and moves left ($\Delta = L$) or right ($\Delta = R$). Thus, a TM is defined by a finite set of quintuples of the form $(q, A, q', A', \Delta)$, which can be written as an integer in a standard encoding scheme, e.g., using repeated applications of our pairing function $\langle \cdot, \cdot \rangle$. We can further assume every integer encodes a TM, with the simple convention that an "invalid" encloding corresponds to a TM that has no legal moves. Furthermore, we will almost exclusively use the binary alphabet set $\{0, 1\}$.

We will use multitape TMs, which are similarly defined. The time complexity of deterministic TM $M$ is

$$\mathrm{time}_M(n) = \max\{\#\text{ of steps in } M(x), \text{ for } |x| = n\}.$$

For any function $f(n)$,

$$\text{DTIME}[f] = \{L \mid \text{ for some } M\ , L(M) = L, \text{ and } \text{time}_M(n) \leq f(n), \text{ for all large } n.\}.$$

In other words, DTIME$[f]$ consists of problems that are computable by some TM with running time at most $f(n)$ asymptotically. By some silly tricks such as enlarging the alphabet set, and the set of finite states, one can show that any constant factor does not matter. Thus DTIME$[f]$ = DTIME$[O(f)]$. For technical reasons we will only consider "nice" functions $f(n)$, called fully time-constructible functions. This means that there is some TM $M$, which for every $n$, and any input $x$ of size $n$, $M(x)$ runs in exactly $f(n)$ steps and then halts.[3] Almost any reasonable functions $\geq n$, such as $n$, $n^k$, $n^i(\log n)^j$, $2^{(\log)^k}$, $2^{n^k}$, and $2^{2^{\cdot^{\cdot^{2^n}}}}$. They are also closed under addition, multiplication, exponentiation, etc. We will always assume $f(n) \geq n$, the time needed to read the input.

The model of TM is chosen because it is relatively robust. One can show, for instance, that any $k$-tape TM running in time $f(n)$ can be simulated by a 2-tape TM in time $O(f(n)\log(f(n)))$. For our purposes we will only need the more trivial simulation in time $O(f(n)^2)$, even by 1-tape TM. This simulation can be seen easily as follows: Devide the single tape into $2k$ tracks, and use a large alphabet set with more than $2^{2k}$ symbols, say. Keep on the single tape the contents of all $k$ tapes, together with a mark for each head position. Then one step of the computation of the $k$-tape TM is simulated by the 1-tape TM with 2 sweeps. Note that each sweep of the tape area which has been used takes at most $O(f(n))$ steps. For time complexity, the default model is multitape TMs as in the following definitions.

Let poly denote the class of polynomials, or simply $n^i + i$, $i = 1, 2, \ldots$. Then the union

$$\text{P} = \bigcup_{f \in \text{poly}} \text{DTIME}[f]$$

is the class of deterministic polynomial time. Clearly this definition is invariant when restricted to 1-tape TMs. One can similarly define exponential time classes

$$\text{E} = \bigcup_{k>0} \text{DTIME}[2^{kn}]$$

$$\text{EXP} = \bigcup_{k>0} \text{DTIME}[2^{n^k}]$$

One can define space complexity similarly. In one aspect it is even simpler, since we can use $k$ tracks to mimic $k$ tapes and there is no additional space overhead in the simulation. So we will have just one work tape. However in another respect, there is a slight complicaiton. This happens when we wish to study sublinear space complexity, which contain important problems. In order to account for sublinear space, we use a separate read-only input tape, in addition to a read-write work tape.

[3] It is a fact in complexity theory that there exist functions which are not fully time-constructible, but we will not be concerned with that.

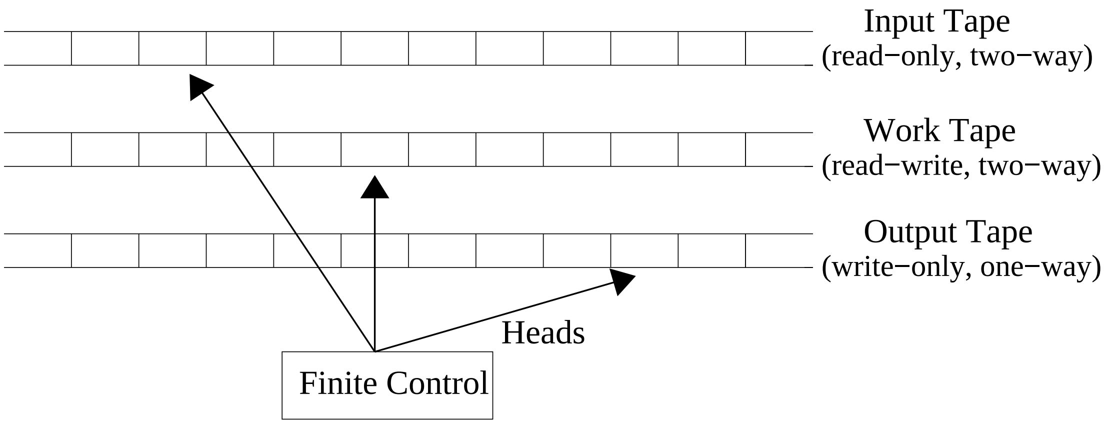


Figure 1.2: Multi-Tape Offline Turing Machine

On the read-only input tape, the input of length $n$ is written, but these $n$ cells do not count toward space complexity. We only count the number of tape cells used on the read-write work tape. Thus for space complexity, the standard model is what is known as an off-line TM, which has one read-only input tape, and one read-write work tape. Then one can define, in an obvious way,

$$\text{space}_M(n) = \max\{\# \text{ of cells on work tape used in } M(x), \text{ for } |x| = n\}.$$

Again we restrict to "nice" functions, called fully space-constructible functions.

**Exercise:** Define fully space-constructible functions.

For any such function $f(n)$, define

$$\text{DSPACE}[f] = \{L \mid \text{ for some } M\ , L(M) = L, \text{and space}_M(n) \leq f(n), \text{ for all large } n.\}.$$

We define

$$\text{PSPACE} = \bigcup_{f \in \text{poly}} \text{DSPACE}[f],$$

the class of polynomial space. (There is a reason why we omit the word "deterministic", as we shall explain later.) We also have deterministic logarithmic space L = DSPACE[log]. Note again that constant factors do not matter, thus DSPACE[log] = $\bigcup_{c>0}$ DSPACE[$c$ log].

The notion called *non-determinism* is of central importance in complexity theory. This notion is related to the process of guess and verification. We will consider two examples. Boolean Satisfiability and Graph Accessibility.

Suppose we are given a Boolean formula $\varphi$ in propositional logic, i.e., it is a well-formed-formula made up from logical AND ($\wedge$), OR ($\vee$) and NOT ($\neg$), and Boolean variables $x_1, x_2, \ldots, x_n$. The problem is whether there is a truth assignment for the variables $x_1, x_2, \ldots, x_n$ such that $\varphi$ evaluates to True. This problem is called the (Boolean) Satisfiability Problem. The set of all satisfiable formulae is denoted as SAT.

No polynomial time algorithm is know for SAT. But whenever a formula $\varphi(x_1, x_2, \ldots, x_n)$ is satisfiable, i.e., $\varphi \in$ SAT, there is an assignment, if one is given that assignment, one can easily verify that $\varphi$ is satisfiable, in particular it evaluates to True under that assignment. We consider this process as consisting of two stages, first *guess* a witness, in this case a satisfying assignment, second *verification*, to check indeed it is a satisfying assignment. Note that $\varphi$ is satisfiable if and only if such a guess exists which will pass the verification stage. Of course, one can consider this guess-and-verify to go hand in hand. In this Satisfiability Problem for example, we can imagine one guesses one bit at a time for each variable, and evaluates the formula as each variable is assigned. We will formally define non-deterministic Turing Machines (NTM) as TMs where for each state and tape symbol being read $(q, A)$, the transition function $\delta$ provides a finite set of allowable next-step transitions consisting of a state $q'$, a tape symbol $A'$ and a left or a right movement, $\Delta = L$ or $R$, of the tape head. A NTM is defined by this set of quintuples of the form $(q, A, q', A', \Delta)$. For DTM, every $(q, A)$ has at most one quintuple of the form $(q, A, q', A', \Delta)$ signifying at most one legal move; with NTM, each $(q, A)$ can have a finite set of quintuples the first two entries being $q$ and $A$ (this finite set is part of the definition of $\delta$).

These two views of non-deterministic computations, namely, 2 stage guess and verify or multiple valid moves per step, are clearly equivalent for SAT. For problems such as SAT, or in general, for non-deterministic time classes (which is at least linear $cn$), the first approach, guess-and-verify, is more intuitive. However, for formal treatment of non-deterministic computations, it is easier with the second approach. This is especially so when one is dealing with possibly non-terminating computations. [4] Another place the second approach is more suitable is when we consider space bounded computation with sub-linear space bound as in the following example (where there is not enough space to guess and write down all the non-deterministic moves at the beginning.)

Consider the following Graph Accessibility Problem: Given a directed graph $G$, and two vertices $s$ and $t$, we ask whether there is a directed path from $s$ to $t$. We denote by GAP the set of all instances $(G, s, t)$ where a directed path exists in $G$ from $s$ to $t$. (This problem, when restricted to undirected graphs, is also very important. By replacing each undirected edge by a pair of directed edges in the opposite direction, it is clear that the directed GAP is a more general problem than the undirected version.)

For GAP, there is a simple polynomial time algorithm based on Depth-First-Search (DFS). However DFS uses at least linear space. By contrast, if the space bound is limited, we have the following non-deterministic algorithm: Start from $v_0 = s$, successively guess the next vertex $v_i$ and verify that $(v_{i-1}, v_i)$ is an edge. We can keep a counter to count up to $n$, and accept iff within $n - 1$ steps, some $v_k = t$. Note that we only need to keep at any time a pair of current vertices on the work tape; when we guess for $v_{i+1}$ we need only to remember $v_i$ (in order to verify that $(v_i, v_{i+1})$ is an edge), but we no longer need to keep

[4] We will not be concerned with non-terminating computations. For example, one can design a subroutine which acts as a time clock that runs exactly $n^i + i$ steps and shuts itself off, or a space bound which marks $f(n)$ (e.g. $f(n) = \log n$ or $n^i + i$) many tape squares.

$v_{i-1}$ and all previously guessed vertices. In addition we only need to keep a counter to count up to $n$. For a graph with $n$ vertices, to name a vertex takes only $\log n$ space. Thus GAP is in non-deterministic logspace.

We will only be concerned with time/space bounded NTM, and thus we can assume each computational path terminates within the specified time/space bound. Then for a NTM $N$ and a fully time constructible $f$ we define

$$\text{time}_N(x) = \max\{n{+}1, \min_p\{\#\text{ of steps along } p, \text{ where } N(x) \text{ accepts along computational path } p\}\},^{5}$$

$$\text{time}_N(n) = \max\{\text{time}_N(x), |x| = n\},$$

$$\text{NTIME}[f] = \{L \mid \text{ for some } N\ , L(N) = L, \text{ and } \text{time}_N(n) \le f(n), \text{ for all large } n.\}.$$

Of course $L(N) = L$ means that $x \in L$ iff there is some computation path $p$, along which $N$ accepts $x$. Similarly we can define non-deterministic space classes.

$$\text{space}_N(x) = \max\{1, \min_p\{\#\text{ of worktape cells used in } N(x), \text{ where } N(x) \text{ accepts along } p\}\},^{6}$$

$$\text{space}_N(n) = \max\{\text{time}_N(x), |x| = n\},$$

$$\text{NSPACE}[f] = \{L \mid \text{ for some } N\ , L(N) = L, \text{ and } \text{space}_N(n) \le f(n), \text{ for all large } n.\}.$$

$$\text{NP} = \bigcup_{f \in \text{poly}} \text{NTIME}[f],$$

$$\text{NPSPACE} = \bigcup_{f \in \text{poly}} \text{NSPACE}[f],$$

and also

$$\text{NL} = \bigcup_{c>0} \text{NSPACE}[c \log].$$

To recap, formally a non-deterministic TM (NTM) is simply a TM which has possibly more than one legal moves at any configuration. These next-moves are specified by the current state and the tape symbols it is currently reading. A NTM accepts an input $x$ iff there exists a legal computational path that ends in acceptance.

It is obvious that L $\subseteq$ NL $\subseteq$ P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXP.

We will show later that NSPACE$[f] \subseteq$ DSPACE$[f^2]$, thus NPSPACE = PSPACE and NPSPACE will not be used.

The relationship of P and NP is *the most outstanding* open problem in Computer Science.

[5] When $N$ does not accept $x$, the time complexity is $n + 1$.

[6] When $N$ does not accept $x$, the space complexity is 1.

## 1.4 Hierarchy Theorems

The complexity classes as defined attempt to classify *decidable* computational problems according to the *inherent* computational complexity of each problem. The first question one must ask is: Does such a stratification really exist, i.e., could this be merely a definitional mirage?

In order to truly establish the validity of the existence of computational complexity theory, one must prove that problems do indeed have different inherent computational complexity. This should not depend on merely the fact that, while we have fast algorithms for some problems, we do not *currently* have fast algorithms for some others. The proof must establish for a given time or space bound $f(n)$, the existence of *some* decidable computational problems which do not possess *any* algorithm within that bound $f(n)$.

The results establish these existence theorems are known as Hierarchy Theorems. One can show, for any two time complexity functions $T_1(n)$ and $T_2(n)$, if $T_2$ grows sufficiently fast compared to $T_1$, then there are indeed problems which can be solved in time $T_2$ but can not be solved in time $T_1$. (Here we will also require a technical condition of $T_2$ being time constructible.) In fact, if $\lim_{n\to\infty} \frac{T_1(n)\log T_1(n)}{T_2(n)} = 0$, then the above statement already holds. We will prove a weaker theorem

Theorem 1.2 *Given any $T_1(n), T_2(n)$, if $T_2(n)$ is time constructible and $\lim_{n\to\infty} \frac{T_1(n)^2}{T_2(n)} = 0$, then there is a language $L \in \mathrm{DTIME}[T_2(n)] - \mathrm{DTIME}[T_1(n)]$.*

The proof also establishes in particular,

Theorem 1.3 *Given any totally recursive time bound $T(n)$, there is a recursive language $L$ not in $\mathrm{DTIME}[T(n)]$.*

There are also nonderministic time hierarchy theorems.

The Hierarchy Theorem plays the same role as the existence of undecidable problems. Not only that, the proof also adapts the diagonalization method. (We have now seen it for the third time.) On the minus column, just as Cantor's slick proof establishing the existence of transcendental numbers, these Hierarchy Theorems do not give us the specificity that certain well known problems are hard. The present day research in complexity theory is much more dominated by the quest for such "real" problems.

We will only give a proof sketch:

*Proof.* The general idea is diagonalization. First we note that all TMs can be effectively enumerated as $M_1, M_2, \ldots$. For a time constructible bound $T(n)$, one can also design a "clock", a subroutine, which runs concurrently with any TM and shuts it off after $T(n)$ steps. We can effectively enumerate all "clocked" TMs, all of which runs within time $T(n)$, and every language in $\mathrm{DTIME}[T(n)]$ is accepted by one machine on this list.

We will enumerate all the Turing Machines $M_1, M_2, \ldots$, and consider each $M_i$ on longer and longer inputs. We simulate each of them on these inputs, for up to $T_2(n)$ steps for inputs of length $n$. To be more precise, we will allocate a disjoint segment $S_i = \{\langle i, y \rangle \mid i \geq 1, y \in \Sigma^*\}$, and simulate $M_i$ on every $x \in S_i$. If the simulation on $M_i(x)$ terminates in less than $T_2(|x|)$, then *we will do the opposite,* i.e., we accept $x$ if $M_i(x)$ rejects, and we reject $x$ if $M_i(x)$ accepts. If the simulation of $M_i(x)$ does not terminate within $T_2(|x|)$ steps, then we can decide arbitrarily, accept or reject.

To account for the possibility that the simulation of $M_i(x)$ does not terminate within $T_2(|x|)$ steps, we allocated an infinite subset of inputs for every $M_i$, Since one can simulate (easily) a multitape TM $M_i$ with running time $T_1(n)$ by a one tape TM in time $T_1(n)^2$, for sufficiently large $x \in S_i$, the simulation of $M_i(x)$ will terminate within $T_2(|x|)$ steps. Now suppose $L \in \text{DTIME}[T_1(n)]$. Then for some TM $M_i$ accepting $L$ with runing time $T_1(n)$, the simulation will terminate with a different outcome, for sufficiently large $n$. Thus the language $L'$ defined by this simulation does not belong to $\text{DTIME}[T_1(n)]$. However, by the construction it is in $\text{DTIME}[T_2(n)]$. ♣

Note that the simulation is carried out by a single TM, which must have a fixed number of tapes. It is known that any $k$-tape TM with running time $T(n)$ can be simulated by a 2-tape TM in time $O(T(n)\log(T(n)))$. This is the only modification needed in the above proof to get a tighter hierarchy theorem quoted earlier.

The method of this proof is sometimes called *delayed diagonalization*. This is because, the "diagonalizing action" which kills the TM $M_i$ may happen at a later stage (in our case, for an input in $S_i$, perhaps much longer than $i$). The set $S_i$ has infintely many strings, thus arbitrarily long strings. Thus the "diagonalizing action" will happen eventually, for every TM with running time $T_1(n)$.

There is also a similar version of the Hierarchy Theorem for space complexity. In fact since for space complexity we can simulate any TM in just one tape without any loss in asymptotic space complexity, the theorem reads even tighter: ($S_2(n) \geq \log n$ is needed to carry out some basic counting operations in simulations.)

Theorem 1.4 *Given any $S_1(n)$ and $S_2(n) \geq \log n$, if $S_2(n)$ is fully space constructible and $\lim_{n\to\infty} \frac{S_1(n)}{S_2(n)} = 0$ (i.e, $S_1 = o(S_2)$), then there is a language $L \in \text{DSPACE}[S_2(n)]$ $-\text{DSPACE}[S_1(n)]$.*

The approach is the same.

**Exercise:** Prove the Space Hierarchy Theorem.

# Chapter 2

# Basic Concepts

**Chapter Outline:** In this chapter, we discuss some basic concepts. We define reductions, NP-completeness, alternations, relativizations, leading to the Polynomial-time Hierarchy.

## 2.1 P and NP

In Chapter 1, we introduced deterministic and non-deterministic Turing machines and their complexity bounded versions. Building on these notions, we defined deterministic and non-deterministic time complexity classes. We continue the discussion by first recalling these classes.

Definition 2.1 *For any time-constructible function $f(n)$,*

$$\begin{aligned} \mathrm{DTIME}[f] &= \{L \mid \textit{for some deterministic TM } M,\ L = L(M) \textit{ and } M \textit{ runs in time } f(n)\} \\ \mathrm{NTIME}[f] &= \{L \mid \textit{for some non-deterministic TM } M,\ L = L(M) \textit{ and } M \textit{ runs in time } f(n)\} \end{aligned}$$

Since every deterministic TM can also be viewed as a non-deterministic TM, we have the following proposition.

Proposition 2.2 *For any function $f(n)$, $\mathrm{DTIME}[f] \subseteq \mathrm{NTIME}[f]$.*

We obtain the two central complexity classes, P and NP by considering deterministic and non-deterministic TMs that run in polynomial time:

Definition 2.3

$$\begin{aligned} \mathrm{P} &= \bigcup_{f \in \mathrm{Poly}} \mathrm{DTIME}[f] \\ \mathrm{NP} &= \bigcup_{f \in \mathrm{Poly}} \mathrm{NTIME}[f] \end{aligned}$$

*where Poly is the set of all polynomials.*

It follows from these definitions that P $\subseteq$ NP, and perhaps the greatest open problem in computer science asks whether this containment is in fact proper, namely NP $\neq$ P.

## 2.2 NP-Completeness

One of the most useful notions in complexity theory is that of completeness. In general, one can define complete languages for any complexity class. This section is devoted to a discussion of complete languages for the class NP. We start by defining another important concept called reduction.

Definition 2.4 (Karp reduction) *A language $L_1$ reduces to another language $L_2$ by a Karp reduction, denoted as $L_1 \leq_m^p L_2$, if there is a function $f : \Sigma^* \rightarrow \Sigma^*$, such that*

1. *$x \in L_1 \iff f(x) \in L_2$, and*
2. *$f$ is deterministic polynomial time computable.*

Karp reduction is also known as *polynomial time many-one reduction*, which is the polynomial time version of the recursion theoretic notion of *many-one reduction*. There are other notions of reduction which will be discussed later. We now proceed to define the notion of NP-completeness.

Definition 2.5 *A language $L$ is said to be* NP*-hard (under Karp reductions) if every language $L' \in$ NP Karp reduces to $L$. $L$ is said to be* NP*-complete if $L \in$ NP and it is* NP*-hard.*

We next exhibit a canonical NP-complete language $A_{\mathrm{NP}}$. It is defined as follows.

$$A_{\mathrm{NP}} = \{\langle M, w, 1^t\rangle |\ M \text{ is a non-deterministic TM that accepts } w \text{ within } t \text{ steps}\}$$

Proposition 2.6 *$A_{\mathrm{NP}}$ is* NP*-complete.*

*Proof.* It is easy to see that $A_{\mathrm{NP}}$ is in NP. Given $\langle M, w, 1^t\rangle$ as input, we simulate $M$ on $w$ (non-deterministically) for $t$ steps and *accept* the input if $M$ accepts $w$. Clearly, this algorithm runs in non-deterministic polynomial time.

We next show that $A_{\mathrm{NP}}$ is NP-hard. Let $L \in$ NP via a non-deterministic TM $M$ that runs in time $O(n^c)$, where $c$ is some constant. Our reduction algorithm, given input $w$, simply outputs $\langle M, w, 1^{n^c}\rangle$. It is clear that $w \in L$ iff $\langle M, w, 1^{n^c}\rangle \in A_{\mathrm{NP}}$. Our reduction algorithm takes (deterministic) time $O(n^c)$, and hence runs in polynomial time. ♣

We note that this proof can be easily adapted to many other complexity classes, as long as the class has an enumeration by "clocked" machines. For example, for the class PSPACE, we can define an enumeration of space bounded (by $n^j + j$) TMs $M_{\langle i,j\rangle}$, and form such a "universal language", which will be complete for PSPACE.

$$A_{\mathrm{PSPACE}} = \{\langle M, w, 1^t\rangle |\ M = M_{\langle i,j\rangle} \text{ accepts } w \text{ within } t = |w|^j + j \text{ space}\}$$

Thus mere existence of complete languages for a standard complexity class which can be represented by an enumeration of TMs is no surprise. However these are not "natural" languages. Having shown the existence of an NP-complete problem, we turn our attention to "natural" or "real-world" problems that are NP-complete. The great importance of NP-completeness resides with these "natural" NP-complete problems. One such famous problem is SAT.

$$\mathrm{SAT} = \{\varphi | \varphi \text{ is a satisfiable boolean formula }\}$$

The famous Cook–Levin theorem states that SAT is NP-complete.

Theorem 2.7 (Cook–Levin) SAT *is* NP*-complete.*

The proof goes by showing that, given a polynomial time NTM $M$ and an input $w$, the computation of the machine $M$ on $w$ can be encoded into a boolean formula so that $M$ accepts $w$ if and only if the formula is satisfiable. Moreover, if the machine runs in polynomial time, the length of formula will also be polynomial in the input length. The basic idea is to use Boolean variables and logical connectives $\vee, \wedge, \neg$ to form a propositional logic formula, which is in CNF (a conjunction of some disjunctions of literals), which "says" the computation $M(x)$ accepts for some non-deterministic computational path. For a formal proof of the theorem, we refer the reader to standard textbooks [2, 3, 4]. It really should come as no surprise that boolean logic can express this. But the impact of realizing SAT is such a universal problem for NP is tremendous.

Following the work of Cook, Karp showed that many other "natural" problems are also NP-complete. It is with the realization of this abundance of "natural" NP-complete problems that the importance of NP-completeness is truly demonstrated. Apart from its theoretical importance, the notion of NP-completeness is also very useful from a practical perspective. If an NP-complete problem is shown to be solvable in polynomial time, then every problem in NP would also be solvable polynomial time, and hence, it would imply that NP = P. But, it is widely believed that NP $\neq$ P. One indication is perhaps that there are thousands of NP-complete problems and no poly-time algorithm has been found for any of them. (Although this is admittedly a weak argument in favor of the Conjecture.) So, showing a problem to be NP-complete is taken to be a strong indication that the problem is not solvable in polynomial time and hence computationally hard. In fact, following this scheme, over the past three decades, thousands of "practical" problems have been shown to be NP-complete. A book by Garey and Johnson [2], now a classic, includes a compendium of many of these NP-complete problems.

Going back to SAT, we saw that, given a boolean formula, it is NP-hard to decide whether it is satisfiable or not. In this formulation, we allow the input to be any arbitrary boolean formula. But in fact Cook's proof shows that it remains NP-hard, even if we restrict the input formula to be in conjunctive normal form (CNF) (i.e. the formula is an AND of ORs such as $(\overline{x_1}\vee\overline{x_2}\vee x_3\vee x_4)\wedge(x_1\vee\overline{x}_6))$. It is common to refer to the above version (CNF-SAT) simply as SAT.

Once we have established a certain problem $\Pi$ to be NP-complete, to prove the NP-completeness of another problem $\Pi'$ in NP, we can reduce the known NP-complete problem $\Pi$ to the given problem $\Pi'$. This is simply a consequence of the transitivity of Karp reductions. The task is made easier if we start with a problem known to be NP-complete, and it has as restrictive a form as possible. For example, it is easier to give a reduction from the CNF version of SAT than from the general one. Proceeding along these line, we next show that an even more restricted version of SAT, namely 3-SAT, is also NP-complete.

3-SAT is the set of all satisfiable boolean expressions of the form

$$\varphi = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{3} l_{i,j}$$

where the literal $l_{i,j}$ is either $x$ or $\overline{x}$, for some input variable $x$. Thus, $(x_1\vee\overline{x}_2\vee x_3)\wedge(x_2\vee\overline{x}_3\vee x_4)$ is an example of a 3-SAT expression. We now reduce SAT to 3-SAT.

In order to transform SAT to 3-SAT, we use a construction termed the "butterfly construction". The idea is that for any clause that is longer than three components, we split the clause into clauses with 3 literals, by adding extra variables. Consider the following formula:

$$\varphi = (x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5 \vee x_6)$$

We simply split this formula into 3-clauses as follows:

$$\begin{aligned}
\varphi &\rightarrow (x_1 \vee x_2 \vee y_1) \wedge (\overline{y}_1 \vee x_3 \vee x_4 \vee x_5 \vee x_6) \\
&\rightarrow (x_1 \vee x_2 \vee y_1) \wedge (\overline{y}_1 \vee x_3 \vee y_2) \wedge (\overline{y}_2 \vee x_4 \vee x_5 \vee x_6) \\
&\rightarrow (x_1 \vee x_2 \vee y_1) \wedge (\overline{y}_1 \vee x_3 \vee y_2) \wedge (\overline{y}_2 \vee x_4 \vee y_3) \wedge (\overline{y}_3 \vee x_5 \vee x_6)
\end{aligned}$$

Call the last formula $\varphi'$. Now we notice the following: If we have assigned all $x_i$ to be false in $\varphi$, then in order to satisfy $\varphi'$, we can reason from left to right in $\varphi'$ one clause at a time, and we see that all $y_j$ must be true: $x_1, x_2$ are false, so $y_1$ is true; then $\overline{y}_1, x_3$ are false, and so $y_2$ is true; etc. But still this would leave the last clause false. (Imagine an air bubble being pushed from left to right.)

On the other hand, if we have assigned some $x_i$ to be true, then we can "propagate" from this $x_i$ on both sides by assigning each $y_j$ appropriately so that all 3-literal clauses are satisfied. For example suppose $x_3$ is true. Then it looks something like the following, where $\uparrow$ represents true and $\downarrow$ represents false for the whole literal (including the not symbol):

This is the motivation for calling it the butterfly construction.

| $y_1$ | $\overline{y}_1$ | $x_3$ | $y_2$ | $\overline{y}_2$ | $y_3$ | $\overline{y}_3$ |
|---|---|---|---|---|---|---|
| ↑ | ↓ | ↑ | ↓ | ↑ | ↓ | ↑ |

Given any instance of SAT in CNF, we apply the same transformation to each clause with length greater than 3 by adding new variables for each clause. If a clause has fewer than 3 literals, there is a trivial transformation to make a conjunction of 3-clauses equivalent to it. Then it is clear that the original formula is satisfiable iff the transformed formula is satisfiable.

We have proved the following theorem.

THEOREM 2.8 *3-SAT is* NP-*complete.*

One can also easily prove that the problem remains NP-complete if we restrict to SAT instances with exactly $k$ literals per clause, for any $k \geq 3$. This is called $k$-SAT problem. It is interesting to note that 2-SAT is decidable in polynomial time as it can be formulated as a graph of logical implications and one can find if the implications are consistent or not in polynomial time.

**Exercise:** Show that 2-SAT is in P.

With the NP-completeness of 3SAT in hand, we can show that many other problems are NP-Complete by reducing 3-SAT to them.

### 2.2.1 NP-completeness of VertexCover, IndependentSet, Clique

We begin by defining VertexCover, or VC:

$$\text{VC} = \{\langle G, k\rangle \mid \exists S \subseteq V(G), |S| \leq k, \text{ every edge of } G \text{ is incident to at least one vertex of } S\}$$

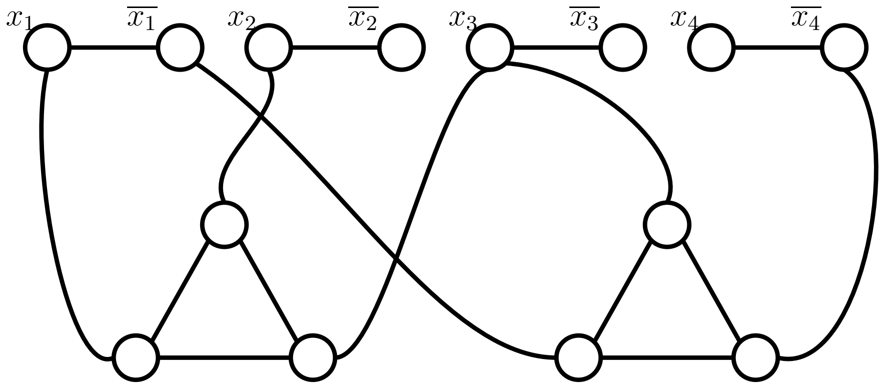
We now utilize a gadget-based construction to reduce SAT to VertexCover. The idea of such a construction is to create objects in one system (in this case, we'll be creating specialized graphs) that correspond to the fundamental objects in another (clauses and variables), such that a certain relation among the constructed objects in the first system exists if and only if another relation exists among all objects in the second. In the case of the construction for VC, we want certain vertices to be covered iff the assignments to variables they represent correspond to a satisfying assignment.

Now, we give the construction. We reduce 3Sat to VC. We will find the limitation on the size of the clauses to be advantageous here. Assume there are $n$ variables $x_1, x_2, \ldots, x_n$ and $m$ clauses. First, we construct vertices $x_1, \overline{x_1}, x_2, \overline{x_2}, \ldots, x_n, \overline{x_n}$. These vertices will correspond to the possible truth assignments to the $n$ variables present in the formula. We add edges $(x_1, \overline{x_1}), (x_2, \overline{x_2}), \ldots, (x_n, \overline{x_n})$, which forces the constraint that we must select at least one value for each variable. We call these the intra-variable edges.

Then, we construct vertices $\hat{l}_{11}, \hat{l}_{12}, \hat{l}_{13}, \hat{l}_{21}, \hat{l}_{22}, \hat{l}_{23}, \ldots, \hat{l}_{m1}, \hat{l}_{m2}, \hat{l}_{m3}$, which correspond to the literals of each clause. These nodes are connected in a triangle for each clause: $(\forall i, 1 \leq i \leq m)\{(\hat{l}_{i1}, \hat{l}_{i2}), (\hat{l}_{i2}, \hat{l}_{i3}),\ (\hat{l}_{i3}, \hat{l}_{i1}))\}$. We call these the intra-clause edges.

Finally, each literal node $\hat{l}_{ij}$ in a clause is connected to its corresponding vertex $l_{ij}$. For example, if $l_{ij}$ is $x_k$, we have $(\hat{l}_{ij}, x_k)$; and similarly, if $l_{ij}$ is $\overline{x_k}$, it will be $(\hat{l}_{ij}, \overline{x_k})$. We call these the inter-formula edges.

Here is how $(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_3 \vee \overline{x_4})$ would be represented:



Now we'll step back and reason about our construction. Suppose the formula represented is satisfiable. Then we can construct a vertex cover of size $n + 2m$. We pick the $n$ vertices corresponding to a satisfying truth assignment $\sigma$. (If $x_1$ is set to "true" in $\sigma$, include the vertex labelled $x_1$ in the vertex cover, else include the one labelled $\overline{x_1}$.) Then, in each triangle corresponding to a clause, we cover two literals which include all literals which are set false. Note that if the formula is satisfied under $\sigma$, then any clause has at least one literal which is true, hence at most two literals which are false. With one vertex for each pair of $x_i$ and $\overline{x_i}$, and 2 vertices in each triangle, we note that all of the intra-variable and intra-clause edges will be covered no matter what the assignment. We note also that all the inter-formula edges are covered as well, if $\sigma$ is a satisfying assignment: An inter-formula edge connecting a variable node to a satisfied literal is covered by the variable node; an inter-formula edge connecting a variable node to an unsatisfied literal is covered by the literal node in the triangle for the clause. So we have a vertex cover of $n + 2m$ vertices.

Conversely, suppose there exists a covering with $n + 2m$ vertices. We show that this covering must represent a satisfying assignment. In order to cover all of the intra-variable and intra-clause edges, we need at least $n+2m$ vertices covered, 1 for each variable and 2 for each clause. Therefore these $n+2m$ vertices consist of exactly one for each pair $(x_i, \overline{x_i})$, and exactly two for each triangle. But this implies that only 2 of the 3 inter-formula edges for each clause are covered by literal nodes in the triangle for the clause, the third inter-formula edge must be covered by the node representing the literal. Hence, if we assign to true any literal precisely when it is in the vertex cover, the formula is satisfied.

Therefore, the formula is satisfiable iff there exists a cover of size $n+2m$. We have proved

THEOREM 2.9 *VC is* NP*-complete.*

We also note that INDEPENDENTSET is NP-Complete. First we define the problem:

$$\text{IS} = \{\langle G, \ell \rangle \mid \exists S \in V(G), |S| \geq \ell \text{ such that } \forall u, v \in S : (u, v) \notin E(G)\}$$

Observe that, if $C$ is a vertex cover of a graph $G$, then its complement $V - C$ is an independent set, and vice versa. Thus, a graph has a vertex cover of size $\leq k$ if and only if it has an independent set of size $\geq n - k$. We conclude that INDEPENDENTSET is NP-complete. They are essentially the same problem.

THEOREM 2.10 *IS is* NP*-complete.*

Finally, we note that CLIQUE is NP-Complete. CLIQUE is also essentially the same problem as INDEPENDENTSET:

$$\text{CLIQUE} = \{\langle G, \ell \rangle \mid \exists S \in V(G), |S| \geq \ell \text{ such that } \forall u, v \in S : (u, v) \in E(G)\}$$

As such, by flipping the edges of $G$ (construct a $G'$ such that $V(G') = V(G)$ and $(u, v) \in E(G') \iff (u, v) \notin E(G)$), we see that by the VC reduction, CLIQUE is also NP-Complete.

THEOREM 2.11 CLIQUE *is* NP*-complete.*

### 2.2.2 HAMILTONIANCIRCUIT

Another graph problem we can define is HAMILTONIANCIRCUIT, or HC. We define a Hamiltonian circuit to be an ordering of the vertices of a graph $G$ as $v_1, v_2, \ldots, v_n$, such that we can travel through all the vertices once and return to where we started: $(v_1, v_2), (v_2, v_3), \ldots,$ $(v_{n-1}, v_n)$, $(v_n, v_1) \in E(G)$. Then,

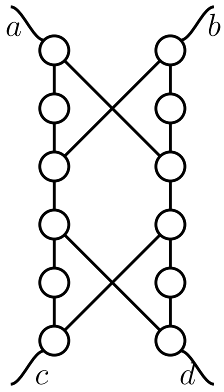
$$\text{HC} = \{\langle G \rangle \mid \exists \text{ a Hamiltonian circuit in } G\}$$

It is straightforward that HC is in NP, as we can verify that an ordering of the vertices represents a cycle in polynomial time.

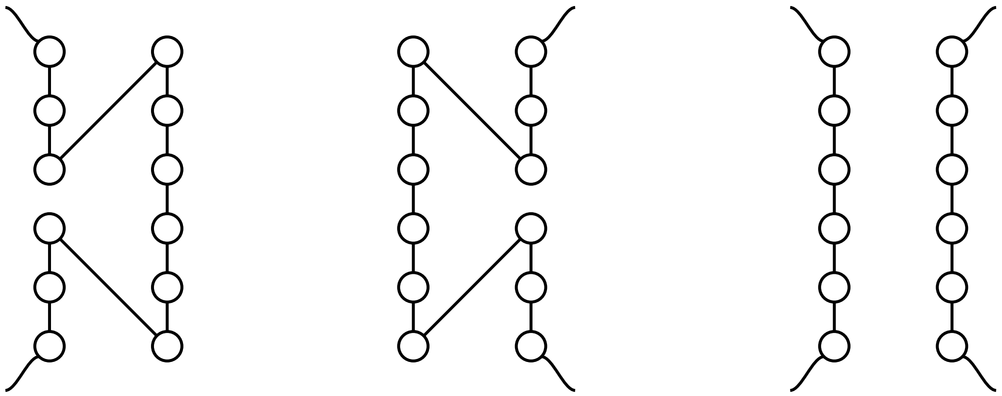
#### Reduction from VC

What follows is a proof of the NP-Completeness of HC. As with VC, we will utilize a gadget-based construction. However, instead of reducing from 3SAT, we will reduce from VC.

Consider the following graph, which we will refer to as our gadget:



There are only three ways to traverse this graph (which can be verified through inspection): (1) An $a$-$b$-$d$-$c$ pass— enter through $a$ and exit through $c$, passing through $b$ and $d$; or (2) a $b$-$a$-$c$-$d$ pass— enter through $b$ and exit through $d$; or (3) traverse in two passes: first in an $a$-$c$ pass, and then in a $b$-$d$ pass. In all cases reordering of entry and exit is possible. Here are the three graphs corresponding to these paths:



Given $G$, an instance for VC, we construct $G'$, an instance for HC. For each edge $e = (u, v)$ in $G$, we use one such gadget in $G'$ representing the edge $e$. The left half of this gadget represents $u$ and the right half represents $v$. For each vertex $u$, we link together in some arbitrary order, top to bottom, all of the $u$ halves of the gadgets that represent edges incident to $v$. (We'll refer to this as a *linked vertex chain.*) Finally, let $k$ be the size of the VC in question, then we add vertices $a_1, \ldots, a_k$ and connect each of them to both the start and end of each linked vertex chain.

The intention is to represent how the vertices are covered in $G$ by the traversals of each linked vertex chain emanating from $a_1, \ldots, a_k$ in $G'$. In the above pictures, if $u$ is one of the $k$ vertices in the VC, then for each gadget on a linked vertex chain for the vertex $u$, we will traverse either all the vertices (as in the first choice of traversal) when $(u, v)$ is covered by $u$ alone, or half of the vertices (as in the third choice of traversal) when $(u, v)$ is covered by both $u$ and $v$. If $u$ is not in the VC, but $v$ is, then all the vertices of this gadget will be traversed as in the middle choice of traversal.

We now give a short argument for the correctness of this construction. First, we show how to traverse the graph if there is a vertex cover of size $k$. We note that if there is a cover of size less than $k$, it can be easily extended into a cover of size exactly $k$ simply by adding extra vertices to the optimal cover, as they cannot hurt. So we assume $k \leq n$ and have a cover of size $k$. For the $i$th vertex (say $u$) in the cover, we start at $a_i$, trace out the linked vertex chain for $u$ before returning to $a_{i+1}$, and if $i = k$ we return back to $a_1$, completing the tour. For each linked vertex chain, say of $u$, we traverse each gadget corresponding to an edge $(u, v)$ as follows. If the edge is covered by $u$ alone, then we traverse in an *a*-*b*-*d*-*c* pass (where the *a*-*c* side corresponds to $u$), and if the edge is covered by both $u$ and $v$, then we traverse in an *a*-*c* pass. Clearly, from a VC of size $k$ we get a Hamiltonian Circuit.

Now, we claim that if we have a cycle for the graph $G'$, there is a vertex cover of size $k$ in the original graph $G$. If we had a cycle, it would have to include all of the $a_i$ vertices and all of the gadget vertices. But we know that in between each of the visits to the $a_i$ vertices, we can only visit one linked vertex chain corresponding to one vertex $u$ in $G$. We claim that these $k$ vertices form a vertex cover in $G$. For each gadget on this linked vertex chain of $u$ we can see by the way it was visited which edges the vertex $u$ covers in $G$. By the way how each gadget can be traversed, it follows that indeed these $k$ vertices form a vertex cover in $G$.

THEOREM 2.12 *HC is* NP-*complete.*

## 2.3 Polynomial Hierarchy

We defined NP to be the class of all languages accepted by non-deterministic polynomial time TMs. We can give an equivalent characterization of NP.

DEFINITION 2.13 ($\Sigma_1^p$) *A language $L$ is in $\Sigma_1^p$ if there exists a boolean binary predicate $D(\cdot, \cdot)$ computable in (deterministic) polynomial time, and a polynomial $q$ such that, for all input $x$,*

$$x \in L \iff \exists y \in \{0,1\}^{q(|x|)}[D(x,y) = 1]$$

It is easy to see that NP $= \Sigma_1^p$. Suppose $L \in \Sigma_1^p$. We design an NP machine that given an input $x$, simply guesses a string $y$ of length $q(|x|)$ and then accepts if $D(x, y) = 1$. On the other hand, suppose $L \in$ NP, via an NP machine $M$ that runs in time $q(n)$. Then, for any input $x$,

$$x \in L \iff (\exists \text{ a computational path } p \text{ of length } q(|x|)\ )[\ M \text{ accepts } x \text{ along the path } p\ ]$$

Clearly, $L \in \Sigma_1^p$.

In the above formulation of NP, the predicate $D$ is usually called a *verifier* and if $D(x, y) = 1$, then $y$ is called a *certificate* or *proof* for the input $x$. For example, in the case of SAT, we can construct a verifier as follows. Given a formula $\varphi$ and a truth assignment $\sigma$, accept $(\varphi, \sigma)$, if $\sigma$ satisfies $\varphi$. Any satisfying truth assignment of $\varphi$ is a certificate.

We can generalize the definition of $\Sigma_1^p$ to define a larger class as follows.

Definition 2.14 ($\Sigma_2^p$) *A language $L$ is in $\Sigma_2^p$ if there exists a boolean predicate $D$ computable in polynomial time, such that*

$$x \in L \Longleftrightarrow \exists y \forall z [D(x, y, z) = 1]$$

*where $|y|$ and $|z|$ are polynomially bounded in $|x|$.*

We will denote by $\exists^p y$ for $\exists y$ where $|y|$ is polynomially bounded in $|x|$. By padding we may assume there exists some polynomial $q(\cdot)$, such that $\exists^p y$ means $\exists y \in \{0, 1\}^{q(|x|)}$. Again by padding we may assume the same polynomial $q(\cdot)$ works for $\exists^p y$ and $\forall^p z$.

We next give a natural example of a language in $\Sigma_2^p$. We say that a boolean formula $\varphi$ is minimal if there are no shorter formulas that compute the same boolean function. For example, the formula $(x_1 \vee x_2) \vee (x_1 \vee x_3)$ is not minimal, because it has a smaller equivalent formula: $x_1 \vee (x_2 \vee x_3)$. Let MEE (Minimum Equivalent Expression) represent the set of all minimal boolean formulas, i.e., a boolean expression for which there are no shorter boolean expressions equivalent to it. Then the language $\overline{\text{MEE}}$, the complement of MEE, is a language in $\Sigma_2^p$. We can express this language as,

$$\varphi \in \overline{\text{MEE}} \iff \exists \psi \forall \sigma [|\psi| < |\varphi| \text{ and } \varphi(\sigma) = \psi(\sigma)]$$

In the above definition, $\sigma$ refers to various truth assignments and $\varphi(\sigma)$ is the value of the formula for the truth assignment $\sigma$.

Definition 2.15 ($\Pi_2^p$) *If $L$ is in $\Sigma_2^p$, we say that $L^c$, the complement of $L$, is in $\Pi_2^p$. Equivalently, $L$ is in $\Pi_2^p$ if there exists a boolean predicate $Q$ computable in polynomial time, such that*

$$x \in L \Longleftrightarrow \forall y \exists z [Q(x, y, z) = 1]$$

*where $|y|$ and $|z|$ are polynomially bounded in $|x|$.*

A natural example for the class $\Pi_2^p$ is MEE, the set of all minimal propositional boolean formula. We can express MEE as,

$$\varphi \in \text{MEE} \Longleftrightarrow \forall \psi \exists \sigma [|\psi| < |\varphi| \longrightarrow \psi(\sigma) \neq \varphi(\sigma)]$$

We extend the above definitions to define the classes $\Sigma_k^p$ and $\Pi_k^p$. For any integer $k \geq 0$, we define

Definition 2.16 ($\Sigma_k^p$ and $\Pi_k^p$) *A language $L$ is in $\Sigma_k^p$ if there exists a $(k+1)$-ary boolean predicate $D$ computable in polynomial time, such that*

$$x \in L \iff \exists^p y_1 \forall^p y_2 \ldots Q^p y_k [D(x, y_1, y_2, \ldots, y_k) = 1]$$

*where $Q^p$ is either $\exists^p$ for $k$ odd or $\forall^p$ for $k$ even, and the superscript $p$ denotes that all $|y_i|$ are polynomially bounded in $|x|$. In the above definition, $\exists$ and $\forall$ alternate. We say that a language $L$ is in $\Pi_k^p$, if its complement $L^c$ is in $\Sigma_k^p$.*

It is clear that

Lemma 2.17 *For any integer $k \geq 0$,*

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

Definition 2.18 (Polynomial Time Hierarchy (PH)) PH$= \bigcup_k \Sigma_k^p = \bigcup_k \Pi_k^p$

A picture of the polynomially time hierarchy is shown in Figure 2.1.

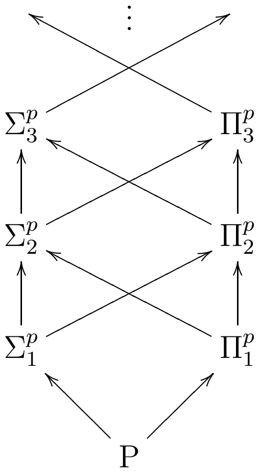


Figure 2.1: Polynomial Time Hierarchy

It is a big open problem whether the polynomially hierarchy is infinite, i.e., whether it has infinitely many distinct levels. It is quite conceivable that the hierarchy collapses to some $k^{th}$ level, meaning, for all $r \geq k$, $\Sigma_r^p = \Sigma_k^p$. Then as we shall see, equivalently, PH $= \Sigma_k^p$. But, most people believe that it is infinite. One can prove a simple result related to collapsing the hierarchy.

Theorem 2.19 *If $\Sigma_k^p = \Pi_k^p$ for some $k$, then* PH$= \Sigma_k^p = \Pi_k^p$ *(the polynomial time hierarchy collapses to the level $k$.)*

**Comment:** We note that $\Sigma_k^p = \Pi_k^p$ implies, and is implied by $\Sigma_k^p \subseteq \Pi_k^p$, as well as $\Pi_k^p \subseteq \Sigma_k^p$. For, say $\Sigma_k^p \subseteq \Pi_k^p$ and $L \in \Pi_k^p$, then $L^c \in \Sigma_k^p$. Thus $L^c \in \Pi_k^p$. Hence $L = (L^c)^c \in \Sigma_k^p$.

*Proof.* To get an idea of the proof, let us suppose that $\Sigma_1^p = \Pi_1^p$. We want to prove that $\Sigma_2^p \subseteq \Sigma_1^p$. Then by Lemma 2.17, $\Sigma_2^p \subseteq \Pi_2^p$, and thus $\Sigma_2^p = \Pi_2^p$, by the comment above.

Consider a language $L \in \Sigma_2^p$. Now for a string $x$,

$$x \in L \iff \exists^p y \forall^p z P(x, y, z)$$

Rewrite this in terms of another predicate $Q(\langle x, y\rangle, z)$, where $Q$ first "unravels" the $\langle x, y\rangle$ into $x$ and $y$ and calls $P$. Hence, we have

$$x \in L \iff \exists^p y \forall^p z Q(\langle x, y\rangle, z)$$

Observe however, that $\{\langle x, y\rangle | \forall^p z Q(\langle x, y\rangle, z)\}$ is a language in $\Pi_1^p$, which is equal to $\Sigma_1^p$ by our assumption. Hence we can write the language as $\{\langle x, y\rangle | \exists^p z E(\langle x, y\rangle, z)\}$ for some other polynomial deterministic predicate $E$. So, we have,

$$x \in L \iff \exists^p y \exists^p z E(\langle x, y\rangle, z)$$

which can be rewritten as

$$x \in L \iff \exists^p \langle y, z\rangle E'(x, \langle y, z\rangle)$$

where the predicate $E'$ does the following: it first unravels the $\langle y, z\rangle$ into $y$ and $z$, and then pairs $x$ and $y$ to form $\langle x, y\rangle$. It then calls $E$ with $\langle x, y\rangle$ and $z$. This shows that $L$ is a language in $\Sigma_1^p$, and hence, $\Sigma_2^p \subseteq \Sigma_1^p$.

In the general case, a similar proof works by induction that $\Sigma_k^p = \Pi_k^p$ implies that $\Sigma_{k+1}^p = \Pi_{k+1}^p$. ♣

The notion of an oracle plays an indispensable role in complexity theory. Next, we discuss oracles. Later we shall present an equivalent characterization of PH using oracles.

## 2.4 Oracle Turing Machines

An *oracle Turing machine* is a multi-tape Turing machine with a special tape called the query tape. The TM also has three special states called $q_?$ (the query state), $q_{yes}$ and $q_{no}$ (the answer states). Let $A$ be a predetermined language. The computation of an oracle TM with oracle $A$ is defined as follows. During its computation, the TM may write a string $z$ on the query tape and enter the query state. In the next step, the machine will enter the state $q_{yes}$ if $z \in A$ and will enter the state $q_{no}$ if $z \notin A$. Informally, whether or not the query string $z$ is in $A$ is determined automatically. We say that the TM asks the query to an oracle for $A$ and the oracle "magically" gives the correct answer in one step. Aside from this ability to ask the oracle about a query string, the TM is otherwise the same as before.

The idea is to abstract away that part of complexity of determining membership in $A$ from the complexity for the machine. The machine is bounded by its own complexity bound for all the other steps in its computation, including to decide what string to ask the oracle and to write down the query. In particular, if the machine is polynomial time bounded, it can only make queries of polynomially bounded length.

Another way to explain the concept of oracle is to think in terms of algorithms. An algorithm $M$, with an oracle $A$, denoted $M^A$, is like a usual algorithm, except that, it gets a "magic box" that would answer any question $M$ can ask regarding membership in $A$, in unit time. We think of the oracle as a procedure, and $M$ can make function calls to the procedure. (Strictly speaking though, an oracle $A$ need not even be a computable set.)

We define $\mathrm{P}^A$ to be the set of all languages that can be computed in deterministic polynomial time, given oracle access to the language $A$. $\mathrm{NP}^A$ is defined similarly. We can define this notion of oracle to classes other than P and NP, as we shall see later. We can further generalize, to have complexity classes as oracles. For example, if $\mathcal{C}$ is a complexity class, $\mathrm{NP}^{\mathcal{C}}$ is defined to be,

$$\mathrm{NP}^{\mathcal{C}} = \bigcup_{A \in \mathcal{C}} \mathrm{NP}^A$$

We discussed Karp reduction (or $p$-time many-one reduction) before. A different reduction called Cook reduction (or $p$-time Turing reduction) can be defined via oracles. We say that a language $B$ *Cook reduces* to a language $A$ in polynomial time, if there is a deterministic polynomial time algorithm that computes $B$, given $A$ as an oracle. We denote this by $B \leq_T^p A$. In other words, $B \leq_T^p A$ iff $B \in \mathrm{P}^A$.

## 2.5 A Characterization of PH

THEOREM 2.20 $\mathrm{NP}^{\mathrm{SAT}} = \Sigma_2^p$

*Proof.* We first show that $\Sigma_2^p \subseteq \mathrm{NP}^{\mathrm{SAT}}$. Let $L$ be a language in $\Sigma_2^p$. So there is a polynomial time computable predicate $D(\cdot, \cdot, \cdot)$ such that,

$$x \in L \iff \exists^p y \forall^p z [D(x, y, z) = 1]$$

where the lengths of $y$ and $z$ are bounded by $q(|x|)$ for some polynomial $q$. We can design a NP oracle machine $M$ that can decide $L$ when given an oracle for SAT. Given $x$, $M$ first guesses a string $y$, then asks the question "Is it true that, $\forall^p z[D(x, y, z) = 1]$", where both $y$ and $z$ are polynomially bounded by $q(|x|)$. By Cook's Theorem of SAT being NP-complete, this can be transformed into a SAT query. (Strictly speaking an UNSAT query.) The SAT oracle can answer this question. If the answer from SAT is 'no', i.e., $\forall^p z[D(x, y, z) = 1]$", $M$ accepts $x$. Such a $y$ will exist iff $x \in L$. So $M$ decides $L$ correctly.

We next show that $\mathrm{NP}^{\mathrm{SAT}} \subseteq \Sigma_2^p$. Let $L \in \mathrm{NP}^{\mathrm{SAT}}$ and $M$ be an NP Turing machine that decides $L$, given access to a SAT oracle. Each path in $M$ may ask many queries to the oracle. We first construct an equivalent NP machine $M'$ that asks at most one SAT question on each computational path, and it makes the query at the end of the path.

Consider a single path in $M$. This path may ask multiple questions to the oracle. Consider the first such question "$\varphi \in$ SAT?". At this point $M'$ does not ask the oracle a question. Instead, it guesses either "yes" or "no" as a non-deterministic move. On the branch it guesses "yes", it proceeds to guess a satisfying assignment to the queried formula $\varphi$. If the guessed assignment satisfies $\varphi$, then $M'$ proceeds with the computation of $M$ with $q_{yes}$. If the guessed assignment does not satisfy $\varphi$, then this non-deterministic rejects. On the branch it guesses "no" it simply takes this "no" as the oracle answer and proceeds with $q_{no}$ in $M$, while remembering that it guessed "$\varphi \notin$ SAT". Subsequent queries to SAT are handled similarly.

Consider any path in $M'$ that survived till the end and completed simulating a path in $M$. This path is similar to the path in $M$, except that it has assumed that certain formulae were unsatisfiable. Note that the fact that it has survived implies that all its assumptions that certain formulae were satisfiable have already been verified. If the path in $M$ being simulated *rejects*, then $M'$ should do so as well. Now suppose the path in $M$ being simulated *accepts*. The path in $M'$ has to decide whether or not to accept. The path (since it has survived) would have verified that the formulae it had assumed to be satisfiable were indeed satisfiable. It now needs to verify whether the formulae it assumed unsatisfiable are indeed unsatisfiable. It does this by asking a *single* SAT query. Suppose the formulas were $\varphi_1, \varphi_2, \ldots, \varphi_k$. $M'$ asks the query $\varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_k$. Notice that all the formulae are unsatisfiable iff the above formula is unsatisfiable. If the oracle confirms that all formulae guessed to be unsatisfiable are indeed unsatisfiable, then this path accepts.

Now $M'$ has an accepting path iff there is an accepting path in $M$. Note that an accepting path of $M'$ corresponds to some valid computational path of $M$, since all the guessed answers to the queries are verified at the end before $M'$ accepts. Now the acceptance of $M'$ can be formulated as a $\Sigma_2^p$ expression: $\exists$ a path $p$ in $M'$ such that $p$ accepts and for all truth assignments $\sigma$, the formula asked at the end of $p$ is not satisfied by $\sigma$. ♣

One can extend Theorem 2.20 to other levels of the hierarchy. The above proof gives the gist of the idea, "guess, delay and verify" queries. However, to carry this out formally for all levels of PH, it is more expedient to use some structural formalism. This is a taste of that branch of complexity theory called *Structural Complexity Theory*. There is a certain elegance to the generality of the arguments given; on the negative side, one can be blinded by the formalism and overlook the "real" point of the argument. Here, despite all the formalism, the "real" point of the argument is "guess, delay and verify" queries.

We define an operator $\exists$.

Definition 2.21 *Given any language $L$, and a polynomial $p$, define*

$$\exists . L = \{x \mid (\exists^p y)\langle x, y\rangle \in L\},$$

*and for any class $\mathcal{C}$,*

$$\exists.\mathcal{C} = \{\exists.L \mid L \in \mathcal{C}\},$$

*where as usual $\exists^p y$ denotes $\exists y \in \{0,1\}^{p(|x|)}$.*

Similarly we can define the operator $\forall$.

Definition 2.22 *Given any language $L$, and a polynomial $p$, define*

$$\forall.L = \{x \mid (\forall^p y)\langle x, y\rangle \in L\},$$

*and for any class $\mathcal{C}$,*

$$\forall.\mathcal{C} = \{\forall.L \mid L \in \mathcal{C}\},$$

*where as usual $\forall^p y$ denotes $\forall y \in \{0,1\}^{p(|x|)}$.*

Clearly $\mathrm{co}\forall.\mathcal{C} = \exists.\mathrm{co}\mathcal{C}$. Also $\exists.\forall.\mathrm{P} = \Sigma_2^p$, etc.

We want to prove the following theorem.

Theorem 2.23 *For any $k$,*

$$\mathrm{NP}^{\mathrm{NP}^{\cdot^{\cdot^{\cdot^{\mathrm{NP}}}}}} = \Sigma_k^p,$$

*where the height of the "tower" on LHS is $k$ (the number of NP's). This is also true relativized to any oracle.*

This is established in a sequence of claims.

Claim 1: for any oracle set $A$,

$$\mathrm{NP}^A = \exists.\mathrm{P}^A$$

It is trivial that $\exists.\mathrm{P}^A \subseteq \mathrm{NP}^A$. The NP machine simply guesses the string in the definition of $\exists.$ and does the $\mathrm{P}^A$ computation.

Given a language $L = L(N^A) \in \mathrm{NP}^A$ for some NP machine $N$, we want to put its acceptance criterion in the form of $\exists.\mathrm{P}^A$. It would be quite trivial if $N$ only asks its queries at the end of a computational path. So, we modify $N$ to $N'$, where on each path $N'$ simulates $N$ and, whenever $N$ makes a query $N'$ simply guesses an answer. At the end of a computational path of $N$, $N'$ accepts iff $N$ accepts and all the query guesses are correct. ♣

Claim 2: for any oracle set $A$,

$$\mathrm{NP}^{\exists.\mathrm{P}^A} = \exists.\forall.\mathrm{P}^A$$

One direction is easy, $\exists.\forall.\mathrm{P}^A \subseteq \mathrm{NP}^{\exists.\mathrm{P}^A}$. A simulating oracle NP machine can guess the string in $\exists.$, and use its oracle in $\exists.\mathrm{P}^A$ to answer queries in $\forall.\mathrm{P}^A$. Note that oracles in complementary classes have the same power.

Now consider any $L = L(N^B)$, where $B \in \exists.\mathrm{P}^A$. Let $B = \{w \mid (\exists^p y)D(w, y)\}$, where $D$ is in $\mathrm{P}^A$. We wish to express the acceptance of $N^B$ on $x$ in $\exists.\forall.\mathrm{P}^A$. So we will guess ($\exists$) a polynomially long non-deterministic path $p$, as well as polynomially many answer bits $b_1, \ldots, b_m$, and some $y_i$ (one for each $b_i = 1$ for the $i$ th query "$w_i \in B$?"), such that, ($\forall$) for any $z_j$ (one for each $b_j = 0$ for the $j$ th query "$w_j \in B$?"), the following predicate holds which is computable in $\mathrm{P}^A$— Machine $N$ on $x$ along path $p$ asks some $m$ queries $w_1, \ldots, w_m$ such that for the $i$ th query, if $b_i = 1$ then $D(w_i, y_i)$, if $b_i = 0$ then $\neg D(w_i, y_i)$, and, given the answers being $b_1, \ldots, b_m$, $N$ accepts $x$. (Note this last use of "some $m$ queries ..." is not a use of quantifiers; the $\mathrm{P}^A$ computation simply follows the path $p$ to arrive at its first query $w_1$, and depending on $b_1$ and its verification by $D$, proceed to find its second query $w_2$, etc.) ♣

Of course since Claim 1 and 2 hold for all oracles $A$, it also holds when we replace $A$ by any class $\mathcal{C}$.

Claim 3: for any oracle A,

$$\mathrm{NP}^{\mathrm{NP}^{\cdot^{\cdot^{\cdot^{\mathrm{NP}^A}}}}} \subseteq \Sigma_k^{p,A},$$

where the number of NP in the tower on LHS is $k$.

If $k = 1$ or 2, Claim 3 is direct from Claim 1 and 2. Assume $k > 2$. Claim 1 gives

$$\mathrm{NP}^{\mathrm{NP}^{\cdot^{\cdot^{\cdot^{\mathrm{NP}^A}}}}} \subseteq \mathrm{NP}^{\exists.\mathrm{P}^{\mathrm{NP}^{\cdot^{\cdot^{\cdot^{\mathrm{NP}^A}}}}}},$$

where the height of the "tower" of NP on LHS is $k$ and the number of NP on top of P on RHS is $k - 2$. Apply Claim 2 to RHS we get

$$\mathrm{NP}^{\exists.\mathrm{P}^{\mathrm{NP}^{\cdot^{\cdot^{\cdot^{\mathrm{NP}^A}}}}}} \subseteq \exists.\forall.[\mathrm{P}^{\mathrm{NP}^{\cdot^{\cdot^{\cdot^{\mathrm{NP}^A}}}}}] \tag{2.1}$$

where the number of NP on top of P on the RHS is $k - 2$. This tower is clearly contained in $\mathrm{NP}^{\mathrm{NP}^{\cdot^{\cdot^{\cdot^{\mathrm{NP}^A}}}}}$ with a tower of NP of height $k - 1$, which, by induction is contained in $\Sigma_{k-1}^{p,A}$. However $\mathrm{P}^{\mathrm{NP}^{\cdot^{\cdot^{\cdot^{\mathrm{NP}^A}}}}}$ is closed under complement, so that it is also contained in $\Pi_{k-1}^{p,A}$. Now "plug in" in (2.1) we get

$$\mathrm{NP}^{\mathrm{NP}^{\cdot^{\cdot^{\cdot^{\mathrm{NP}^A}}}}} \subseteq \exists.\forall.\Pi_{k-1}^{p,A} = \Sigma_k^{p,A}$$

♣

Now the reverse direction.

Claim 4: for any oracle A,

$$\mathrm{NP}^{\mathrm{NP}^{\cdot^{\cdot^{\cdot^{\mathrm{NP}^A}}}}} \supseteq \Sigma_k^{p,A},$$

where the height of the "tower" of NP on LHS is $k$.

This is simple: RHS is $\exists.\Pi^{p,A}_{k-1}$. By induction $\Pi^{p,A}_{k-1} \subseteq \mathrm{coNP}^{\cdot^{\cdot^{\mathrm{NP}^{A}}}}$ where the tower has height $k-1$. But as oracles, complementary classes have the same power, so that $\mathrm{NP}^{\mathcal{C}}$ can handle $\exists.\Pi^{p,A}_{k-1}$, where $\mathcal{C} = \mathrm{NP}^{\cdot^{\cdot^{\mathrm{NP}^{A}}}}$ where the tower in $\mathcal{C}$ has height $k-1$. $\clubsuit$

Claim 3 and 4 proves Theorem 2.23.

## 2.6 Complete Problems for $\Sigma^p_k$

We have seen canonical complete languages for the classes NP and PSPACE (see $A_{\mathrm{NP}}$ and $A_{\mathrm{PSPACE}}$ defined in Section 2.2). Similarly, one can construct canonical complete languages for each level of the polynomial hierarchy $\Sigma^p_k$. In this section, we exhibit "natural" complete languages for each $\Sigma^p_k$.

As usual, we say that a language $L$ is complete (under polynomial time reductions) for $\Sigma^p_k$, if $L \in \Sigma^p_k$ and every language in $\Sigma^p_k$ Karp reduces to $L$. The familiar Cook's theorem already gives us a complete language for $\Sigma^p_1 = \mathrm{NP}$, namely the language SAT. We can get complete languages for higher levels of the hierarchy by generalizing SAT.

Definition 2.24 *A quantified boolean formula with $k$ alternations over a set of $n$ boolean variables $X = \{x_1, x_2, \ldots, x_n\}$ is an expression of the form*

$$F = \exists X_1 \forall X_2 \exists X_3 \ldots Q X_k [f(x_1, x_2, \ldots x_n)],$$

*where $f$ is a (quantifier free) boolean formula over the $n$ variables and $X_1, X_2, \ldots X_k \subseteq X$ is a partition of $X$. Here, $Q$ is the existential quantifier if $k$ is odd, and it is the universal quantifier if $k$ is even.*

We say that $F$ is *valid* if the above expression is logically true. Meaning, there exists some choice of values for the variables in $X_1$ such that, for all choices of values for $X_2$, there exists some choice of values for $X_3$ such that, so on ..., the formula $f$ evaluates to true.

Let $\mathrm{QBF}_k$ be the set of all valid quantified boolean formulas with $k$ alternations. It is quite easy to adapt the proof of Cook's theorem to show that for any $k \geq 1$, $\mathrm{QBF}_k$ is $\Sigma^p_k$-complete.

Theorem 2.25 *For any $k \geq 1$, $\mathrm{QBF}_k$ is $\Sigma^p_k$-Complete.*

## 2.7 Alternating Turing Machines

Another way of looking at the Polynomial Time Hierarchy is to use Alternating Turing Machines. An alternating TM is a nondeterministic Turing machine where each state is labeled as existential or universal. It functions the same way as a non-deterministic Turing machine does, but the notion of acceptance is defined differently.

Recall that the *configuration* of an ordinary Turing machine is a tuple consisting of the current state of the Turing machine, the symbols on the tape, and the position of the head. As the Turing machine processes the input, it moves from one configuration to another. In an Alternating Turing machine, we define an *existential configuration* as one in which the current state of the Turing machine is an existential state, and a *universal configuration* as one in which the state is a universal state.

Acceptance is defined recursively: A final configuration is an accepting or rejecting configuration iff the state is accepting or rejecting. For non-final configurations, the acceptance is defined as: An existential configuration of the Turing machine is an accepting configuration if there is an accepting configuration among its next configurations in one step. A universal configuration is an accepting configuration if all of its next configurations are accepting configurations. Finally $x$ is accepted iff the beginning configuration is an accepting configuration.

The relation between alternating Turing machines and quantifier alternations is as follows. The proof is left as an exercise.

THEOREM 2.26 $\Sigma_k^p$ *is precisely the set of languages $L$ accepted by polynomial time alternating Turing machines with $k$ alternations with the starting configuration as an existential configuration. Similarly,* $\Pi_k^p$ *is the set of languages $L$ accepted by polynomial time alternating Turing machines with $k$ alternations with a universal starting configuration.*

**Exercise:** Prove Theorem 2.26

# Chapter 3

# Space Bounded Computation

**Chapter Outline:** Configuration graphs and the graph accessibility problem. Savitch's Theorem. Immerman-Szelepcsenyi Theorem.

## 3.1 Configuration Graphs

We begin this chapter by discussing configuration graphs. They are very useful in conceptualizing the formal process of computation, and often realize this formal process in terms of concrete graph theoretic questions. They are particularly useful in studying space bounded computations.

Let $M$ be a non-deterministic TM and $x$ be an input of length $n$. At any point during the computation, the *configuration* of $M$, which determines all the relevant future steps of the computation, can be fully described by specifying the current state, the contents of the tape, and the current position of the tape-head. If the machine $M$ is time $T(n)$ bounded, then this information takes no more than $O(T(n))$ bits. For space bounded computation we use off-line TMs. Let $M$ be a $S(n)$ space bounded machine. Note that for space bounded machine the input tape is read only and its content does not change during the computation. Then we can specify the *configuration* of $M$ at any step by the current state ($O(1)$ bits), the position of the read-only head ($\log n$ bits), and the contents of the work tape together with its tape-head ($O(S(n))$ bits). If $S(n) = \Omega(\log n)$, as we will always assume in space bounded computations in these Lectures, we can describe a configuration using $O(S(n))$ bits.

Consider the configuration in which the tape content is the input $x$ (if $M$ is not an off-line TMs, then this $x$ is followed by blank symbols of a combined length equal to the time bound), the position of tape-head is 0, and the current state is the start state of $M$. This is the (unique) *starting configuration.* Also naturally, any configuration in which the current state is a final state of $M$ is called a *final configuration.* Technically TM computation is defined in terms of these configurations. One defines a binary relation on pairs of configurations:

$c \vdash c'$ iff in one step $c$ can be followed by $c'$, i.e., there is a legal move in one step from $c$ to $c'$. For deterministic machines, there can be at most one next configuration after any $c$; for non-deterministic machines there can be more than one. The number of such next valid configurations is completely specified by the transition function $\delta$, and thus there can be at most a constant number of them. This constant depends only on $M$, and not on the input $x$. We can extend these notions to Turing machines that use multiple tapes.

In either model the total number of possible configurations is $2^{O(f(n))}$, if $f$ is the time or space bound. Given an input $x$, we can construct a directed graph with these configurations as nodes. We add an edge from configuration $c$ to $c'$, if $c \vdash c'$. We then mark the start and final configurations on this graph. We call this graph the *configuration graph* of $M(x)$. Notice that whether there is an edge from $c$ to $c'$ can be determined by examining the given configurations and the transition rules of the machine. Given any two configurations $c$ and $c'$, this can be checked out easily, in time linear in $|c| + |c'|$, and in space $O(\log(|c| + |c'|)))$. It is easy to see

PROPOSITION 3.1 *Let $M$ be a non-deterministic Turing machine and $x$ be an input. $M$ accepts $x$ iff at least one final node is reachable from the start node in the configuration graph of $M(x)$.*

THEOREM 3.2 *For any function $f(n) = \Omega(\log n)$,* $\mathrm{NSPACE}[f(n)] \subseteq \mathrm{DTIME}[2^{O(f(n))}]$

*Proof.* Let $M$ be a non-deterministic machine that uses $O(f(n))$ space and $x$ be the input. To simulate the computation of $M$ on $x$ deterministically, we first construct the configuration graph $G$. We then check if there is path from the start configuration to any one of the final configuration. We can do this by depth first search. As the number of nodes is $2^{O(f(n))}$, we can do this in time $2^{O(f(n))}$. The correctness is ensured by Proposition 3.1. ♣

## 3.2 Savitch's Theorem

We next establish an important connection between non-deterministic and deterministic space. We shall show that any non-deterministic computation can be simulated in deterministic space with only a quadratic blow-up in space usage. All the formalism of computation and non-determinism aside, the crux of the matter is the following statement on a graph theoretic problem—The Graph Accessibility Problem (GAP ).

THEOREM 3.3 (SAVITCH) *Given a directed graph $G = (V, E)$ over $n$ nodes and two nodes $s$ and $t$, we can test whether there is a path from $s$ to $t$ in $O((\log n)^2)$ deterministic space.*

**Remark:** Notice that the usual depth first search will use $\Omega(n)$ space. For each node visited, DFS has to mark that it has been visited. Thus it needs at least $n$ bits of space. Of course DFS shows that GAP is in P.

*Proof.* Instead of DFS, we make the following trivial observation: There is a path from $u$ to $v$ with length at most $t$ iff there is some $w$, and there are paths from $u$ to $w$ and $w$ to $v$ of length at most $\lceil t/2 \rceil$ and $\lfloor t/2 \rfloor$ respectively. So we will use a recursive "middle-first" search.

First of all, if $t$ can be reached from $s$, then it can be done so using a path of length at most $n$ (actually $n-1$ suffices; for any path we can remove any cycle, and thus removing all repeated appearances of any vertex along the path.) Then we apply the above observation. We go through all possible choices for a potential middle $w$. We check the existence of the required two paths of length $n/2$ recursively. Given two nodes $x$ and $y$, and integer $t$ (in binary), the following algorithm checks if there is a path of length at most $t$ from $u$ to $v$.

```
REACH($u$, $v$, $t$)
        If $t = 1$ then
            If [($u = v$) or ($\langle u, v \rangle \in E$)] then *accept*
            else *reject*
        If $t > 1$ then
            For each node $x$ in $G$,
                If [REACH($u$, $x$, $t/2$) $\wedge$ REACH($x$, $v$, $t/2$)] then *accept*
        If haven't yet accepted, *reject*.
```

We can check if $t$ can be reached from $s$ by calling REACH($s$, $t$, $n$). One subtle issue is that $n$ has to be a power of 2 for our recursion to work well. If $n$ is not a power of 2, we call REACH($s$, $t$, $n'$), where $n' = 2^{\lceil \log n \rceil}$.

The algorithm has a recursion depth of $\lceil \log n \rceil$. Thus the recursion stack has to store information about $\log n$ recursive calls. For each call, we need to store $u$, $v$, $t$ and $x$ used in that call. This needs $O(\log n)$ space. Thus the total space used is $O((\log n)^2)$. ♣

We use the above theorem to prove

THEOREM 3.4 (SAVITCH) *For any space constructible function* $S(n) = \Omega(\log n)$*,* NSPACE$[S(n)] \subseteq$ DSPACE$[(S(n))^2]$.

*Proof.* Let $M$ be a non-deterministic machine that uses $O(S(n))$ space. We construct a deterministic machine $M'$ that uses $O((S(n))^2)$ space to simulate $M$. Given input $x$, $M'$ considers the configuration graph $G$ of $M(x)$. It uses the above algorithm for GAP to test whether any one of the final configurations of $G$ is reachable from the starting configuration. As there are $2^{O(S(n))}$ nodes in $G$, the space used by $M'$ is $O((S(n))^2)$. One subtle issue is that $M'$ cannot construct the whole $G$, as that would require exponential space. So, it runs the algorithm for GAP without explicitly constructing the graph. Whenever a query "Is $(x, y) \in E$?" is made, it answers it by checking if configuration $y$ can be reached from $x$ in one step. Space constructibility of $S(n)$ is required as we need to compute the number of nodes in $G$ (for which we should know the value of $S(n)$). ♣

We next show that GAP is NL -complete (under log space reductions).

Definition 3.5 *A language $B$ is* NL*-complete if $B \in$* NL*, and for any language $A \in$* NL*, there is a deterministic Turing machine $M$ that uses $O(\log n)$ space such that, for any $x$, $x \in A \iff M(x) \in B$.*

Here we have tacitly extended the model of space bounded TMs by allowing it to have a one-way write only output tape, which, like its one-way read only input tape, does not count toward the space bound. (Of course it is easy to show that, if it has space bound $S(n)$, the output length is bounded by $2^{O(S(n))}$, or else the computation will loop and not terminate.)

**Exercise:** Show that logspace reductions are closed under composition.

Theorem 3.6 GAP *is* NL *-Complete.*

Since GAP is in P, and clearly P is closed under logspace reductions, a corollary is

Corollary 3.7 NL $\subseteq$ P.

*Proof.* It is easy to see that GAP $\in$ NL, as we simply guess a path (edge by edge and verify every step as we go) of length up to $n$.

Let $A \in$ NL via a machine $M$. We give a machine $M'$ that reduces $A$ to GAP . Given an input $x$, $M'$ simply outputs the configuration graph of $M$ on $x$. Clearly $M'$ can be implemented in deterministic logspace. ♣

The spirit of this Theorem is the following: To discuss non-deterministic space bounded computation, instead of the abstract formalism of automata theory, without loss of generality, one can discuss the concrete problem GAP. NL is nothing but GAP.

## 3.3 Immerman-Szelepcsényi Theorem

In this section we prove the Immerman-Szelepcsényi theorem: NL is closed under complementation. Since GAP is NL-*Complete*, it is enough if we show that $\overline{\text{GAP}}$ is in NL .

Given a graph $G$ and vertices $s$ and $t$, we want to devise a non-deterministic algorithm in Logspace, such that there are some computational paths that end in "yes" iff the graph has no path from $s$ to $t$ (we call such a path an $s$-$t$ path). This is a somewhat "unnatural" way of thinking about GAP . Usually, we phrase algorithms such that non-deterministic branches seek affirmative answers. In this case, however, if some branch says yes, then it has incontrovertible evidence that there is NOT an $s$-$t$ path; and moreover if there is NOT an $s$-$t$ path then some computational path should find such evidence.

While this "twist" is surprising, it is not contradictory, logically speaking. For example, one might try to find a "cut" in the graph separating $s$ and $t$. Note that such a cut exists iff there are no paths from $s$ to $t$. However, we can't simply look for a cut, because (presumably) representing the cut would consume too much space.

Let $G$ be a graph with $n$ vertices, among which are two distinguished vertices $s$ and $t$. The algorithm runs in two phases. In the first phase, we count the number of vertices reachable from the start vertex $s$. Define $c_i$ to be the number of vertices reachable from $s$, within $i$ steps. We incrementally compute $c_0, c_1, \ldots, c_{n-1}$. They are computed non-deterministically. From $c_i$ to $c_{i+1}$, we will assume the correct $c_i$ has been computed. Still, with $c_i$ written on its tape, only some computational paths will find out the correct $c_{i+1}$ and proceed. Consequently, only some computational paths will find out the correct $c_{n-1}$ and proceed to the next phase. The unsuccessful paths would know that they are unsuccessful and die. In the second phase the value of $c_{n-1}$ is used to check if $t$ is not reachable from $s$. In the following algorithm, think of any surviving path as saying, "the fact I have not died means I have evidence". The pseudocode is given in Algorithm 1. The ideas used in the algorithm are described next.

We'll incrementally build up counts for $c_0, c_1, \ldots, c_{n-1}$. Finding $c_0$ is easy: only $s$ is reachable from $s$ in zero steps, and so $c_0 = 1$. $c_1$ will be the number of vertices for which there is a directed edge from $s$, to $s$ itself. Suppose we have computed $c_i$ correctly and stored it (only $O(\log n)$ bits are needed to store $c_i$). We describe how to compute $c_{i+1}$ from $c_i$.

We undertake the following process for each $v$, in succession. Our "goal" for each $v$ is to see if we can reach it in at most $i+1$ steps, and if so we should increment $c_{i+1}$ for it. The way we want to find this out is to "re-experience" the entire set of vertices counted in $c_i$, and for each such vertex $u$ (which by definition are reachable from $s$ in at most $i$ steps), we check if $(u, v)$ is an edge in $G$, or $u = v$.

Of course any computational path which correctly computed $c_i$ has in the past encountered all these $c_i$ vertices. However we don't have space to record all the names of these $c_i$ vertices. Thus their identities are necessarily forgotten. So we go through all vertices, from 1 to $n$, and try to "re-experience" this list of vertices reachable within $i$ steps.

So for any fixed $v$, we go through the following:

- For each $u$, we non-deterministically guess whether it was one of the $c_i$ vertices reachable from $s$ in $\leq i$ steps. For any $u$, if we guess "yes", then we try to validate our guess. We do this by non-deterministically guess a path of length at most $i$ from $s$ to $u$. If the validation succeeds, $u$ is reachable from $s$ in $\leq i$ steps. Here's the moment of truth from this $u$: if there is an edge from $u$ to $v$ in $G$, or $u = v$, we increment $c_{i+1}$. If the validation fails, i.e., this particular guessed path did not reach $u$, then our guess was wrong *somewhere*. It could be wrong either because $u$ is not one of the $c_i$ vertices reachable from $s$ in $i$ steps, or our guessed path is not a valid path (even though some such path might exist). In any case some guess along this path is wrong, so halt and die! If we guess that $u$ is not reachable from $s$ in $i$ steps, do nothing and move on to the next $u$. In this whole process, we keep a count $d$ of the number of vertices $u$ for which we guessed "yes".

- Finally, after processing all $u$, check if $d = c_i$. (This is the "quality check": after all one can guess a lot of "no" and avoid all the validation and survive.) If $d = c_i$, we have accounted for all the $c_i$ number of vertices that are reachable from $s$ in $i$ steps, since we have validated all the "yes" guesses. We conclude that all the "no" guesses were correct as well. Therefore, if a computational path lived thus far, and got $d = c_i$, then indeed we have "re-experienced" all the vertices reachable from $s$ in $i$ steps. It follows that we have incremented $c_{i+1}$ for $v$ along this computational path iff $v$ is among the $c_{i+1}$ vertices from $s$ in $\leq i+1$ steps. On the other hand, any path with $d \neq c_i$ simply halts and dies!

**Algorithm 1:** Shows $\overline{\text{GAP}} \in \text{NL}$
**Input:** $\langle G, s, t \rangle$: where, $G = (V, E)$ is a directed graph over $n$ vertices, $s, t \in V$
**Output:** *accept* $\Longleftrightarrow$ there is NO path from $s$ to $t$ in $G$

```
Immerman-Szelepcsényi$(G, s, t)$
(1)   $c_0 = 1$
(2)   for $i = 0$ to $n-2$
(3)      $c_{i+1} = 0$
(4)      foreach node $v \in V$
(5)         $d = 0$
(6)         foreach node $u \in V$
(7)            Guess yes or no: if no, skip the next steps (8) to (12):
(8)               Guess a path of length $i$ from $s$ and if $u$ is not among the
                  vertices on this path, then reject
(9)               $d = d + 1$
(10)              if $[(u, v) \in E] \vee [u = v]$
(11)                 increment $c_{i+1}$
(12)                 goto 6 with next $v$
(13)        if $d \neq c_i$ then reject
(14)  $d = 0$
(15)  foreach node $u \in V$
(16)     Guess yes or no: if no, skip the next steps (17) to (19):
(17)        Guess a path of length $n-1$ from $s$ , and if $u$ is not among the
            vertices on this path, then reject
(18)        if $u = t$ then reject
(19)        $d = d + 1$
(20)  if $d \neq c_{n-1}$ then reject
(21)               else accept
```

Note: a lot of non-deterministic paths will "die" in this computation! Guessing a $u$ improperly for candidacy, for any $v$, will result in death as will guessing a $u$ correctly, but then following the wrong path. However, some sequence of guesses will run this gauntlet correctly all the way, and we'll have a proper count for $c_{n-1}$.

Now that we've computed $c_{n-1}$, we'll use it to answer the $\overline{\text{GAP}}$ question. The idea now is the same: "re-experience" all $c_{n-1}$ vertices and see if $t$ is among them. Thus, we iterate through all the vertices and (non-deterministically) try to account for the $c_{n-1}$ vertices reachable from $s$ once again.

For each node $v \in V$, we do the following: We guess whether there's a path of length $\leq n-1$ from $s$ to $v$. If the guess is "no", move on to the next vertex. If the guess is "yes", we verify it by guessing a path of length $\leq n-1$ and check if $v$ is encountered along the path. If the verification fails, die! If it succeeds, increment a counter $d'$. At the "moment of truth" when a vertex $v$ was verified to be among the $c_{n-1}$ vertices reachable from $s$, we check if it is $t$.

Finally, we accept $\langle G, s, t\rangle$ iff in the end $d' = c_{n-1}$ and $t$ was never marked as one among the $c_{n-1}$ vertices.

Note that, in the pseudocode above we have made some slight modifications. In line (12) after we confirmed that $v$ should be counted in $c_{i+1}$ we proceed to the next $v'$ immediately. Also in line (18) we reject once we confirmed that $t$ IS reachable. A subtle point: In the pseudocode it is no longer the case that a path with some incorrect guesses must die. Conceptually it is easier to think along the description above: All the guesses will be verified eventually one way or another, and any computational path that made an incorrect guess will eventually die. The whole process appears to be merely to re-experience again and again the set of vertices reachable within some number of steps, and we do the important task of computing the various counts $c_i$ and whether $t$ was among $c_{n-1}$ almost as an after thought. ♣

## 3.4 Polynomial Space

We devote this section for a discussion of polynomial space bounded computations. Let us start with a comparison of deterministic and non-deterministic polynomial space bounded computations.

Recall that PSPACE consists of all languages computable by deterministic Turing machines that run in polynomial space. One can consider the analogous non-deterministic class, namely NPSPACE, which consists of all languages computable by non-deterministic Turing machines that run in polynomial space.

Unlike the realm of time bounded computations, where the question of P vs. NP is open, in the realm of space bounded computations, the analogous question of PSPACE vs. NPSPACE is easy to resolve. The following theorem is a corollary to Savitch's theorem.

Theorem 3.8 *PSPACE = NPSPACE.*

Due to the above theorem, we will hardly talk about the notion of non-deterministic polynomial space bounded machines.

Our aim for the rest of the section is two-fold. We first exhibit a complete language for PSPACE. Then, we compare alternating Turing machines and PSPACE Turing machines.

### 3.4.1 QBF is PSPACE-Complete

In Section 2.2, we presented a canonical complete language (called $A_{\text{PSPACE}}$ ) for PSPACE. Here, we exhibit a "natural" PSPACE-complete language.

Definition 3.9 *Let $f$ be a (quantifier free) boolean formula over $n$ boolean variables $x_1, x_2, \ldots, x_n$. A quantified boolean formula $F$ derived from $f$ is given by,*

$$F = \exists x_1 \forall x_2, \ldots, Q x_n[f(x_1, x_2, \ldots, x_n)].$$

*Here, $Q = \exists$ if $n$ is odd and $Q = \forall$, if $n$ is even. We say that $F$ is* valid*, if the above statement is logically true. Meaning, "there exists some choice of $x_1$ (namely, $x_1 = 1$ or $x_1 = 0$) such that, for all choices $x_2$, there exists some choice of $x_3$ such that, so on ..., such that, the formula $f$ evaluates to true under the truth assignment $(x_1, x_2, \ldots, x_n)$.*

Definition 3.10 (QBF)

$$\text{QBF} = \{F | F \textit{ is a valid quantified boolean formula}\}$$

*Example.* Consider the quantified boolean formula

$$F = \exists x \forall y \exists z[(x \vee \neg y) \wedge (\neg y \vee z) \wedge (\neg x \vee y \vee \neg z)]$$

F is a valid formula. To see that, let $x = 1$, and for $y = 0$ we can set $z = 0$ and for for $y = 1$ we can set $z = 1$. Hence, $F \in$ QBF.

Note that even though we defined QBF to have strictly alternating quantifiers starting with an existential quantifier, these requirements are not essential. We can always append dummy variables to conform to this without increasing the size of the expression too much (at most double the size.)

Our goal is to show that QBF is PSPACE-Complete. As the first step, we prove that QBF is in PSPACE. We present a recursive procedure that solves QBF.

SolveQBF($F$)
Execute one of the following cases.

1. Case 1 ($F$ is a formula over zero variables):

- If $F$ is the formula "1" then return($F$ is valid).
- If $F$ is the formula "0" then return($F$ is not valid).

2. Case 2 ($F$ starts with an existential quantifier): Let $F$ start with "$\exists x....$". Define two quantified boolean formulas $F_1$ and $F_2$ by substituting $x = 0$ and $x = 1$ in $F$, respectively. Declare $F$ to be valid if at least one of $F_1$ or $F_2$ is valid. We can check this by making two recursive calls SolveQBF($F_1$) and SolveQBF($F_2$).

3. Case 3 ($F$ starts with an universal quantifier): Let $F$ start with "$\forall x....$". Define two quantified boolean formulas $F_1$ and $F_2$ by substituting $x = 0$ and $x = 1$ in $F$, respectively. Declare $F$ to be valid if both $F_1$ and $F_2$ are valid. We can check this by making two recursive calls SolveQBF($F_1$) and SolveQBF($F_2$) (after SolveQBF($F_1$) use one bit the remember the result, and then reuse the space in the second call of SolveQBF).

Notice that the formulas $F_1$ and $F_2$ have one less variable than $F$. Thus, if we start with a formula over $n$ variables, the depth of the recursive tree would be $n$. So the total space usage is polynomial. In short, we can simply do a tree-travesal of this binary tree of depth $n$ defined by $F$ in polynomial space (actually linear space). We have proved the following theorem.

Theorem 3.11 QBF*is in* PSPACE.

Our procedure for QBF is recursive. We often use recursion while writing space efficient algorithms. We saw another example of recursive procedure while proving Savitch's theorem. In general, the amount of space used by a recursive procedure would be $O(d \cdot s)$, where $d$ is the depth of recursion levels and $s$ is the local space usage per recursive level. It should be clear that one can unravel the recursion easily, by using stacks, and obtain a non-recursive procedure using the same amount of space $O(d \cdot s)$. But, it is often easier to write these procedures recursively.

We next show that QBF is PSPACE-hard. Let $M$ be a machine with space bound $p(n)$ and $x$ be an input. The computation of $M$ on $x$ is captured by the configuration graph $G$ of size $2^{O(p(n))}$. Let $c_0$ and $c_{acc}$ be the initial and accepting configurations of $M$. (We may wolog assume there is a unique accepting configurations of $M$.) Then, we want to check if there is a path of length at most $2^{O(p(n))}$ from $c_0$ to $c_{acc}$. We want to express the above question as a quantified boolean formula.

Let us do this in general among arbitrary two configurations. Let $c_1 \vdash_{2^k} c_2$ denote that there is a path of length at most $2^k$ from configuration $c_1$ to configuration $c_2$. We can use ideas from Savitch's theorem to express the above criterion as a quantified boolean formula. Recall that such a path exists, if and only if there is a configuration $c'$ such that, there are paths of length at most $2^{k-1}$ from $c_1$ to $c'$ and $c'$ to $c_2$:

$$c_1 \vdash_{2^k} c_2 \iff \exists c'[(c_1 \vdash_{2^{k-1}} c') \wedge (c' \vdash_{2^{k-1}} c_2)]$$

We can recursively expand the expression in square brackets. But, such a method would end up in a formula of exponential size (because, the expression in square brackets involves two recursive calls). So, we play a small trick. We can express the criterion as follows. The criterion

$$(c_1 \vdash_{2^{k-1}} c') \wedge (c' \vdash_{2^{k-1}} c_2)$$

is equivalent to

$$\forall(\text{ configurations } u \text{ and } v)[(\ (u = c_1 \text{ and } v = c') \vee (u = c' \text{ and } v = c_2)\ ) \Longrightarrow (u \vdash_{2^{k-1}} v)].$$

The advantage of this transformation is that the later expression used only one syntactic expression of $u \vdash_{2^{k-1}} v$. We can expand it recursively. At the base case $c_1 \vdash_1 c_2$ for $k = 0$, it can be expressed in a quantifier free boolean expression as in Cook's Theorem. Thus for $c_1 \vdash_{2^{O(p(n))}} c_2$ we obtain in polynomial time a quantified boolean formula $F$ such that the machine accepts $x$ if and only if $F$ is valid.

As configuration graph has $2^{O(p(n))}$ vertices, the final formula is of length $O(p(n))$. It uses only polynomial number of quantifiers. Technically, the final formula is not quite in the form required by our definition of quantified boolean formulas (Definition 2.24). But, $F$ can be converted to the required form easily by using standard logical equalities. We have proved that following theorem.

THEOREM 3.12 *QBF is PSPACE-Complete.*

Theorem 3.11 gives us an easy mechanism to compare PH and PSPACE. Recall that, for any $k \geq 1$, QBF$_k$ is $\Sigma_k^p$-Complete (Theorem 2.25). It is easy to see that QBF (with unbounded number of alternations) is a generalization of QBF$_k$ (which can have only some constant number $k$ alternations). Hence, as a corollary to Theorem 3.11, we have the following theorem (which is quite obvious at any rate when you think in terms of a tree traversal for a generic language in $\Sigma_k^p$ defined by quantifiers, or by a $k$-alternating TM.)

THEOREM 3.13 *For any $k > 0$, $\Sigma_k^p \subseteq$ PSPACE. Thus, PH $\subseteq$ PSPACE.*

We can now extend Figure 2.1 to Figure 3.1.

### 3.4.2 APTIME = PSPACE

Our next goal is to connect PSPACE machines and alternating Turing machines (see Section 2.7). We already saw that alternating Turing machines with constant number of alternations accept precisely those languages in the various levels of the polynomial hierarchy (Theorem 2.26). Here we shall consider polynomial time ATMs with unbounded number of alternations.

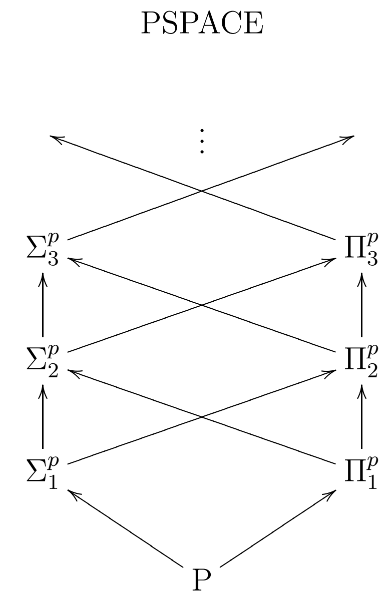


Figure 3.1: Polynomial Time Hierarchy and PSPACE

We say that a language is in the class APTIME, if it is accepted by an alternating Turing machine running in polynomial time. These machines are allowed to use arbitrary number of alternations. But, of course, as the machine runs in polynomial time, it can make at most a polynomial (in input length) number of alternations. We proceed to prove that alternating polynomial time and deterministic polynomial space are equal in power. In other words, we prove that APTIME=PSPACE.

First, let us show that PSPACE $\subseteq$ APTIME. One can give a proof using ideas from Savitch's theorem, similar to that of proving the PSPACE-hardness of QBF. But, as we have already shown that QBF is PSPACE-hard (Theorem 3.12), we will use it to prove the claim. Given a PSPACE machine $M$ and an input $x$, our APTIME machine first deterministically computes a quantified boolean formula $F$ using the reduction given by Theorem 3.12. So, the machine $M$ accepts $x$ if and only if $F$ is valid. Validity of $F$ can easily be tested in APTIME. Without loss of generality, assume that $F$ is in the standard form given by Definition 3.9. Let $F$ be over $n$ boolean variables. The machine will start in existential mode and choose a value for $x_1$ (either 0 or 1). Then, it will enter universal mode and choose a value for $x_2$. In this way, we continue by alternating between existential and universal modes and choose values for $x_1, x_2, \ldots, x_n$. Finally, each computational path will have a full truth assignment in hand. The path will evaluate the underlying boolean formula on this truth assignment. If the evaluation returns "true" then this path will *accept*, else *reject*. From the definition of alternating Turing machines, it is clear that the machine accepts $F$ if and only if $F$ is valid. (One can think of a computation graph of our APTIME machine as a complete binary tree of depth $n$, where the $k^{th}$ level will have $2^k$ nodes labeled $x_k$. Nodes in odd level will be existential and those in even levels will be universal. Each of the $2^n$ leaves will correspond to a full truth assignment and the leaf would evaluate the underlying boolean formula on that truth assignment.) We have proved the following claim.

Theorem 3.14 PSPACE ⊆ APTIME.

We next show that APTIME ⊆ PSPACE. Essentially, we consider the configuration graph of the APTIME machine and do depth first search to find out if its starting configuration is labeled "accept". Our PSPACE procedure is similar to the recursive procedure used to show that QBF is in PSPACE (Theorem 3.11).

We now give a bit more details. Let $M$ be an APTIME machine and $x$ be an input. Each configuration can be stored in polynomial space. We write a procedure that takes as input a configuration $c$ and outputs whether it is an accepting configuration. We use recursion and simply follow the definition of alternating Turing machines. It is easy when $c$ is a leaf configuration. We simply check if $c$ is an accepting state (as specified in the description of $M$). If so, $c$ is an accepting configuration, else it is a rejecting configuration. Suppose the input $c$ is a non-leaf configuration. Consider each of its children $c'$, one by one. Recursively, find out $c'$ is accepting or rejecting. If $c$ is an existential configuration, then $c$ is accepting if and only if at least one of its children is accepting. Similarly, if $c$ is a universal configuration, then $c$ is accepting if and only if all its children are accepting configurations.

Using the above procedure, we can find out if the starting configuration of $M$ is accepting or rejecting. As $M$ runs in polynomial time our procedure will need only polynomial number of levels of recursion. Each level needs only polynomial amount of space. Thus, the entire procedure runs in polynomial space. We have proved the following claim.

Theorem 3.15 PSPACE ⊆ APTIME

Combining Theorem 3.14 and Theorem 3.15, we have the following conclusion.

Theorem 3.16 APTIME = PSPACE

# Chapter 4

# Non-uniform Complexity

**Chapter outline:** Polynomial circuits, P/poly, Sparse sets. Equivalence among the three. Karp-Lipton Theorem. Mahaney's Theorem.

## 4.1 Polynomial circuits, P/poly, Sparse sets

As discussed in Lecture 1, Computational Complexity Theory has its genesis in Computability Theory. This theory is based on Turing's notion of a universal computing machine which is specified by a strictly finite number of rules. A major accomplishment of this theory is to demonstrate the existence of undecidable problems. While this was important both mathematically and philosophically, once we absorbed the notion of a strictly finitary device attempting to compute infinitely many instances of a problem as formulated by Turing Machines, the fact that there are some computational problems for which no TM can compute correctly for all instances becomes not so surprising.

If we carry out a "delayed" diagonalization proof, one can easily produce undecidable problems much like the Halting Problem, which nevertheless have very simple combinatorial structures. For example, we can have undecidable problems for which there are at most one string per length $n$. We can even make it much much sparser; all we need to do is to reject all strings at length $n$ except for the string $1^n$, and only for a very sparsely distributed set of values of $n$'s (say, $n = 2^{2^k}$), we carry out a "delayed" diagonalization (say, against the machine $M_k$). Combinatorially, this set is rather simple: we can, for example, represent this set at length $n$ by $x_1 \wedge \overline{x_1}$ (when it is the empty set $\emptyset$) or $x_1 \wedge x_2 \wedge \ldots \wedge x_n$ (when it consists of just $1^n$). Of course which case it is for each $n$ is undecidable computationally.

However, by a simple counting argument, most of $2^{2^n}$ Boolean functions from $\{0,1\}^n$ to $\{0,1\}$ do not have such simple representations. So it appears that there is a differently kind of complexity which is not accounted for by the consideration of computability in the TM model, a tiny box attempting to decide for all possible length input. Moreover, in today's complexity community, most researchers believe that the crux of the matter as to why such problems as NP-complete problems seem to resist any polynomial time solution is not due to the lack of a single strictly finitary device attempting to compute infinitely many instances, *but rather* simply it is too "complicated" at a large length $n$. This notion of "complicatedness" refers to a finite number ($2^n$) of instances for any fixed large $n$. The intuitive feeling is that for these apparently hard problems such as NP-complete problems, for large $n$, the number of logical operations required to solve all instances of length $n$ simply grows faster than any fixed polynomial in $n$, perhaps more likely an exponential function in $n$. Of course these are still conjectures with no proofs.

To formulate this intuition, one needs to define a different notion of "combinatorial" complexity, one which focuses attention on a finite length $n$, and ignores the issue of whether one can "string" them all together by a "uniform" finitary device such as a TM. This leads to the notion of non-uniform complexity.

We consider Boolean circuits. A Boolean circuit on $n$ input bits is a directed acyclic graph (DAG) where each node is labeled by either an input variable $x_i$ or a logical gate. We may, without loss of generality, assume these gates are $\wedge, \vee, \neg$ as they already form a complete logical basis. (Actually $\wedge, \neg$, or $\vee, \neg$ already do, but usually we include all three.) The default assumption is that nodes labeled by either $\wedge$ or $\vee$ have indegree 2, and nodes labeled by $\neg$ have indegree 1. Sometimes we allow $\wedge, \vee$ to have arbitrary indegrees, in which case we say the circuit has unbounded fan-in. A Boolean circuit may have one or more node labelled as output gate(s). The default is a single output node. Then a Boolean circuit computes a Boolean function in the obvious sense. The size of a Boolean circuit is the number of edges (wires) in the graph. If the outdegree is 1 for every node other than the input and output node (a.k.a. an internal node) then the Boolean circuit is called a Formula. In this case the Boolean circuit is essentially a tree, except at the input level. By De Morgan's Law, one can push the $\neg$ gates to the input level of the Formula, without increasing its size (or at most by $+n$ if not all negated variables are present). In a circuit or a formula, when all negation signs appear directly on the variables, sometimes the edges connecting $\neg$ and its respective variables are not counted in size, i.e., we couunt size in terms of inputs in the literals $x_1, x_2, \ldots, x_n \overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}$. This process of pushing $\neg$ gates to the input level can be carried out for Boolean circuits in general, with at most an increase of size by a factor of 2. (Replicate each node $v$ with a copy for $\overline{v}$.) The depth of a Boolean circuit is the longest path from input nodes to output(s).

A Boolean function $f_n : \{0,1\}^n \to \{0,1\}$ is always computable by a Boolean circuit as well as a Boolean formula; A Disjunctive Normal Form (DNF) or a Conjunctive Normal Form (CNF) will do. These are of depth 2 (in constant depth circuits, we do not count for depth the wires connecting the $\neg$ gates to the input level variables, i.e., we count the depth in terms of inputs by literals.) However in terms of size, it is easy to show that many Boolean functions have DNF and CNF size exponetial in $n$.

**Exercise:** The Parity function on $n$ bits

$$\oplus : \{0,1\}^n \to \{0,1\}$$

where $\oplus(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} x_i \bmod 2$ requires DNF and CNF size $2^{n-1}$.

The circuit size $C(f_n)$ of a Boolean function $f_n$ is the size of the smallest Boolean circuit which computes $f_n$. Similarly one can define formula size of $f_n$ (and such definitions can be carried out for a variety of other models of computation: e.g., for any depth $k$, we can consider the depth $k$ size of $f_n$, or if we consider only monotone circuit in which $\neg$ gates does not appear, we can define the monotone circuit size for any monotone function $f_n$.)

The class of problems we will focus on consists of Boolean function families $\{f_n\}_{n=1,2,\ldots}$ with $C(f_n) = n^{O(1)}$, i.e., function families which can be computed by circuit families with size bounded by a polynomial in $n$. A language $L$ is said to have polynomial size circuits, if its family of characteristic functions at length $n$ $\{L^{=n}\}$ does. A complexity class is said to have polynomial size circuits, if every language in the class does.

Theorem 4.1 *P has polynomial size circuits.*

*Proof.* The proof is simple. Take any TM with time bound $T(n)$, and input $x$ of length $n$. Then one can write down a square array of $T(n) \times T(n)$ cells, where each cell $(i, j)$ refers to the computation of $M(x)$ at time step $i$ and memory location $j$. It takes $O(1)$ gates to describe the following: the state $M$ is in at time $i$, whether the tapehead is at location $j$ at time $i$, and what content is in tape square $j$ at time $i$. For $i = 1$ this is determined by the initial configuration. For $i > 1$, each cell $(i, j)$ is determined by cells $(i-1, j-1)$, $(i-1, j)$ and $(i-1, j+1)$. For example if the tapehead is not at these three locations at time $i-1$, then the cell at $(i, j)$ should remain the same as $(i-1, j)$; if the tapehead is among the three locations at time $i-1$, then the cell at $(i, j)$ should be decided according to the finitary transition function of $M$. In any case locally it is computed by a Boolean function of size $O(1)$. Thus the circuit which simulates the whole computation of $M(x)$ has size at most $O(T^2(n))$.

Note also that this circuit can be constructed by a logspace computation. ♣

A corollary of this theorem is that the Circuit Value Problem (CVP) is P-complete under logspace reductions.

**Circuit Value Problem:** Given a Boolean circuit $C_n$ on $n$ inputs and an input of $n$ bits $(b_1, b_2, \ldots, b_n)$, is $C_n(b_1, b_2, \ldots, b_n) = 1$?

Corollary 4.2 *CVP is P-complete.*

The robustness of a complexity measure is demonstrated by its invariance under small changes. We noted that, e.g., the class P is unchanged whether we use multitape or onetape TMs, or a variety of other models. This notion of having polynomial size circuits is also quite robust. It does not depend on the particular logical basis functions $\wedge, \vee, \neg$ we chose. Nor does it change if we were to count size in terms of nodes, or allow unbounded fan-in, or push all negations $\neg$ to the bottom level, etc. Moreover, there are several equivalent definitions for this class.

An advice function is $l : \mathbf{N} \to \Sigma^*$. Thus, for every length $n$, the function gives a string $l(n)$. The advice function is polynomially bounded if the length $|l(n)| = n^{O(1)}$.

DEFINITION 4.3 *Let $\langle \cdot, \cdot \rangle$ be a standard pairing function, let $\mathcal{C}$ be a complexity class, then $L \in \mathcal{C}$/poly if there exists some set $L' \in \mathcal{C}$ and a polynomially bounded advice function $l$, such that for all $x \in \Sigma^{=n}$,*

$$x \in L \Leftrightarrow \langle x, l(|x|) \rangle \in L'.$$

It is important to note that the "advice" $l(|x|)$ only depends on the length of $x$, but not $x$ itself. Thus, for $L \in$ P/poly the advice function $l$ can only give polynomially many bits of advice to the P computation of $L'$ for all $2^n$ inputs $x \in \Sigma^{=n}$.

THEOREM 4.4 $L \in$ P/poly *iff $L$ has polynomial size circuits.*

*Proof.* If $L \in$ P/poly we can adapt the proof above for Theorem 4.1 to show that $L$ has polynomial size circuits.

If $L$ has polynomial size circuits, then we can define an advice function $l$, such that $l(n)$ is a binary description of the circuit at length $n$, and the P language $L'$ on input $\langle x, s \rangle$ simply evaluates the circuit encoded in $s$ on $x$. ♣

This equivalence of P/poly and having polynomial size circuits is so well entrenched that people frequently speak of one term and literally mean the other. Often P/poly is a shart hand for having polynomial size circuits.

Another well-known equivalent condition of P/poly refers to sparse sets.

DEFINITION 4.5 *A set $S \subset \Sigma^*$ is called a sparse set if $|S^{=n}| = n^{O(1)}$, i.e., there are at most a polynomial number of strings at length $n$ in $S$.*

Clearly $S$ is sparse iff $|S^{\leq n}| = n^{O(1)}$, since $|S^{=n}| \leq |S^{\leq n}| \leq (n+1) \max_{0 \leq m \leq n} |S^{=m}|$. Sparse sets are considered as sets with low information content. Note that generally a set at length $n$ may have up to $2^n$ strings.

THEOREM 4.6 $L \in \mathrm{P}^S$ *for some sparse set $S$ iff $L$ has polynomial size circuits.*

*Proof.* Suppose $L \in \mathrm{P}^S$ via a P-time oracle TM $M$ and a sparse set $S$. On length $n$ inputs, $M$ can only query strings with length up to some polynomial of $n$ in the oracle $S$. Up to that length there are only polynomially many strings in $S$. If we explicitly listed all these strings, then we can adapt the proof that P has polynomial size circuits to show that $L$ also has polynomial size circuits.

Conversely, If $L \in \mathrm{P}/poly$ via advice function $l$, we can encode the advice string $l(n)$ in a sparse oracle. E.g.,

$$S = \{\langle 1^n, p_n\rangle | p_n \text{ is a prefix of } l(n), n \geq 0\}.$$

Given $S$, we can "suck" out the bits of $l(n)$ as follows: By querying $\langle 1^n, 0\rangle$ and $\langle 1^n, 1\rangle$ one can find the first bit of $l(n)$. Having found a prefix $p$ of $l(n)$, we query $\langle 1^n, p0\rangle$ and $\langle 1^n, p1\rangle$, and we can extend the prefix $p$ by one more bit (or we found that $p = l(n)$ is already fully extended.) ♣

The class P/poly is the non-uniform analog of P. It contains the class P, as well as non-recursive sets. Since it also includes non-recursive sets, it is not possible to include the class P/poly in any uniform complexity classes, such as EXP. However, the intuitive feeling is that sets in P/poly are "easy", in a combinatiorial sense. The belief is that NP $\neq$ P because NP $\not\subset$ P/poly. This remains the most concrete approach to a proof of NP $\neq$ P.

However is it possible to include higher uniform complexity classes than P in the class P/poly? What if NP $\subset$ P/poly? An unlikely complexity theoretic consequence would provide evidence that indeed NP $\not\subset$ P/poly. This is the content of the famous Karp-Lipton Theorem.

## 4.2 Karp–Lipton Theorem

In this section, we prove the classical Karp-Lipton theorem. It deals with the question of whether NP $\subset$ P/poly? It is believed that NP $\not\subset$ P/poly. Karp-Lipton theorem gives some evidence in this direction. It says that if NP $\subset$ P/poly, then PH $= \Sigma_2^P \cap \Pi_2^P$.

By NP-completeness, NP $\subset$ P/poly is equivalent to SAT $\in$ P/poly. Also by the equivalent formulations of P/poly, this is equivalent to the assertion that SAT(or any other NP-complete language, or every NP-complete language, or every language in NP) has polynomial size circuits. It is also equivalent to the assertion that SAT is polynomial time Turing reducible to a sparse set $S$: SAT $\in \mathrm{P}^S$.

**Self Reducibility:** The main idea is self-reducibility of SAT. Suppose we are given a circuit $C$ that is suppopsed to compute SAT for all Boolean formulae of length at most $n$. It may or may not compute SAT correctly. It may make errors in two ways. It may reject a $\varphi \in$ SAT, and/or it may also accept a $\varphi \notin$ SAT. We can convert $C$ into a circuit $C'$, that does not make errors of the second type. $C'$ works by using $C$ as a black box. Upon input $\varphi$ over the variables, $x_1, \ldots, x_m$, it feeds $\varphi$ to $C$. If $C$ rejects, $C'$ rejects. If $C$ accepts, $C'$ tries to find a satisfying truth assignment $\sigma$ to $\varphi$. To do that, $C'$ converts $\varphi$ to $\varphi_1$, by setting all the occurrences of $x_1$ in $\varphi$ to be TRUE. And asks $C$ if $\varphi_1$ is satisfiable. If so set $\sigma(x_1)$=TRUE, else set $\sigma(x_1)$=FALSE. Now repeat this process, with $\varphi_1$ in place of $\varphi$ and $x_2$ in place of $x_1$, to get the value for $\sigma(x_2)$. Continue this process for the other variables we find a truth assignment $\sigma$. Then we check whether $\sigma$ really satisfies $\varphi$, if so accept, else reject $\varphi$. If $C$ is indeed a correct circuit for SAT, $C'$ would also be a correct one. But, no matter whether $C$ is a correct circuit for SAT, $C'$ may make errors only on one side. It may reject a satisfiable formula. But, it would never accept an unsatisfiable formula, because it only accepts when it has verified a satisfying truth assignment. This process of reducing an instance of a problem (in this case SAT) to another instance of the same problem of a smaller "size" (in this case the number of variables left unassigned) is called self-reducibility.

Theorem 4.7 (Karp-Lipton) *If* NP $\subset$ P/poly*, then* PH $= \Sigma_2^P \cap \Pi_2^P$.

**Remark:** Originally Karp and Lipton proved the collapse to the third level of the Polynomial Hierarchy. They noted in their paper that Sipser improved it to the second level of PH. The proof we give uses an idea first noted by Hopcroft.

*Proof.* The assumption implies that SAT has polynomial circuits. Then, we show that $\Pi_2^P \subseteq \Sigma_2^P$. This would imply that PH collapses to $\Sigma_2^P \cap \Pi_2^P$.

Let $L \in \Pi_2^P$. So, for some P-time predicate $D$, we have

$$x \in L \iff \forall^p y \exists^p z[D(x,y,z) = 0]$$

where $|y|$ and $|z|$ are polynomially bounded in $|x|$. We can convert the question of

$$\exists^p z[D(x,y,z) = 0]$$

into a SAT question via Cook's Theorem. Meaning, we have a polynomial time computable function $\varphi(.,.)$ such that

$$x \in L \iff \forall^p y[\varphi(x,y) \in \mathrm{SAT}]$$

Let $x$ be an input. We need to check whether $x \in L$. We simulate the $\Pi_2^P$ computation of $L$ as follows. Let the length of the formulas $|\varphi(x,y)|$ be at most $p(n)$, where $n = |x|$ and $p(.)$ is some polynomial. By our assumption, there is a (poly-sized) circuit $C^*$ that would work correctly for SAT for all formulas of length at most $p(n)$. Our $\Sigma_2^P$ computation, in its "there-exists" phase, guesses a circuit $C$. Then uses self-reducibility to convert it into a one-sided error circuit $C'$. Now it enters the "for all" phase and checks whether

$$\forall^p y[C'(\varphi(x,y)) = 1]$$

Suppose $x \in L$. One of the paths in the "there-exists" phase would guess $C^*$, a correct circuit for SAT. The corresponding $C'$ will also be a correct circuit for SAT. As $x \in L$, $\forall^p y$, $\varphi(x,y)$ will be satisfiable and our $C'$ would accept all of them for these polynomially bounded $y$'s. So we see that,

$$(\exists^p C)(\forall^p y)[C'(\varphi(x,y)) = 1]$$

Suppose $x \notin L$. Then, there is some $y_0$, polynomially bounded in length, for which $\varphi(x, y_0)$ is not satisfiable. So, for all the guesses $C$, the corresponding $C'$ would reject $\varphi(x, y_0)$, because $C'$ never accepts any unsatisfiable formula. Thus, we see that,

$$(\forall^p C)(\exists^p y)[C'(\varphi(x, y)) = 0]$$

More formally, define the following deterministic polynomial algorithm $A(x, C, y)$: it first computes $C'$ from $C$ using self-reducibility. Then, it computes the formula $\varphi(x, y)$ using Cook's theorem. Output $C'(\varphi(x, y))$. The claim is then

$$x \in L \iff (\exists^p C)(\forall^p y)[A(x, C, y) = 1]$$

♣

## 4.3 Mahaney's Theorem

In the last section, we discussed Karp–Lipton theorem, which says that if SAT has polynomial size circuits, then the polynomial hierarchy collapses to the second level. In other words, if SAT Cook reduces to some sparse set S, then the hierarchy collapses to the second level. A natural question at this point is, what happens if SAT Karp reduces to a sparse set. Notice that this is a stronger assumption. Mahaney's theorem says that if SAT Karp reduces to a sparse set, then NP=P. The theorem also has implications in the context of Berman-Hartmanis conjecture, which we will discuss later. In this section, we prove Mahaney's theorem. This proof follows the idea of Ogihara and Watanabe.

Definition 4.8 *Let $\sigma = \sigma_1\sigma_2\ldots\sigma_m$ and $\tau = \tau_1\tau_2\ldots\tau_k$ be binary strings. We say $\sigma$ is to the left of $\tau$ (denoted $\sigma \preceq_l \tau$) if either $\sigma$ is an extension of $\tau$, or the first bit from the left where they differ $\sigma$ has bit 0 and $\tau$ has bit 1. Formally, let $i = \max\{j \mid 0 \leq j \leq \min\{k, m\}, (\forall 1 \leq j' \leq j)[\sigma_{j'} = \tau_{j'}]\}$. Then $\sigma \preceq_l \tau$ if either $i = k$ or $i < k$ with $\sigma_{i+1} < \tau_{i+1}$.*

We can view the above definition pictorially as follows. Consider the binary tree of partial assignments to the $n$ variables. Let $\sigma$ and $\tau$ be two partial assignments, $\tau = \tau_1\tau_2\ldots\tau_k$ and $\sigma = \sigma_1\sigma_2\ldots\sigma_m$. $\sigma$ is to the *left* of $\tau$, if $\sigma$ is below $\tau$ (i.e. $\tau$ is a prefix of $\sigma$) or $\sigma$ is in a branch to the left of $\tau$. The order $\preceq_l$ is the tree traversal order of Left-Right-Root.

We next define a language called *left cut* of SAT. Let $\varphi$ be a formula and $\tau$ be a partial assignment. The pair $\langle\varphi, \tau\rangle$ is in $L_{\text{SAT}}$, if some assignment $\sigma$ to the left of $\tau$ satisfies $\varphi$. Formally,

Definition 4.9 *The leftcut set of* SAT *is the set* $\text{L}_{\text{SAT}} = \{\langle\varphi, \tau\rangle \mid \varphi$ *is a formula on* $n$ *variables and* $\tau \in \{0,1\}^k, 0 \leq k \leq n$, *such that* $\exists\ \sigma \in \{0,1\}^n$ *with* $\sigma \preceq_l \tau$ *and* $\varphi|_\sigma =$ *True.*$\}$

Clearly $L_{\mathrm{SAT}} \in \mathrm{NP}$. The crucial property of $L_{\mathrm{SAT}}$ is the following. Let $\varphi$ be a formula and $\sigma$ and $\tau$ be two partial assignments, such that $\sigma \preceq_l \tau$. Then, if $\langle \varphi, \sigma \rangle$ is in $L_{\mathrm{SAT}}$, then $\langle \varphi, \tau \rangle$ is also in $L_{\mathrm{SAT}}$. In particular if $\varphi \in \mathrm{SAT}$, then $\langle \varphi, \epsilon \rangle \in L_{\mathrm{SAT}}$.

Suppose we are given $\langle \varphi, \sigma \rangle$ and $\langle \varphi, \tau \rangle$ and we have to bet on one of these to be in $\mathrm{L_{SAT}}$. Clearly, we should place our bet on $\langle \varphi, \tau \rangle$, if $\sigma \preceq_l \tau$. In its contra-positive, if at least one of them is out of $\mathrm{L_{SAT}}$, then we know $\langle \varphi, \sigma \rangle$ is out. The proof hinges on this property of $\mathrm{L_{SAT}}$.

Theorem 4.10 (Mahaney) *For any sparse set $S \neq \emptyset$,* $\mathrm{SAT} \ \leq_m^p \ S \iff \mathrm{P} = \mathrm{NP}$.

*Proof.* Let $S$ be any non-empty sparse set. The implication $\mathrm{P} = \mathrm{NP} \Longrightarrow \mathrm{SAT} \ \leq_m^p \ S$ is trivial, so we need only prove that $\mathrm{SAT} \ \leq_m^p \ S \Longrightarrow \mathrm{P} = \mathrm{NP}$.

Suppose $\mathrm{SAT} \ \leq_m^p \ S$. We shall design a polynomial time algorithm to solve SAT. First of all, notice that $\mathrm{L_{SAT}}$ is in NP. (Given $\langle \varphi, \tau \rangle$, we guess a (full) truth assignment $\sigma$, then verify that $\sigma \preceq_l \tau$ and $\sigma$ satisfies $\varphi$; if so accept, else reject.) From the assumption $\mathrm{SAT} \ \leq_m^p \ S$, it follows that $\mathrm{L_{SAT}} \leq_m^p \ S$, via some polynomial time computable function $f$.

We will design a polynomial time algorithm for SAT. Let $\varphi$ be the input formula over $n$ variables and we need to check if it is satisfiable or not. If $\varphi$ is satisfiable, the algorithm would, in fact, output the (lexicographically) left-most satisfying assignment.

Let us consider the binary tree formed by assignments on $\varphi$. The root of this tree corresponds to the empty assignment (denoted by the empty string $\epsilon$, where no variable is assigned a value). Nodes of the tree correspond to partial assignments. We will do a breadth-first search on this tree, starting with the root. The tree has exponentially many nodes and we cannot hope to do a full search. Instead, as we go along, we will prune the tree and explore only parts of the tree. At any point of time, we will maintain only polynomially many nodes. We will ensure that, if $\varphi$ is satisfiable, the left-most satisfying assignment is a descendant of a node not pruned away.

Consider strings of the form $\langle \varphi, \sigma \rangle$, where $\sigma = \sigma_1 \sigma_2 \dots \sigma_m$ is some partial assignment for the input formula $\varphi$. Length of these strings is at most $|\varphi| + n$. As the reduction $f$ runs in polynomial time, its output on these strings can be at most polynomial in $n$. Let this number be $l$. The sparse set can have at most $N$ strings of length $\leq l$, where $N$ is polynomial in $l$, and thus also a polynomial in $n$. As we explore the binary tree, level by level, we will maintain at most $N$ nodes, at any point of time.

First, we explore the tree until we reach a level $k$, where the number of nodes $2^k > N$. At this point, we start pruning. We run the reduction $f$ on all these $2^k$ partial assignments (with $\varphi$ as the formula) and obtain $2^k$ output strings. We first look for duplicates. Let $\sigma_1$ and $\sigma_2$ be two partial assignments at level $k$ such that $f(\langle \varphi, \sigma_1 \rangle) = f(\langle \varphi, \sigma_2 \rangle)$. If $\sigma_1 \preceq_l \sigma_2$, then we throw away $\sigma_2$, i.e. we will not explore that subtree any further. If $\sigma_2 \preceq_l \sigma_1$, then we throw away $\sigma_1$. Note that as $f(\langle \varphi, \sigma_1 \rangle) = f(\langle \varphi, \sigma_2 \rangle)$, either both $(\varphi, \sigma_1)$ and $(\varphi, \sigma_1)$ are in $\mathrm{L_{SAT}}$ or both are out. After removing all duplicates, if more than $N$ nodes survive, we throw away the left-most partial assignment at this level. We keep removing the left-most nodes until we have only $N$ nodes. Clearly, after all is said and done at this level, we will have at most $N$ nodes that survive.

Then we continue at the next level, expanding only these subtrees. In general, suppose $r \leq N$ nodes survive at level $m$. In level $m+1$, we consider their $2r$ children. If $2r \leq N$, we can move on to level $m+2$. If not, we run the reduction $f$ on these $2r$ nodes, and obtain $2r$ output strings. We first eliminate duplicates and then, if more than $N$ nodes still remain, we eliminate the the left-most nodes, until we have at most $N$ nodes. Continuing this way, finally we will reach the leaf level of the tree. Here, all the nodes correspond to full truth assignments of $\varphi$. We will have at most $N$ surviving full assignments. For each of these assignments, we check if any of them satisfy $\varphi$. If so $\varphi$ is satisfiable. Otherwise, we claim it is not satisfiable.

Clearly, we maintain at most $N$ nodes at any level of the tree and $N$ is polynomial in $n$. Depth of the tree is $n$. Thus the algorithm runs in polynomial time. Suppose $\varphi$ is satisfiable. We argue that the left-most satisfying assignment will remain a descedant of some node kept at any level, and thus survive at the leaf-level. We do this by induction on the level number $m$. This is clearly so at the root level. Consider level $m$ and let the number of nodes surviving at this level be $r$. By induction, the left-most satisfying assignment is a descendant of one of the $r$ surviving nodes. We consider the $2r$ children at level $m+1$. If $2r > N$. Suppose $f(\langle\varphi, \sigma_1\rangle) = f(\langle\varphi, \sigma_2\rangle)$ and $\sigma_1 \preceq_l \sigma_2$. So we remove a duplicate. Let $\sigma_1 \preceq_l \sigma_2$. In this case, we throw away $\sigma_2$. Clearly, the left-most satisfying assignment is not a descendant of $\sigma_2$, for otherwise we would have the contradiction that $\langle\varphi, \sigma_2\rangle \in \mathrm{L}_{\mathrm{SAT}}$ and $\langle\varphi, \sigma_1\rangle \notin \mathrm{L}_{\mathrm{SAT}}$, as $\sigma_1$ is strictly to the left of $\sigma_2$. Thus, throwing away $\sigma_2$ did not violate the inductive hypothesis.

After removing duplicates, if still some $d > N$ nodes exist, let these nodes be $\sigma_1, \sigma_2, \ldots, \sigma_d$. Then we removed the left-most node $\sigma_1$. We argue that this is also a correct procedure. By contradiction, suppose the left-most satisfying assignment $\sigma$ is a descendant of $\sigma_1$. Then, $\sigma$ is to the left of all the $d$ surviving nodes. Thus, all of $\langle\varphi, \sigma_1\rangle, \langle\varphi, \sigma_2\rangle, \ldots \langle\varphi, \sigma_d\rangle$ are in $\mathrm{L}_{\mathrm{SAT}}$. So, $f(\langle\varphi, \sigma_1\rangle), f(\langle\varphi, \sigma_2\rangle), \ldots f(\langle\varphi, \sigma_d\rangle)$ are all in $S$. Note that, all these strings are distinct and there are $d > N$ of them. But, by our assumption of sparseness of $S$, there can be at most $N$ strings in $S$ (at this length). This contradiction shows that throwing away the left-most node also maintains the inductive hypothesis. This completes the induction proof. ♣

# Chapter 5

# Randomization

Randomness, both as a proof technique as well as a computational resource, has a significant role in the modern theory of algorithm and complexity. In this Chapter we start our study of randomization in computation.

## 5.1 Basic Probability

It is perhaps one of the primal experiences, much like the primal experiences with arithmetic quantity or geometric shape, that people have come to "know" randomness. Any systematic exploration was started much later, starting with Pascal, Fermat and Laplace. But it is always a nettlesome quesiton as to what exactly is "randomness", and what exactly is "probability". I don't think this question has ever been truly satisfactorily answered, despite a lot of work on this topic. Perhaps there is no one single answer. However, brushing aside the "nature" of what is "probability", modern mathematics basically took the following perspective, after Kolmogorov. We start with an arbitrary measure space $(\Omega, \mathbf{E}, \mu)$, where $\Omega$ is some underlying set, equipped with a measure function $\mu : \mathbf{E} \rightarrow \mathbf{R}_+$, where $\mathbf{E}$ is a collection of subsets of $\Omega$ called a $\sigma$-algebra satisfying certain closure properties, and (being a probability measure) $\mu(\Omega) = 1$. (Being a measure, $\mu$ must satisfy $\sigma$-additivity, namely for disjoint countable family $\{S_i\} \subseteq \mathbf{E}$, $\mu(\bigcup_{i=1}^{\infty} S_i) = \sum_{i=1}^{\infty} \mu(S_i)$. $\mathbf{E}$ is a $\sigma$-algebra if $\Omega \in \mathbf{E}$, and $\mathbf{E}$ is closed under countable union, intersection, and complement. We will not discuss further the properties of a $\sigma$-algebra. For most of what we do, the set $\Omega$ will be finite, and $\mathbf{E}$ consists of all subsets of $\Omega$.)

The Kolmogorov foundation of probability theory is a brilliant device to refocus the "theory of probability" as an internal mathematical subject, excluding all issues having to do with what is "randomness" as we experience it in the external world. It essentially gets rid of the issue of any reference to how this "probability theory" is to be related to the every day (or not so every day) notion of "chance" in the external world. A gambler (or any user of probability theory) has to decide what is an appropriate probability space to presume, as a mathematical model for his particular situation at hand. Ths modeling is external to any mathematical theory of probability. The mathematics of "probability theory" in the sense of Kolmogorov only helps to "calculate" the outcomes once the basic probability of events have been assigned. Thus, in Kolmogorov's probability theory, there is no meaning of a "fair coin"; instead, one merely sets up a suitable probability space, such as $\{H, T\}$, and postulate that $\mu(H) = \mu(T) = 1/2$. It is murkier what a quantum physicist really means when he talks about the "probability" of observing this or that quantum state.

What if we want "two independent fair coin flips"? In Kolmogorov's probability theory, we can set up a product probability space, $\{H, T\}^2$, and assign $\mu(HH) = \mu(HT) = \mu(TH) = \mu(TT) = 1/4$. Note that in this 4-point space, there is, strictly speaking, no notion of time.

While the Kolmogorov foundation is fine, it does seem to be somewhat sterile, and lacking certain intuition of "probabilistic thinking". For one thing, we do like to have a primitive notion of an independent coin flip. Moreover, if after several independent fair coin flips, we decided to have another one. This should not disturb any previous probability calculation. Instead in Kolmogorov's framework we must re-constitute a new probability space all over again. Technically all previous probability calculations must be carried out now in this new probability space. Of course they take the same values, but conceptually they now take place in a different probability space. This, I find unnatural and unsatisfactory. Frequently in an algorithm we want to be able to flip more coins as we go along. And this notion of time step is natural and intuitively helpful. It is somewhat unnatural to suppose we must have a super probability space in place a priori, which in its very definition has a built-in structure of how many coin flips we can have. (There are ways around this in Kolmogorov's framework, but they all seem contrived and unnatural.)

Luckily, for almost everything we discuss here, it will be over some finite (or at most countably infinite and recursive) space. Most of the time it will just be over some $\{0, 1\}^n$, or something similar. There will be no philosophical difficulties by taking a naive approach to the concept of probability, but one which does admit a primitive notion of a new independent bit. In principle we can enumerate all the basic events and assign them "atomic probabilities" that add to one. Therefore we will take the following point of view. We will assume whenever we need we can have an independent additional coin flip (which may not be a fair coin). We will operate at a more intuitive level of "probability", as if we had a definite meaning of a physical "randomness". Thus we can say, for example, perform the following random trials independently for certain number of times with certain probability. We take this approach with the understanding that, whenever any potential difficulty should arise, we immediately retreat back to the safe cocoon of Kolmogorov foundation of measure space, and effectively say that: the only meaningful thing is what's happening in the following measure space; any implication to the "outside world" (such as what exactly does it mean by a random step) is not the responsibility of our probability analysis and any conclusions thereof.

### 5.1.1 Markov's Inequality

The first inequality to consider is Markov's Inequality. It deals with any random variable that take only nonnegative values and the estimate is in terms of its expectation. Technically we need to assume its has a finite expectation, however, one can apply it with abandon, for when the nonnegative random variable has infinite expectation the estimate is trivially true.

THEOREM 5.1 *Let $X$ be a random variable such that $X \geq 0$ and $0 \leq \mathrm{E}\left[X\right] < \infty$. For all $a > 0$,*

$$\Pr[X \geq a] \quad \leq \quad \frac{\mathrm{E}\left[X\right]}{a}$$

The expectation $\mathrm{E}\left[X\right]$ is defined to be $\int X d\mu$. This theorem gives us a bound on the probability that $X$ takes on a value that is *greater* than its expected value by a given amount. We give a quick proof of the theorem:

*Proof.* Let $I$ be an indicator variable for the event $X \geq a$. That is,

$$I = \begin{cases} 1 & \text{if } X \geq a \\ 0 & \text{if } X < a \end{cases}$$

Clearly, $\mathrm{E}\left[I\right] = \Pr[X \geq a]$. Then,

$$\Pr[X \geq a] = \int I d\mu = \int_{X \geq a} 1 d\mu \leq \int_{X \geq a} \frac{X}{a} d\mu \leq \int \frac{X}{a} d\mu = \frac{\mathrm{E}\left[X\right]}{a}.$$

♣

### 5.1.2 Chebyshev Inequality

THEOREM 5.2 *For any random variable $X$ with finite $\mathrm{E}\left[X\right]$ and $\mathrm{Var}(X)$, and let $a > 0$,*

$$\Pr[|X - \mathrm{E}\left[X\right]| \geq a] \leq \frac{\mathrm{Var}(X)}{a^2}$$

Here the variance $\mathrm{Var}(X)$ is defined as $\mathrm{E}\left[(X - \mathrm{E}\left[X\right])^2\right] = \mathrm{E}\left[X^2\right] - \mathrm{E}\left[X\right]^2$, and technically we must assume it is finite. However, the above estimate is (trivially) true even if it is infinite.
*Proof.* Define a random variable $Y = (X - \mathrm{E}\left[X\right])^2$. Then,

$$\begin{aligned} \Pr[|X - \mathrm{E}\left[X\right]| \geq a] &= \Pr[Y \geq a^2] \\ &\leq \frac{\mathrm{E}\left[Y\right]}{a^2} \\ &= \frac{\mathrm{Var}(X)}{a^2} \end{aligned}$$

In the above derivation, the inequality is obtained using the Markov's inequality. ♣

### 5.1.3 Chernoff Bound

It may appear that Markov's inequality is pretty weak. However it is very applicable in many situations, mainly because it only assumes that the random variable under consideration to be non-negative and does not assume anything about its distribution. Judiciously applied it can yield remarkably sharp bounds. In this section, we use it to prove the Chernoff bound, which is a powerful inequality dealing with sums of independent random variables.

Before we give the statement of the Chernoff bound, we introduce the variables it will use. Let $\{X_1, X_2, \cdots, X_n\}$ be a set of $n$ independent, identically distributed random variables such that each $X_i$ has

$$\Pr[X_i = 1] = \Pr[X_i = -1] = \frac{1}{2}.$$

Let $S_n$ be their sum: $S_n = \sum_{i=1}^{n} X_i$.

THEOREM 5.3 *For all $n$, and for all $a > 0$,*

$$\Pr[S_n \geq a] \quad \leq \quad e^{-a^2/2n}$$

Before proving this theorem, we note that it can be restated as

$$\Pr[S_n \geq \epsilon \cdot n] \quad \leq \quad e^{-\frac{1}{2}\epsilon^2 n} = (e^{-\frac{1}{2}\epsilon^2})^n$$

or

$$\Pr[S_n \geq \alpha \cdot \sqrt{n}] \quad \leq \quad e^{-\frac{1}{2}\alpha^2}$$

If we think of $\epsilon$ and $\alpha$ as positive constants, then the first inequality says the probability of having an $\Theta(n)$ deviation from expectation (0) is exponentially small, and the second inequality says that this "tail probability" balances out at $\Theta(\sqrt{n})$ away from expectation. This last statement is in accord with the central limit theorem, which states that

$$\lim_{n\to\infty} [S_n \geq \alpha \cdot \sqrt{n}] = \int_{\alpha}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx.$$

The advantage of the Chernoff Bound is that it is valid for all $n$ and $a$, and not merely a statement of limit.

Now, we prove the theorem.

*Proof.* The trick is to consider the exponentiation of $S_n$, namely the random variable $e^{\lambda S_n}$. Here $\lambda > 0$ is some number to be fixed later. We compute the expectation of $e^{\lambda S_n}$ as follows.

$$\mathrm{E}\left[e^{\lambda S_n}\right] = \mathrm{E}\left[e^{\lambda \sum_{i=1}^{n} X_i}\right] = \mathrm{E}\left[\prod_{i=1}^{n} e^{\lambda X_i}\right] = \prod_{i=1}^{n} \mathrm{E}\left[e^{\lambda X_i}\right]$$

To get the last equality, we use the assumption that the $\{X_i\}$ are independent random variables and hence $\{e^{\lambda X_i}\}$ are also independent. (Recall that, for independent random variables $X$ and $Y$, the expectation is multiplicative $\mathrm{E}[XY] = \mathrm{E}[X]\,\mathrm{E}[Y]$.) For any $i$, the random variable $e^{\lambda X_i}$ takes values $e^{\lambda}$ and $e^{-\lambda}$, each with probability $1/2$. Its expectation is

$$\mathrm{E}\left[e^{\lambda X_i}\right] = \cosh(\lambda) = \frac{e^{\lambda} + e^{-\lambda}}{2}$$

The above quantity is bounded by

$$\frac{e^{\lambda} + e^{-\lambda}}{2} \leq e^{\lambda^2/2}.$$

One can prove the above inequality by analyzing the Taylor expansion of LHS and RHS. We do this analysis at the end. Using this bound, we have

$$\mathrm{E}\left[e^{\lambda S_n}\right] \leq \prod_{i=1}^{n} e^{\lambda^2/2} = e^{\lambda^2 n/2}$$

We can now apply Markov's inequality to get:

$$\Pr[S_n \geq a] = \Pr[e^{\lambda S_n} \geq e^{\lambda a}] \leq \frac{\mathrm{E}\left[e^{\lambda S_n}\right]}{e^{\lambda a}} \leq \frac{e^{\lambda^2 n/2}}{e^{\lambda a}}$$

The above inequality is true for any $\lambda$, so we are now free to choose $\lambda$ to optimize it. To get the tightest upper bound, we minimize the the exponent in the above function. Standard calculus technique shows that a minimum occurs at $\lambda = a/n$. This gives us

$$\Pr[S_n \geq a] \quad \leq \quad e^{-a^2/2n}$$

Finally, we show that

$$\frac{e^{\lambda} + e^{-\lambda}}{2} \leq e^{\lambda^2/2}.$$

Taylor expansion of LHS is

$$\cosh\lambda = 1 + \frac{\lambda^2}{2!} + \frac{\lambda^4}{4!} + \cdots$$

Taylor expansion for the function $e^x$ shows that

$$e^{\lambda^2/2} = 1 + \frac{\lambda^2}{2} + \frac{(\lambda^2/2)^2}{2!} + \frac{(\lambda^2/2)^3}{3!} + \cdots$$

Compare the two series term by term. The $k$th term of LHS is $\lambda^{2k}/(2k)!$, while that of $e^{\lambda^2/2}$ is $(\lambda^2/2)^k/k!$. Focusing only on the even factors, we have $(2k!) \geq 2^k \cdot k!$. Thus,

$$\cosh\lambda \leq e^{\lambda^2/2}.$$

There are several different forms of Chernoff Bound which will be useful. They all concern tail probabilities of sums of independent random variables. Let $\{X_1, X_2, \cdots, X_n\}$ be a set of $n$ independent 0-1 random variables, with

$$\Pr[X_i = 1] = p_i, \quad \Pr[X_i = 0] = 1 - p_i,$$

and let $p = \sum_{i=1}^{n} p_i/n$. We define the centralized random variables $\{Y_1, Y_2, \cdots, Y_n\}$ where $Y_i = X_i - p_i$, then

$$\Pr[Y_i = 1 - p_i] = p_i, \quad \Pr[Y_i = -p_i] = 1 - p_i,$$

and $\mathrm{E}[Y_i] = 0$. Let $S_n = \sum_{i=1}^{n} Y_i = \sum_{i=1}^{n} X_i - pn$.

Theorem 5.4 *For any* $\Delta > 0$,

$$\Pr[S_n \geq \Delta] \leq e^{-2\Delta^2/n}.$$

*By symmetry, the bound applies to* $-S_n$ *as well, and so*

$$\Pr[|S_n| \geq \Delta] \leq 2e^{-2\Delta^2/n}.$$

Note that when we take $\Delta = \delta pn$, then

$$\Pr[|\sum_{i=1}^{n} X_i - pn| > \delta pn] < 2e^{-2\delta^2 p^2 n}.$$

Here is another form. Let $\{X_1, X_2, \cdots, X_n\}$ be a set of $n$ independent 0-1 random variables, with $\Pr[X_i = 1] = p$. Then

Theorem 5.5 *For any* $0 < \delta < 1/2$,

$$\Pr[\sum_{i=1}^{n} X_i > (1+\delta)pn] < e^{-\delta^2 pn/4}$$

*and*

$$\Pr[\sum_{i=1}^{n} X_i < (1-\delta)pn] < e^{-\delta^2 pn/2}.$$

Theorem 5.5 is better than Theorem 5.4 for small $p$.

For not necessarily 0-1 random variables,

Theorem 5.6 *Let* $X_i$, $1 \leq i \leq n$, *be mutually independent with all* $\mathrm{E}[X_i] = 0$ *and all* $|X_i| \leq 1$. *Let* $S_n = \sum_{i=1}^{n} X_i$. *Then for all* $a > 0$,

$$\Pr[S_n > a] < e^{-a^2/2n}.$$

The proofs of these versions of Chernoff Bound all follow similar lines.

A version of this type of bound also holds with hypergeometric distribution. Randomly pick $n$ balls without replacement, from $N$ black and white balls, with $pN$ black balls. Let $S$ be the number of black balls among $n$ balls picked.

Then

Theorem 5.7 *For any* $0 \le t \le 1$,

$$\Pr[|S/n - p| \ge t] \le 2e^{-2nt^2}.$$

This is known as the *Hoeffding bound.*

### 5.1.4 Universal Hashing

Definition 5.8 (Universal family of hash functions) *Let* $U$ *and* $T$ *be finite sets. Let* $S$ *be an index set for a family of functions* $\{h_s : U \to T\}_{s\in S}$. $\{h_s\}_{s\in S}$ *is called a* universal family of hash functions *if* $\forall \alpha, \beta \in T, \forall x, y \in U, x \neq y$,

$$\Pr_{s\in S}[h_s(x) = \alpha \wedge h_s(y) = \beta] = \frac{1}{|T|^2}$$

Notice that the RHS of the equation above $1/|T|^2$ is the probability of getting $\alpha$ and $\beta$ when we choose two elements independently and uniformly at random from $T$.

The proper treatment in the Kolmogorov framework will be to define a measure space with the underlying set $S$, equipped with the uniform distribution. Then for all fixed $x \in U$, the map $Z_x : s \mapsto h_s(x)$ is a random variable.

We have $\forall x \neq y \in U$, $\forall \alpha, \beta \in T$, $\Pr_{s\in S}[Z_x(s) = \alpha \wedge Z_y(s) = \beta] = \frac{1}{|T|^2}$. Hence, $\forall \alpha \in T, \forall x \in U$, pick $y \in U$, and $y \neq x$, (we assume $|U| \ge 2$),

$$\begin{aligned}\Pr_{s\in S}[Z_x(s) = \alpha] &= \sum_{\beta\in T} \Pr_{s\in S}[Z_x(s) = \alpha \wedge Z_y(s) = \beta] \\ &= \sum_{\beta\in T} \frac{1}{|T|^2} \\ &= \frac{1}{|T|}\end{aligned}$$

So, $Z_x$ is a uniform random variable on $T$. It follows that for any $x \neq y \in U$ and $\alpha, \beta \in T$,

$$\Pr_{s\in S}[Z_x(s) = \alpha \ \wedge \ Z_y(s) = \beta] = \Pr_{s\in S}[Z_x(s) = \alpha] \cdot \Pr_{s\in S}[Z_y(s) = \beta].$$

So, for any $x \neq y \in U$, the random variables $Z_x$ and $Z_y$ are independent. The definition of universal hash function is equivalent to asking a set of *pairwise independent* and unifromly distributed random variables, $\{Z_x\}_{x\in U}$. The random variables in this set may be jointly dependent, but any two of them are independent.

However, instead of thinking in terms of pair-wise independent random variables $Z_x(s)$, we rather think of $h_s(x) = Z_x(s)$ as a random map from $U$ to $T$, by randomly choosing an index $s \in S$. The two views are of course completely equivalent here.

**An Example:** Let $p$ be a prime number. Then $\mathbb{Z}/p = \{0, 1, \ldots, p-1\}$ with the operations $+$ and $\cdot$ is a finite field. Consider the map $h_{s=(a,b)} : x \mapsto ax + b$ for $a, b \in \mathbb{Z}/p$. We will verify that $\{h_{(a,b)}\}_{a,b \in \mathbb{Z}/p}$ is a universal family of hash functions.

For all $x, y, \alpha, \beta \in \mathbb{Z}/p, x \neq y$, how many pairs $(a, b) \in \mathbb{Z}/p$ are there satisfying the following equations?

$$\begin{aligned} ax + b &= \alpha \\ ay + b &= \beta \end{aligned}$$

(In the above equations, $a$ and $b$ are the unknowns.) The determinant of this $2 \times 2$ linear system is

$$\det \begin{pmatrix} x & 1 \\ y & 1 \end{pmatrix} = x - y \neq 0.$$

Therefore, there exists a unique solution such that this equation holds. Thus,

$$\Pr_{s=(a,b) \in (\mathbb{Z}/p)^2} [h_s(x) = \alpha \wedge h_s(y) = \beta] = \frac{1}{p^2}.$$

So, $\{h_{(a,b)}\}_{a,b \in \mathbb{Z}/p}$ is a universal family of hash functions.

This can be generalized to any finite field $\mathrm{GF}(p^n)$. It is known that for any prime $p$ and any $n \geq 1$, there is a finite field of $p^n$ elements. Up to isomorphism such a field is unique. In particular, for any $k \geq 0$, the polynomial $X^{2 \cdot 3^k} + X^{3^k} + 1$ is an irreducible polynomial in $\mathbb{Z}_2[X]$, and therefore we have an explicit finite field in the form of

$$\mathbb{Z}_2[X]/(X^{2 \cdot 3^k} + X^{3^k} + 1).$$

In the definition of this particular family of universal hash functions via affine linear functions $ax + b$, if it is defined over a finite field $\mathrm{GF}(2^n)$, we can truncate any number of bits from $n$ to make $|T| = 2^k$, for any $0 \leq k \leq n$. Whenever in the following we speak of a family of universal hash functions, unless otherwise stated, we always refer to this family of affine linear functions, and if necessary, over that particular family of finite fields $\mathrm{GF}(2^n)$, with $n = 2 \cdot 3^k$.

The power of universal hash functions come from the fact that on the one hand they behave more or less like a random function, on the other hand they can be succintly specified by only $2n$ bits.

## 5.2 Randomized Algorithms: MAXCUT

As a taste for randomized algorithms, we discuss the graph problem MAXCUT. Let $G = (V, E)$ be an undirected graph over $n$ vertices. A cut $C$ of $G$ is a partition of vertices $V$ into disjoint union $V_1 \cup V_2$. We also identify a cut with the set of edges between $V_1$ and $V_2$, i.e., $e \in C$ consists of those edges with one of its incident vertices in $V_1$ and the other in $V_2$.

DEFINITION 5.9 (MAXCUT) *A* MAXCUT *of a graph $G = (V, E)$ is a cut $C$ such that $|C|$ is maximized over all cuts of $G$.*

Similar to MAXCUT, MINCUT of $G$ is defined as the minimum $|C|$ over all cuts of $G$. We know that MINCUT is in P. It can be solved using maximum network flow between all pairs of vertices.

We know that MAXCUT is NP-complete, therefore we do not expect to solve it efficiently. However, we can look for approximate solutions. To quantify the accuracy of our approximations we will introduce a new term. We want a polynomial-time algorithm that achieves a cut $C$ such that

$$\frac{|C|}{|C^*|} \geq r$$

where $C^*$ is a maximum cut. Such an algorithm is called an $r$-approximation.

### 5.2.1 Deterministic MAXCUT Approximation Algorithm

We can define a greedy algorithm that achieves a 1/2-approximation:
For a graph $G = (V, E)$ with $V = \{1, \ldots, n\}$, define $E_i = \{(k, i) \in E : k < i\}$. Initially, let $V_1 = \{1\}$ and $V_2 = \emptyset$. Then, for each $i$ from 2 to $n$, add $i$ to either $V_1$ or $V_2$ so that the number of edges in $E_i$ that are on the cut is maximized, i.e., put $i$ in $V_1$ iff $|\{k \in V_2 : (k, i) \in E_i\}| \geq |\{k \in V_1 : (k, i) \in E_i\}|$. We claim that this heuristic achieves 1/2-approximation.

Let $C$ be the cut obtained by the algorithm. The disjoint sets $E_1, E_2, \ldots, E_n$ partition $E$. So, $|E| = \sum_i E_i$. For each $i \in V$, let $E'_i = E_i \cap C$. Then, $C = \bigcup_i E'_i$. As sets $E_i$ are disjoint, the sets $E'_i$ are also disjoint. Thus, $|C| = \sum_i |E'_i|$. The main observation is that for each $i \in V$, we have $E'_i \geq E_i/2$. We conclude that $|C| \geq |E|/2$. As the size of maximum cut $|C^*| \leq |E|$, $|C| \geq |C^*|/2$.

### 5.2.2 Randomized MAXCUT Approximation Algorithm

We present a randomized 1/2-approximation algorithm for MAXCUT. Then we show that it can be derandomized in polynomial time. This example illustrates the ideas of randomization and derandomization in a simple setting.

The randomized algorithm is very simple. Assign a monkey at each vertex and have each monkey to throw a dart. If it throws to the left, assign the vertex to $V_1$, and if it throws to the right, assign it to $V_2$. More formally, given a graph $G = (V, E)$, we assign each vertex independently with equal probability to either $V_1$ or $V_2$. This will give us a cut $C$ of $G$, and we will show that the expected size of $C \geq |C^*|/2$.

Consider an edge $(i, j) \in E$. $\Pr[(i, j) \in C] = 1/2$. For $e \in E$, define $\chi_e$ to be a random variable such that $\chi_e = 1$ if $e \in C$ and $\chi_e = 0$ if $e \notin C$. Then $|C| = \sum_{e \in E} \chi_e$. Thus,

$$\mathrm{E}[|C|] = \sum_{e \in E} \mathrm{E}[\chi_e] = \sum_{e \in E} \Pr[e \in C] = \frac{|E|}{2} \geq \frac{1}{2}|C^*|.$$

The first equality follows from linearity of expectation, $E[X+Y] = E[X] + E[Y]$, for any two random variables $X$ and $Y$. This formula holds even if $X$ and $Y$ are not independent.

### 5.2.3 Derandomizing MAXCUT Approximation Algorithm Using Universal Hash Functions

Let $G = (V, E)$ be a graph with $V = \{0, \ldots, n-1\}$. Set $k$ so that $2^k \geq n > 2^{k-1}$. Choose $a$ and $b$ at random from $\mathrm{GF}(2^k)$. For each $i \in V$, treat $i$ as a member of $\mathrm{GF}[2^k]$ compute $ai + b$. Assign $i$ to either $V_1$ or $V_2$ according to the first bit of $ai + b$. We claim that the expected size of cut obtained is $|E|/2$.

Let $\chi_{(a,b)}(i) =$ the first bit of $ai + b$. We know that $\{ai + b\}_{a,b \in \mathrm{GF}(2^k)}$ is a universal family of hash functions. Thus, $\{\chi_{(a,b)}\}_{a,b \in \mathrm{GF}(2^k)}$ is a universal family of hash functions. Then, a cut $C$ obtained by the above randomized algorithm is given by $C = \{(i,j) | \chi_{(a,b)}(i) \neq \chi_{(a,b)}(j)\}$.

Because $\{\chi_{(a,b)}\}_{a,b \in \mathrm{GF}(2^k)}$ is a universal family of hash functions, $\Pr[\chi_{(a,b)}(i) \neq \chi_{(a,b)}(j)] = 1/2$. Thus, using the analysis from Section 3, we have $\mathrm{E}[|C|] = |E|/2$.

We can derandomize the above algorithm in polynomial time. There are less than $4n^2$ different choices for $(a, b)$. To derandomize, we can examine the cuts created by $\{\chi_{(a,b)}\}$ for all $a, b \in \mathrm{GF}(2^k)$ in polynomial time. One of these cuts is guaranteed to be at least $|C^*|/2$ because $\mathrm{E}[|C|] \geq |C^*|/2$. This gives us a deterministic $r$-approximation algorithm for MAXCUT.

Although this derandomized algorithm does not give a better approximation ratio than the greedy algorithm, it is a parallel algorithm. For each pair $(a, b)$, the determination of which side of the cut each vertex is on is independent of the other vertices. Thus, this can be executed in parallel. Additionally, the cut produced from each pair $(a, b)$ is independent of the cuts from other pairs. Therefore, all cuts can be determined in parallel. After these cuts have been computed, the maximum can be found with an NC computation.

### 5.2.4 Goemans-Williamson Algorithm

Goemans and Williamson gave an approximation algorithm for MAXCUT with error ratio of about 12%. Randomization plays an important role here combined with semidefinite optimization.

Typically a subset of $[n]$ can be described by a binary sequence in $\{0, 1\}^n$. Although this is not essential, we will find it here more convenient to describe such a subset $V_1 \subseteq V$, which corresponds to a cut $V_1 \cup (V \setminus V_1)$, by a vector $x \in \{-1,1\}^n$, by letting $x_i = -1$ iff $i \in V_1$. Then the cut size is $\sum_{e=\{i,j\}}(x_i - x_j)^2/4$. Therefore the MAXCUT problem seeks to maximize

$$\frac{1}{4}\sum_{e=\{i,j\}}(x_i - x_j)^2$$

subject to the constraint $x \in \{-1,1\}^n$. This constraint can be expressed as

$$x_i^2 = 1, \forall i \in V.$$

Such a problem is called a quadratic programming problem, which, in its generality, is also NP-hard (which is of course no surprise, since we have just reduced an NP-hard problem MAXCUT to it.)

The next trick is to linearize the problem by introducing a set of new variables $y_{ij}, 1 \leq i, j \leq n$, with the intention that $y_{ij} = x_i x_j$. Under this new set of variables, the objective function becomes

$$\frac{1}{4}\sum_{e=\{i,j\}}(y_{ii} + y_{jj} - 2y_{ij})$$

to be maximized subject to

$$y_{ii} = 1, \forall i \in V.$$

Observe that a "solution" $y_{ij}$ need not correspond to any real solution $x_i$. For example, if $y_{ij} = x_i x_j$, then the matrix $Y = (y_{ij})$ is symmetric and positive semi-definite, being the product of $X$ and its transpose $X^{\mathrm{T}}$,

$$Y = XX^{\mathrm{T}} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \left( \, x_1 x_2 \ldots x_n \, \right)$$

Not only that, since $X$ is $n \times 1$, $Y$ is of rank 1 (assuming $X \neq 0$, which is implied by $y_{ii} = 1$). However, we choose to ignore this rank condition, yet preserve the constraint that $Y$ is symmetric and positive semi-definite. The reason for this is that there is a polynomial time algorithm, based on the ellipsoid method, which solves this "semi-definite" programming problem optimally.

What we have done is called a "relaxation" of the original problem. The requirement of $Y$ being positive semi-definite amounts to a seemingly infinitely many inequalities

$$v^{\mathrm{T}} Y v \geq 0, \forall v \in \mathbf{R}^n.$$

These are linear constraints on $y_{ij}$. It is a trick of the "semi-definite" programming algorithm which can handle these infinitely many inequalities implicitly. We will not dwell further on this, and simply assume such a polynomial time algorithm is available.

Coming back to MAXCUT, in polynomial time we find a symmetric and positive semi-definite $Y^*$, with $y^*_{ii} = 1$ which maximizes $\frac{1}{4}\sum_{e=\{i,j\}}(y_{ii} + y_{jj} - 2y_{ij})$ among all $Y$ satisfying these constraints. Let the maximum value be $M^*$. Since this is a relaxation of the original problem, $M^*$ is an upper bound of the maximum cut size.

It is known that a symmetric and positive semi-definite matrix $Y^*$ can be expressed as a product of $UU^{\mathrm{T}}$, where $U^{\mathrm{T}} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}$, and where $u_i \in \mathbf{R}^n$. Moreover this decomposition can be found in polynomial time.

Thus, $y_{ii} = u_i^{\mathrm{T}} u_i = ||u_i||^2 = 1$ says that each vector $u_i$ is a unit vector. Moreover, "formally" retracing the expansion of $\sum_{e=\{i,j\}}(x_i - x_j)^2$ into $\sum_{e=\{i,j\}}(y_{ii} + y_{jj} - 2y_{ij})$, we get

$$M^* = \frac{1}{4} \sum_{e=\{i,j\}} ||u_i - u_j||^2.$$

The next idea of Goemans and Williamson is brilliant. *Randomly* choose a hyperplane $\Pi$ in $\mathbf{R}^n$. This amounts to choosing a unit vector $v \in \mathbf{R}^n$, uniformly on the unit sphere, as the normal vector to $\Pi$. (This can be carried out approximately with exponentially small error; we are ignoring issues of discretizing the process here.) Now partition $V$ according to which side of $\Pi$ the vector $u_i$ falls. More precisely, assign vertex $i$ to $V_1$ iff the inner product $\langle u_i, v\rangle > 0$.

If we express the cut size thus formed as a sum of 0-1 random variables, which indicate whether edge $e = \{i, j\}$ belongs to the cut, then the expectation is

$$\sum_{e=\{i,j\}} \Pr[\ \Pi \text{ separates } u_i, u_j].$$

To investigate this probability $\Pr[\ \Pi$ separates $u_i, u_j]$, we only need to think of it in terms of 2-dimensional space spanned by $u_i$ and $u_j$. Clearly this probability is $\theta_{ij}/\pi$ where $\theta_{ij}$ is the angle between $u_i$ and $u_j$. Meanwhile the length $||u_i - u_j||$ is clearly $2\sin\frac{\theta_{ij}}{2}$.

Simple calculus shows that the function

$$f(\theta) = \frac{\theta}{4}\sin^2\frac{\theta}{2}$$

achieves minimum .69002507... at 2.331122370.... It follows that the expectation of the cut

$$\sum_{e=\{i,j\}} \frac{\theta_{ij}}{\pi} \geq \frac{.69002507}{\pi} \sum_{e=\{i,j\}} ||u_i - u_j||^2 \approx 0.878 \cdot M^*.$$

This is a randomized MAXCUT algorithm that achieves 87.8% approximation ratio. There are ways to derandomize this algorithm, but we will not discuss this problem any further.

## 5.3 Randomized Algorithm: Two Stage Hashing

One simple application of universal hashing is a Dctionary data structure by two stage hashing developed by Fredman, Komlos, and Szemeredi (FKS). We will define a quantity called the collision number, use no more complicated a tool as the Markov inequality to prove its properties. The collision number is generally useful elsewhere as well.

Given a finite set $U$ and a subset $N \subseteq U$, we want to build an efficient dictionary. A dictionary is a data structure where we store information about the elements of $N$. Then, upon receiving a query $q \in U$, we want to look-up the dictionary and tell whether $q \in N$ or not efficiently. For example, let $U$ be the set of all "words" of length five over the English alphabet and let $N$ be the set of all proper English words of length five. This is where the name dictionary comes about. Various performance criteria of interest are time required to build the dictionary, space occupied by the dictionary and look-up time.

There are many deterministic schemes to implement a dictionary. These are usualy based on some balanced tree structure, and typically take $O(\log n)$ steps for each operation. Yao was the first to question this fundamental methodology ("Should tables be sorted") and proposed hashing as an efficient alternative. The FKS two-stage hashing scheme is extremely simple. It shows that given a universe $U$ and a subset $N \subseteq U$, where $n = |N|$, one can build a dictionary for $N$ that has the following properties:

- The dictionary can be built in $O(n)$ expected time.
- The dictionary occupies $O(n)$ space.
- Dictionary lookup takes $O(1)$ time.

Let us choose a set $T$ to be our hash table. Size of the table $|T|$ will be fixed later. We choose a suitable hash function $h : U \mapsto T$ and for each element $u \in U$, hash it to the bucket numbered $h(u)$ in $T$. Then, given a query $q \in U$, we can check $q \in N$ or not by scanning the bucket given by $h(q)$. We say that elements $a, b \in N$ collide under $h$, if $h(a) = h(b)$. It is clear that more the number of collisions, it will take more time to look-up. Thus, the goal is to choose a good hash function $h$ that minimizes the number of collisions, and yet does not use too much space.

Let $H$ be a universal family of hash functions from $U$ to $T$. Recall that such a family satisfies the following property. For any $u_1, u_2 \in U$ with $u_1 \neq u_2$,

$$\Pr_{h \in H}[h(u_1) = h(u_2)] = \frac{1}{|T|}.$$

Choose a hash function $h$ at random from $H$. Let $C$ be the random variable that gives the number of collisions under $h$. For $a, b \in N$, $a \neq b$, let $X_{a,b}$ be the indictor variable that denotes whether $a$ and $b$ collide under $h$ (i.e. $X_{a,b} = 1$, if $h(a) = h(b)$ and 0, otherwise). Then,

$$C = \sum_{\substack{\{a,b\} \subset N \\ a \neq b}} X_{a,b}$$

Since the probability that a pair $(a, b)$, where $a \neq b$, collides under a random hash function $h$ is $\frac{1}{|T|}$, the expected number of collisions is

$$E(C) = \binom{n}{2} / |T|$$

Set $|T| = n^2$. Then, $E(C) < \frac{1}{2}$. By Markov's inequality, $\Pr[C \geq 1] \leq \frac{1}{2}$. So, $\Pr[C < 1] \geq \frac{1}{2}$. Since $C$ is a integer variable, $C < 1$ implies that $C = 0$. Hence, if we choose $h$ at random, with probability at least $1/2$, there will be no collisions under $h$. We can try various $h$ at random until we find a $h$ with $C = 0$. The expected number of trials needed is just 2. This scheme is wonderful, except that, we have to allocate $O(n^2)$ space to get this bound with a single random hash function. We can do better by using the following two-stage hashing scheme.

In the first stage of the scheme, we allocate only $|T| = n$. Then, for a random $h \in H$, $E(C) < \frac{n}{2}$. Using Markov's inequality again, $\Pr[C \geq n] \leq \frac{1}{2}$. By trying various hash functions at random, we can find a hash function with $C \leq n$. The expected number of trial will be only 2. Suppose we have found a hash function $h_0$ with the number of collisions $C_0 \leq n$. Next we proceed to the second stage of the scheme.

Let $N_i$ be the set of elements of $N$ that got mapped to $i^{th}$ bucket of $T$ under $h_0$. Let $n_i = |N_i|$. In the second stage, for each bucket $1 \leq i \leq |T|$, we build a hash table $T_i$ for the set $N_i$, with $|T_i|$ to be $n_i^2$. Fix a universal family of hash functions $H_i : U \mapsto T_i$. Then, for a hash function $h$ chosen at random from $H_i$, the expected number of collisions is

$$E(C) = \binom{n_i}{2} \frac{1}{|T_i|} = \binom{n_i}{2} \frac{1}{n_i^2} < 1/2.$$

Thus, $\Pr[C > 1] \leq 1/2$, which implies $\Pr[C = 0] \geq 1/2$. For each $i$, by trying various $h$ at random, with an expected number of two trials, we can find a $h_i$ with no collisions. Thus in total we expect to try no more than $2n$ times.

What will be the total size of the hash table (both stages combined)? It is given by

$$\text{Total size} = |T| + \sum_{i=1}^{|T|} |T_i| = n + \sum_{i=1}^{n} n_i^2.$$

We now need a bound for the summation. We obtain a bound by using the fact that the number of collisions in first stage $C_0 \leq n$. We express $C_0$ in a different way to get the bound.

$$\begin{aligned} C_0 &= \sum_{i=1}^{n} \binom{n_i}{2} \\ &= \frac{1}{2}\left(\sum_{i=1}^{n} n_i^2 - \sum_{i=1}^{n} n_i\right) \end{aligned}$$

Note that $\sum_{i=1}^{n} n_i = |N| = n$. Thus,

$$\sum_{i=1}^{|T|} n_i^2 = 2C_0 + \sum_{i=1}^{n} n_i \leq 2n + n = O(n).$$

So, the total space needed to store the main hash table $T$ and the second level tables is only $O(n)$ (We will need space to store the hash functions $h_0$ and various chosen $h_i$. But this needs only $O(\log |U|)$ space, which we assume to be negligible). We next compute time required to build the tables. In the first stage, expected number of trials to find a "good" hash function $h_0$ is 2. Given a hash function $h$ we can tell whether it is good or bad in time $O(n)$. Thus, expected time for first stage is $O(n)$. In the second stage, for each bucket $i$, expected number of trials to find a good $h_i$ is 2 and the time needed to tell whether a $h$ is good or bad is $O(n_i)$. Thus, total expected time in second stage is $O(n_1) + O(n_2) + \cdots + O(n_{|T|}) = O(n)$.

Now let us turn to look-up operation. Given a query $q$, apply $h_0$ to $q$ to get the relevant bucket $i = h_0(q)$, then apply the second level hash function $h_i$ to $q$ and scan the bucket $h_i(q)$ in table $T_i$. As each $h_i$ is chosen to be collision-free, there will only one entry in the bucket $h_i(q)$. If this entry is $q$, then report that $q \in N$. If it is not, report that $q \notin N$. This process takes $O(1)$ time since we simply apply hash functions (We can construct universal family of hash functions, where computing any $h(x)$ takes time $O(\log |U|)$, which is assumed to be negligible).

## 5.4 Randomized Complexity Classes

### 5.4.1 Definitions

Definition 5.10 (BPP) *BPP stands for bounded error probabilistic polynomial time. A language $L$ is in* BPP*, if there is a boolean predicate $D(\cdot,\cdot)$, computable in deterministic polynomial time such that,*

$$\begin{aligned} x \in L &\implies \Pr_y[D(x,y) = 1] \geq 3/4 \\ x \notin L &\implies \Pr_y[D(x,y) = 1] \leq 1/4, \end{aligned}$$

*where the number of random bits used, $|y|$ is polynomial in length of the input $|x|$.*

DEFINITION 5.11 (RP) *RP was the earliest version of such classes, and simply stands for randomized polynomial time. A language $L$ is in* RP*, if there is a boolean predicate $D(\cdot,\cdot)$, computable in deterministic polynomial time such that,*

$$\begin{aligned} x \in L &\implies \Pr_y[D(x,y)=1] \geq 1/2 \\ x \notin L &\implies \Pr_y[D(x,y)=1] = 0, \end{aligned}$$

*where the number of random bits used, $|y|$ is polynomial in length of the input $|x|$.*

DEFINITION 5.12 (ZPP) *A language $L$ is said to be in* ZPP *if there is a polynomial time computable function $D : \{0,1\}^* \times \{0,1\}^* \mapsto \{0,1,?\}$ such that if $x \in L$, for any $y$, its output $D(x,y) \in \{1,?\}$, and if $x \notin L$, for any $y$, its output $D(x,y) \in \{0,?\}$. Moreover, $D$ should have high success probability:*

$$\begin{aligned} x \in L &\implies \Pr_y[D(x,y)=1] \geq 1/2 \\ x \notin L &\implies \Pr_y[D(x,y)=0] \geq 1/2, \end{aligned}$$

*where the number of random bits used, $|y|$ is polynomial in length of the input $|x|$.*

### 5.4.2 Amplification of BPP

In this section, we show that the probability of success of a BPP algorithm can be "amplified" to be as high as exponentially close to 1, with only a polynomial amount of extra work. We start with the Chernoff bound in Theorem 5.4.

Let $L$ be a language accepted by a probabilistic polynomial time TM $M$, in the following sense:

$$\begin{aligned} x \in L &\implies \Pr[M(x;r)=1] \geq \frac{1}{2}+\epsilon \\ x \notin L &\implies \Pr[M(x;r)=1] \leq \frac{1}{2}-\epsilon \end{aligned}$$

for random strings $r$ of an appropriate polynomial length in $n=|x|$. Here, $\epsilon$ can be as low as $1/p(n)$ for some fixed polynomial $p(\cdot)$. We wish to amplify the success probability of the algorithm. In particular, we want to get exponentially close to 1, meaning, for any fixed polynomial $q(\cdot)$, we want a machine $M'$ with

$$\begin{aligned} x \in L &\implies \Pr[M'(x;r')=1] \geq 1-e^{-q(n)} \\ x \notin L &\implies \Pr[M(x;r)=1] \leq e^{-q(n)} \end{aligned}$$

We require that $M'$ run in polynomial time, and hence the length of the random string $|r'|$ used by $M'$ should also be polynomially bounded.

The idea is to run $M$ a large polynomial number of times, and take the majority vote. Given an input $x$, $M'$ will simply run the machine $M$ on input $x$ some $2m+1$ times, with $m \geq q(n)/(4\epsilon^2)$, say, and accept $x$ iff at least $m+1$ runs of $M$ on $x$ accept. In this process we will need independent and uniformly chosen random $r_1, r_2, \cdots, r_{2m+1}$, with a total length of the random string $|r'| = O(|r|q(n)/\epsilon^2)$.

Let $X_i$ be the 0-1 random variable indicating the $i$-th run $M(x; r_i)$, $p = \Pr[M(x; r) = 1]$, and $S = \sum_i (X_i - p)$. Suppose $x \notin L$, then $p \leq \frac{1}{2} - \epsilon$. Apply Theorem 5.4 with $\Delta = (2m+1)\epsilon$, we get

$$\Pr[M'(x) \text{ accepts }] = \Pr[\sum_i X_i \geq m+1] \leq \Pr[S \geq (2m+1)\epsilon] \leq e^{-q(n)}.$$

Similarly, if $x \in L$, then $p \geq \frac{1}{2} + \epsilon$, and

$$\Pr[M'(x) \text{ rejects }] \leq e^{-q(n)}.$$

When $\epsilon^{-1}$ is polynomially bounded, so is $m = O(q(n)/\epsilon^2)$. This achieves exponentially small error probability.

## 5.5 Sipser-Lautemann Theorem: BPP $\subseteq$ PH

Let $L$ be a language in BPP. We will show that $L \in \Sigma_2^p$. Without loss of generality, there is a polynomial time computable predicate $D(\cdot, \cdot)$ such that,

$$\begin{aligned} x \in L &\implies \Pr_{y \in \{0,1\}^m}[D(x,y) = 1] \geq 1 - 1/m \\ x \notin L &\implies \Pr_{y \in \{0,1\}^m}[D(x,y) = 1] \leq 1/m \end{aligned}$$

where the number of random bits used $m$ is polynomially bounded in input length $n = |x|$. For any input $x \in \{0,1\}^n$, define its witness set $W_x = \{y \in \{0,1\}^m | D(x,y) = 1\}$. So, if $x \in L$ then the witness set $W_x$ is "fat", whereas, if $x \notin L$, the witness set is "thin". We will show how to test whether $W_x$ is fat or thin in $\Sigma_2^p$ and thereby prove that $L \in \Sigma_2^p$. Let us first formalize the notion of fat and thin sets and prove some properties of such sets.

We say that $S \subseteq \{0,1\}^m$ is *fat*, if $|S|/2^m \geq 1 - 1/m$. $S$ is said to be thin, if $|S|/2^m \leq 1/m$. In general a subset $S$ may be neither fat nor thin. But, the witness sets we are interested are always either fat or thin. For a string $u \in \{0,1\}^m$, define $S \oplus u = \{s \oplus u | s \in S\}$. Here, $s \oplus u$ denotes the $m$-bit string obtained by bit-wise XOR of $s$ and $u$: if $s = s_1 s_2 \ldots s_m$ and $u = u_1 u_2 \ldots u_m$, then $s \oplus u = s_1 \oplus u_1 \cdots s_m \oplus u_m$. Think of $\{0,1\}^m$ as a vector space, $S$ as a subset of this space and $u$ to be a vector in it. Then, $S \oplus u$ is nothing but the subset obtained by "shifting" $S$ by $u$.

We first discuss informally the effect of shifting fat and thin sets. Suppose $S$ is fat. Then, if we choose a suitable number of shift vectors $u_1, u_2, \ldots, u_r$ at random, with high probability, the union of these shifts will "cover" the entire space: $\bigcup_{i=1}^{r}(S \oplus u_i) = \{0,1\}^m$. On the other hand, if $S$ is a thin, then for any set of vectors $u_1, u_2, \ldots, u_r$, where $r < m$, the shifts will not cover the space: $\bigcup_{i=1}^{r}(S \oplus u_i) \neq \{0,1\}^m$. We will formally prove these claims. Observe that, with these claims, it is easy to put $L$ in $\Sigma_2^p$: the condition "there is a set of vectors such that $W_x$ shifted by these vectors covers the entire space" can be expressed as a $\Sigma_2^p$ predicate!

Lemma 5.13 *Let $S \subseteq \{0,1\}^m$.*

1. *If $S$ is thin, then for any $r < m$, for any set of $r$ vectors, the shifts cannot cover the entire space:*
$$\Pr_{u_1,u_2,\ldots,u_r \in \{0,1\}^m}\left[\bigcup_{i=1}^{r}(S \oplus u_i) = \{0,1\}^m\right] = 0.$$
2. *If $S$ is fat, then with high probability, randomly chosen shifts will cover the entire space:*
$$\Pr_{u_1,u_2,\ldots,u_r \in \{0,1\}^m}\left[\bigcup_{i=1}^{r}(S \oplus u_i) = \{0,1\}^m\right] \geq 1 - \frac{2^m}{m^r}$$

*Proof.* The first part is obvious. For any vectors $u_1, u_2, \ldots, u_r$, the union of the shifts has cardinality,
$$\left|\bigcup_{i=1}^{r}(S \oplus u_i)\right| \leq r \cdot |S|$$
Since we assume that $r < m$ and $S$ is thin, $|\bigcup_{i=1}^{r}(S \oplus u_i)| < 2^m$. Thus, the shifts cannot cover the entire space $\{0,1\}^m$ which has cardinality $2^m$.

To prove the second part, we will bound probability of the negation of the event under consideration. A set of vectors $u_1, u_2, \ldots, u_r$ do not cover the entire space means that some vector $y \in \{0,1\}^m$ is not covered by these shifts. So,
$$\Pr_{u_1,u_2,\ldots,u_r}\left[\bigcup_{i=1}^{r}(S \oplus u_i) \neq \{0,1\}^m\right] \leq \sum_{y \in \{0,1\}^m} \Pr_{u_1,u_2,\ldots,u_r}\left[y \notin \bigcup_{i=1}^{r}(S \oplus u_i)\right]$$
Fix any $y \in \{0,1\}^m$. By the properties of $\oplus$ function, we see that $r \notin S \oplus u$ iff $r \oplus u \notin S$ (we use the fact that for any $u$, $u \oplus u = 0$). Thus, $y$ will not be covered by the $r$ shifts iff $\{y \oplus u_1, y \oplus u_2, \ldots, y \oplus u_r\} \cap S = \emptyset$. For any $u$ chosen uniformly at random from $\{0,1\}^m$, $y \oplus u$ is distributed uniformly in $\{0,1\}^m$. (We use the fact that the function $f_y : u \mapsto y \oplus u$ is 1-1.) So, for a randomly chosen $u$, $\Pr[y \oplus u \notin S] \leq 1/m$, because $S$ is fat. By independence, it follows that
$$\Pr_{u_1,u_2,\ldots,u_r}[\{y \oplus u_1, y \oplus u_2, \ldots, y \oplus u_r\} \cap S = \emptyset] = \prod_{i=1}^{r} \Pr_{u_i}[y \oplus u_i \notin S] \leq \left(\frac{1}{m}\right)^r.$$

We conclude that,

$$\begin{aligned}
\Pr_{u_1,u_2,\ldots,u_r}\left[\bigcup_{i=1}^{r}(S\oplus u_i)\neq\{0,1\}^m\right] &\leq \sum_{y\in\{0,1\}^m}\Pr_{u_1,u_2,\ldots,u_r}\left[y\notin\bigcup_{i=1}^{r}(S\oplus u_i)\right]\\
&= 2^m\cdot\left(\frac{1}{m}\right)^r\\
&= \frac{2^m}{m^r}.
\end{aligned}$$

Part 2 of the lemma follows from the above bound. ♣

Using Lemma 5.13, we can show that BPP $\subseteq \Sigma_2^p$. For a suitable value of $r$, the lemma will show that if $S$ is fat, then for some set of $r$ vectors, the shifts would cover the entire space, and if $S$ is thin, then for any set of $r$ vectors, the shifts would not cover the entire space. The above property can be tested in $\Sigma_2^p$. Formal proof is given below.

Theorem 5.14 (Sipser–Lautemann) BPP $\subseteq \Sigma_2^p$.

*Proof.* Let $L \in$ BPP. Without loss of generality, there is a polynomial time computable predicate $D(\cdot,\cdot)$ such that,

$$\begin{aligned}
x\in L &\implies \Pr_{y\in\{0,1\}^m}[D(x,y)=1]\geq 1-1/m\\
x\notin L &\implies \Pr_{y\in\{0,1\}^m}[D(x,y)=1]\leq 1/m
\end{aligned}$$

where the number of random bits used $m$ is polynomially bounded in input length $n=|x|$. For an input $x\in\{0,1\}^n$, define its witness set $W_x=\{y|D(x,y)=1\}$. If $x\in L$ then the witness set $W_x$ is fat, whereas, if $x\notin L$, the witness set is thin. Choose $r=m/2$. Then, from Lemma 5.13, we have

$$\begin{aligned}
x\in L &\implies \Pr_{u_1,u_2,\ldots,u_r}\left[\bigcup_{i=1}^{r}(W_x\oplus u_i)=\{0,1\}^m\right]\geq 1-\frac{2^m}{m^r}\\
x\notin L &\implies \Pr_{u_1,u_2,\ldots,u_r}\left[\bigcup_{i=1}^{r}(W_x\oplus u_i)=\{0,1\}^m\right]=0.
\end{aligned}$$

For sufficiently large $m$, the first probability $>0$. So, if $x\in L$, there exists $r$ vectors such that the shifts cover the entire space, and if $x\notin L$, for any $r$ vectors, the shifts do not cover the entire space. Observe that, for any $u_1,u_2,\ldots,u_r$,

$$\bigcup_{i=1}^{r}(S\oplus u_i)=\{0,1\}^m \iff \forall y\in\{0,1\}^m\,[\vee_{i=1}^{r}(y\oplus u_i\in W_x)]$$

It follows that,

$$x \in L \iff \exists u_1, u_2, \ldots, u_r \forall y \in \{0,1\}^m \left[\vee_{i=1}^r (y \oplus u_i \in W_x)\right].$$

$y \oplus u_i \in W_x$ simply means that $D(x, y \oplus u_i) = 1$. So we conclude that,

$$x \in L \iff \exists u_1, u_2, \ldots, u_r \forall y \in \{0,1\}^m \left[\vee_{i=1}^r (D(x, y \oplus u_i) = 1)\right].$$

The predicate $\vee_{i=1}^r (D(x, y \oplus u_i) = 1)$ is testable in polynomial time, as $r$ is polynomial in $n$ and $D$ is a polynomial time predicate. We conclude that $L \in \Sigma_2^p$. ♣

BPP is closed under complementation. So, we have also shown that BPP $\in \Pi_2^p$. We can also prove this claim directly by exhibiting a $\Pi_2^p$ predicate. To do that, we first rephrase Lemma 5.13. Observe that, for any set $S \subseteq \{0,1\}^m$ and $u \in \{0,1\}^m$, $(S \oplus u)^c = S^c \oplus u$. Hence, for any $u_1, u_2, \ldots, u_r$,

$$\left[\bigcup_{i=1}^r (S \oplus u_i) = \{0,1\}^m\right] \iff \left[\bigcap_{i=1}^r (S^c \oplus u_i) = \emptyset\right].$$

Moreover, if $S$ is thin then $S^c$ is fat. So, Lemma 5.13 can be rephrased as follows. Suppose $S$ is fat. Then $S^c$ is thin. Applying Part 1 of the lemma to $S^c$, we see that for any set of $r < m$ shift vectors, $u_1, u_2, \ldots, u_m$, the intersection of the shifts is non-empty. Now suppose $S$ is thin. Then $S^c$ is fat. Applying Part 2 of the lemma to $S^c$, we see that for randomly chosen $r$ shift vectors $u_1, u_2, \ldots, u_m$, with high probability, the intersection of the shifts is empty. Formally,

Lemma 5.15 *Let $S \subseteq \{0,1\}^m$.*

1. *If $S$ is fat, then for any $r < m$, for any set of $r$ vectors, the intersections of the shifts is non-empty:*

$$\Pr_{u_1, u_2, \ldots, u_r}\left[\bigcap_{i=1}^r (S \oplus u_i) \neq \emptyset\right] = 1.$$

2. *If $S$ is thin, then with high probability, randomly chosen shifts will have an empty intersection:*

$$\Pr_{u_1, u_2, \ldots, u_r}\left[\bigcap_{i=1}^r (S \oplus u_i) = \emptyset\right] \geq 1 - \frac{2^m}{m^r}$$

Applying Lemma 5.15 to witness sets, with $r = m/2$, we have

$$x \in L \implies \forall u_1, u_2, \ldots, u_m \left[\bigcap_{i=1}^r (W_x \oplus u_i) \neq \emptyset\right]$$

$$x \notin L \implies \exists u_1, u_2, \ldots, u_m \left[\bigcap_{i=1}^r (W_x \oplus u_i) = \emptyset\right]$$

The above property can be expressed as a $\Pi_2^p$ predicate:

$$x \in L \iff \forall u_1, u_2, \ldots, u_m \exists y \left[\wedge_{i=1}^r (y \oplus u_i) \in W_x)\right]$$

## 5.6 Isolation Lemma

The isolation lemma provides a mechanism to approximately compute the size of some set $S \subseteq \{0,1\}^m$, such as a BPP witness set. In this section, we state and prove the lemma. In the subsequent sections, we use the lemma to give an alternative proof that BPP $\subseteq \Sigma_2^p$, and also discuss approximate counting.

Let $S$ and $T$ be finite sets and $\mathcal{H}$ be a universal family of hash functions from $S$ to $T$. Two distinct elements $x, y \in S$ are said to *collide* under a hash function $h \in \mathcal{H}$, if $h(x) = h(y)$. We say that a hash function $h$ *isolates* an element $x \in S$, if no element in $S$ collides with $x$. One can imagine that $h$ "likes" $x$ and gives it a separate seat in $T$ to sit alone comfortably! A set of hash functions $\{h_1, h_2, \ldots, h_r\}$ is said to isolate an element $x \in S$, if one of these function $h_i$ isolates $x$. The set of functions is said to *isolate all of* $S$, if for every element $x \in S$, there is some function $h_i$ in the set that isolates $x$.

Suppose we choose $r$ hash functions $h_1, h_2, \ldots, h_r$, uniformly and independently at random from $\mathcal{H}$. What is the probability that the set of these functions isolates all of $S$? The answer depends on the size of $S$ compared to the size of $T$ and the value of $r$. Intuitively, if $S$ is sufficiently smaller than $T$ and $h$ is large enough, with high probability, the randomly chosen set of functions will isolate all of $S$. On other hand, if $S$ is large compared to $T$ and $r$ is small enough, then there is not much space in $T$ to give "separate seats" for all the elements in $S$. For suitable choices of $|T|$ and $r$, we can make the probability of isolation zero.

Lemma 5.16 (Isolation Lemma) *Let $\mathcal{H}$ be a universal family of hash functions from S to $T$. Let $h_1, h_2, \ldots, h_r$ be chosen uniformly and independently at random from $\mathcal{H}$, where $r > 1$.*

1. *If $|S| \geq r|T|$, then*

$$\Pr_{h_1,h_2,\ldots,h_r}[\{h_1, h_2, \ldots, h_r\} \text{ isolates all of } S] = 0.$$

2. *Suppose $|S|^{r+1} \leq |T|^r$. Then,*

$$\Pr_{h_1,h_2,\ldots,h_r}[\{h_1, h_2, \ldots, h_r\} \text{ isolates all of } S] > 1 - \frac{|S|^{r+1}}{|T|^r}.$$

*Proof.* The first part of lemma is quite obvious. Any hash function $h \in \mathcal{H}$ can isolate at most $|T| - 1$ elements in $S$ (it can assign "separate seats" for some $|T| - 1$ elements in $S$ and then map all the other elements to the last "seat" in $T$). So, any set of $r$ hash functions can together isolate at most $r(|T| - 1)$ elements. As $|S| \geq r|T|$, no set of $r$ hash functions can isolate all of $S$.

We next prove the second part of the lemma. Fix any distinct elements $x, y \in S$. From the properties of universal family of hash functions, the probability that $x$ and $y$ collide under $h$ is $1/|T|$. So, for any element $x \in S$,

$$\begin{aligned}\Pr_{h\in\mathcal{H}}[h \text{ does not isolate } x] &\leq \sum_{y\in S-\{x\}} \Pr_{h\in\mathcal{H}}[x \text{ and } y \text{ collide under } h] \\ &< \frac{|S|}{|T|}\end{aligned}$$

It follows that, for any $x$ in $S$, if we choose $h_1, h_2, \ldots, h_r$ uniformly and independently at random,

$$\Pr_{h_1,h_2,\ldots,h_r}[\text{none of the } h_i \text{ isolate } x] < \left(\frac{|S|}{|T|}\right)^r.$$

Finally, the probability that randomly chosen $r$ hash functions do not isolate all of $S$ can be bounded by summing up over all possible elements in $S$:

$$\begin{aligned}&\Pr_{h_1,h_2,\ldots,h_r\in\mathcal{H}}[\{h_1, h_2, \ldots, h_r\} \text{ does not isolate all of } S] \\ \leq\ & \sum_{x\in S} \Pr_{h_1,h_2,\ldots,h_r\in\mathcal{H}}[\text{none of } h_1, h_2, \ldots, h_r \text{ isolate } x] \\ <\ & |S| \times \left(\frac{|S|}{|T|}\right)^r\end{aligned}$$

We conclude that

$$\Pr_{h_1,h_2,\ldots,h_r\in\mathcal{H}}[\{h_1, h_2, \ldots, h_r\} \text{ isolates all of } S\ ] > 1 - \frac{|S|^{r+1}}{|T|^r}$$

♣

## 5.7 BPP $\subseteq \Sigma_2^p$ - Another Proof Using the Isolation Lemma

This section shows how to use the isolation lemma to put BPP in $\Sigma_2^p$. We first present the proof idea informally. Let $L$ be a language in BPP, via a randomized algorithm that uses $m$ random bits, where $m$ is polynomial in the input length. We assume that the algorithm has been suitably amplified. We will fix parameters $r$ and size of the target set $T$ suitably in Lemma 5.16. For any input $x$, let $W_x \subseteq \{0,1\}^m$ be the set of witness strings on which the algorithm accepts the input $x$. The parameters will be chosen appropriately, so that if $x \in L$, $W_x$ will be large enough that no set of $r$ hash functions $h_1, h_2, \ldots, h_r$ will isolate all of $W_x$; on the other hand, if $x \notin L$, $W_x$ will be small enough so that, with high probability, randomly chosen $r$ hash functions $h_1, h_2, \ldots, h_r$ will isolate all of $W_x$, and in particular there will exist some set of $r$ hash functions that isolates all of $W_x$. Hence, $x \notin L$ iff there exist a set of $r$ hash functions $\{h_1, h_2, \ldots, h_r\}$ that isolates all of $W_x$. The later condition can be expressed as a $\Sigma_2^p$ predicate, thereby placing $L$ in $\Pi_2^p$. A formal proof follows.

Theorem 5.17 BPP $\subseteq \Sigma_2^p \cap \Pi_2^p$.

*Proof.* Let $L \in$ BPP. We prove that $L \in \Pi_2^p$. As BPP is closed under complementation, the lemma follows. Without loss of generality, there is a deterministic polynomial time boolean predicate $D(\cdot, \cdot)$ such that for any input $x \in \{0,1\}^n$,

$$\begin{aligned} x \in L &\implies \Pr_{y \in \{0,1\}^m}[D(x,y) = 1] \geq \frac{1}{2} \\ x \notin L &\implies \Pr_{y \in \{0,1\}^m}[D(x,y) = 1] \leq \frac{1}{4m} \end{aligned}$$

where the number of random bits used $m$ is polynomial in the input length $n$. Wolog we can assume $m$ is a power of 2. Fix $r = m$, and let $T$ be a set of size $2^m/(2m)$. (As it is usually the case in such proofs, the above choices of parameters such as the extent of amplification, value of $r$ and size of $|T|$ are not crucial. We can fix them in many ways to make the proof work! One such setting is given above). Let $\mathcal{H}$ be a universal family hash functions from $\{0,1\}^m$ to $T$. For any input $x$, let $W_x = \{y | D(x,y) = 1\}$. Suppose $x \in L$. Then,

$$|W_x| \geq 2^{m-1} = r \cdot |T|.$$

So, Lemma 5.16 shows that no set of $r$ hash functions $\{h_1, h_2, \ldots, h_r\}$ from $\mathcal{H}$ isolates all of $W_x$. On the other hand, suppose $x \notin L$. Then, from our choice of parameters,

$$\frac{|W_x|^{r+1}}{|T|^r} \leq \frac{1}{4m}.$$

Again, Lemma 5.16 implies that if we choose $h_1, h_2, \ldots, h_r$ uniformly and independently at random from $\mathcal{H}$,

$$\Pr_{h_1, h_2, \ldots, h_r}[\{h_1, h_2, \ldots, h_r\} \text{ isolates } W_x] \geq 1 - \frac{1}{4m}.$$

In particular, it follows that,

$$\begin{aligned} x \in L &\implies \forall h_1, h_2, \ldots, h_r[\{h_1, h_2, \ldots, h_r\} \text{ does not isolate } W_x] \\ x \notin L &\implies \exists h_1, h_2, \ldots, h_r[\{h_1, h_2, \ldots, h_r\} \text{ isolates } W_x] \end{aligned}$$

So, we have,

$$x \notin L \iff \exists h_1, h_2, \ldots, h_r[\{h_1, h_2, \ldots, h_r\} \text{ isolates } W_x].$$

Given particular $h_1, h_2, \ldots, h_r$, the predicate "$\{h_1, h_2, \ldots, h_r\}$ isolates $W_x$" can be expressed as

$$(\forall y \in W_x)(\exists 1 \leq i \leq r)(\forall y' \in W_x - \{y\})[h_i(y) \neq h_i(y')].$$

At first glance, it seems we need to write a $\Pi_3^p$ predicate to express the above condition. But, we can do better and express it as a coNP predicate, because $\exists 1 \leq i \leq r$ is a bounded quantifier and can be eliminated. This is a general principle in logic, but to be totally concrete, we have $\{h_1, h_2, \ldots, h_r\}$ isolates $W_x$ if and only if

$$(\forall y \in W_x)\{(\forall y_1 \in W_x - \{y\})[h_1(y) \neq h_1(y_1)] \vee \ldots \vee (\forall y_r \in W_x - \{y\})[h_r(y) \neq h_r(y_r)]\}$$

which is equivalent to

$$(\forall y \in W_x)(\forall y_1, \ldots, y_r \in W_x - \{y\})[(h_1(y) \neq h_1(y_1)) \vee \cdots \vee (h_r(y) \neq h_r(y_r)].$$

We conclude that, $x \notin L$ if and only if

$$\begin{array}{l}(\exists h_1, \ldots, h_r)(\forall y \in W_x)(\forall y_1, \ldots, y_r \in W_x - \{y\}) \\ \quad [(h_1(y) \neq h_1(y_1)) \vee (h_2(y) \neq h_2(y_2)) \vee \cdots \vee (h_r(y) \neq h_r(y_r)].\end{array}$$

We have shown that $L \in \Pi_2^p$. ♣

## 5.8 Approximate Counting Using Isolation Lemma

Using Sipser's Isolation Lemma, Stockmeyer showed how to do approximate counting in $\mathrm{P}^{\Sigma_2^p}$. In fact, his techniques does approximate counting at the level of $\mathrm{RP}^{\mathrm{NP}}$ already. Let us be more precise.

Definition 5.18 (#P) *#P is a function class. A function $f : \{0,1\}^* \to \mathbf{N}$ is in #P, iff there is a p-time NTM $M$, such that $f(x) = \#$ of accepting paths of $M(x)$.*

So typically any NP langugae has a "counting" version; e.g., #SAT($\phi$) is the number of satisfying assignments to the formula $\phi$; #HAM($G$) is the number of Hamiltonian circuits in $G$. One can easily develop a notion of polynomial time reduction for these functions. It is no surprise that both functions #SAT and #HAM are #P-complete under this reduction. This follows from the fact that Cook's reduction is parsimonious, i.e., they preserve the number of solutions. (For instance, in Cook's reduction from a generic NP langugae to SAT, every accepting computation is in a unique way associated with a satisfying assignment.)

Less obvious, yet also #P-complete, is the permanent function: For any $n$ by $n$ matrix $A = (a_{ij})$,

$$\mathrm{per}(A) = \sum_{\sigma \in S_n} a_{1,\sigma 1} a_{2,\sigma 2} \ldots a_{n,\sigma n},$$

where the sum is over all permutations $\sigma : i \mapsto \sigma i$. In other words, the permanent function is defined much as the determinant function, except there are no more minus signs. For a 0-1 matrix, per($A$) counts the number of perfect matchings of the bipartite graph with matrix $A$.

Valiant showed that the permanent function is also #P-complete (even though the decision problem of graph matching is in P.)

Definition 5.19 (PP) *PP stands for probabilistic polynomial time. A language $L$ is in* PP*, if there is a boolean predicate $D(\cdot,\cdot)$, computable in deterministic polynomial time such that,*

$$\begin{aligned} x \in L &\implies \Pr_{y\in\{0,1\}^m}[D(x,y)=1] \geq 1/2 \\ x \notin L &\implies \Pr_{y\in\{0,1\}^m}[D(x,y)=1] < 1/2, \end{aligned}$$

*where the number of random bits used, $m$, is polynomial in length of the input $|x|$.*

Note that, unlike BPP and RP, every polynomial time NTM defines a PP language, whereas not every polynomial time NTM defines a language in BPP or RP. BPP, RP and ZPP are "promise classes" in the sense that a polynomial time predicate (or equivalently a polynomial time NTM) defines a language in BPP, RP or ZPP only if it satisfies some global conditions. These conditions are, moreover, not decidable in general. Therefore we do not have an enumeration of these classes simply by an enumeration of their acceptors. This implies that we do not have a universal language, nor a complete language by this process. It is an open problem whether such complete languages exist. (Of course if BPP = P, then they indeed exist.) For PP, one can easily enumerate the class, and complete languages exist. For example, the set of Boolean formulae on $n$ variables having at least $2^{n-1}$ satisfying assignments is such a language.

It is also easy to see that

Theorem 5.20

$$\mathrm{P}^{\mathrm{PP}} = \mathrm{P}^{\#P}.$$

Next we discuss approximate counting. Given a formula $\varphi$ over $n$ variables, the goal is to approximately count the number of satisfying truth assignments of $\varphi$. Let $S = \{\sigma|\varphi(\sigma) = 1\}$ be the set of all such assignments. We want to compute the first $c \cdot \log n$ bits of $|S|$, where $c$ is any constant. This can be done by a polynomial time algorithm using a $\Sigma_2^p$ oracle, as shown by Stockmeyer. In fact, his technique can accomplish the same task with just an NP oracle, if we are ready to settle for a randomized polynomial time algorithm.

Our main tool is the isolation lemma, Lemma 5.16. We first rephrase the lemma in a more suitable format:

Lemma 5.21 *Let $S \subseteq \{0,1\}^n$, and let $\mathcal{H}$ be a family of 2-universal hash functions from $\{0,1\}^n$ to $\{0,1\}^k$. For all $m \geq k$, choose $h_1, h_2, \ldots, h_m$ independently at random from $\mathcal{H}$.*

1. *if $|S| \leq 2^{k-1}$ then*

$$\Pr_{h_1,\ldots,h_m}[\forall x \in S \text{ some } h_i \text{ isolates } x\,] \geq 1 - \frac{1}{2^{m-k+1}}$$

*2. if $|S| > m2^k$ then*

$$\Pr_{h_1,\dots,h_m}[\forall x \in S \text{ some } h_i \text{ isolates } x\,] = 0.$$

The idea is to try all values of $k$ from 1 to $n$, and attempt to find a $k$ such that $|S| \approx 2^k$. (We may assume our $S \neq \emptyset$; at any rate with one query to SAT we can verify this.) For any $\emptyset \neq S \subseteq \{0,1\}^n$, there is some $k_S$, where $1 \leq k_S \leq n$, such that $2^{k_S-1} \leq |S| \leq 2^{k_S}$. If we take every $k$ in the range $1 \leq k \leq n+1$, and randomly pick $m = 2n$ hash functions $h_1, \dots, h_m : \{0,1\}^n \to \{0,1\}^k$, then for each $k \geq k_S + 1$, we would get *isolation* with probability $\geq 1 - \frac{1}{2^n}$. For each $k$ we ask the SAT oracle, whether the chosen set of $h_1, \dots, h_m$ has the property that "$\forall x \in S$, one of $h_i$ isolates $x$". Since there are only $m = 2n$ hash functions this is a SAT query. We pick the least $k_0$ such that the oracle confirms *isolation*. Then $k_0 \leq k_S + 1$, with probability $\geq 1 - \frac{1}{2^n}$. (We abort if for no $k$ the chosen hash functions achieve *isolation*; this happens with exponentially small probability.) Also by the second part of Lemma 5.21, we know definitely $|S| \leq 2n2^{k_0}$. Denote by $U = 2n2^{k_0}$, then with high probability,

$$\frac{U}{8n} \leq |S| \leq U.$$

This give us a randomized polynomial time algorithm using a SAT oracle to approximate $|S|$ within $O(n)$. We'd like to do better. To do this, we use a little trick to amplify the accuracy.

First we build a set $S'$ such that

$$S' = \underbrace{S \times S \times \cdots \times S}_{m \text{ times}} \quad \subseteq \{0,1\}^{nm}$$

where $m$ is polynomial in $n$. Then we run the previous algorithm on the set $S'$ to get an estimate $U'$ for $S'$ such that

$$\frac{U'}{8nm} \leq |S'| \leq U'.$$

Now we set $e(S) = (U')^{1/m}$. It follows that

$$\frac{e(S)}{(8nm)^{1/m}} \leq |S| \leq e(S).$$

By choosing $m$ to be a sufficiently large polynomial in $n$, we can get

$$e(S) \cdot \left(1 - \frac{1}{n^c}\right) \leq |S| \leq e(S),$$

for any constant $c$.

## 5.9 Unique Satisfiability: Valiant–Vazirani Theorem

Assuming NP $\neq$ P, SAT cannot be solved in polynomial time. One may suspect that it is difficult to design a polynomial time algorithm for SAT because the input formula may have myriad truth assignments and it is hard to get hands on one of them. This suspicion leads to the following interesting problem, called USAT (Unique SAT). We are given a formula $\varphi$ which is guaranteed to be either unsatisfiable or has exactly one satisfying truth assignment. Is USAT solvable in polynomial time? Here we show some evidence that it is unlikely. We prove that if USAT is solvable in polynomial time, then NP = RP.

Theorem 5.22 *Suppose there is a polynomial time algorithm $A$, which for a given boolean formula $\varphi$, answers*

$$A(\varphi) = \begin{cases} \text{No} & \textit{if } \varphi \textit{ has no satisfying assignments;} \\ \text{Yes} & \textit{if } \varphi \textit{ has exactly one satisfying assignment;} \\ \text{Yes/No} & \textit{if } \varphi \textit{ has more than one satisfying assignments.} \end{cases}$$

*Then* NP = RP.

*Proof.* Note that the algorithm $A$ may output anything (YES or NO) if it is given a formula with more than one satisfying truth assignment. Assuming $A$ runs in polynomial time, we design a RP algorithm for SAT. The idea is to use some coin tosses and convert $\varphi$ into more constrained formula $\varphi'$, so that if $\varphi$ is unsatisfiable, $\varphi'$ will also be unsatisfiable. And if $\varphi$ is satisfiable, with non-trivial probability, exactly one of the satisfying assignments of $\varphi$ will satisfy $\varphi'$. Once we are successful in obtaining such a $\varphi'$, we can apply the algorithm $A$ to it.

To start with assume that $\varphi$ is satisfiable and let $\#\varphi$ be the number of satisfying truth assignments of $\varphi$. In order to convert $\varphi$ to $\varphi'$, the algorithm needs an estimate for $\#\varphi$. It is computationally hard to estimate $\#\varphi$. So, we simply pick a $k$ with $1 \leq k \leq n$ at random! The hope is that $2^{k-1} \leq \#\varphi \leq 2^k$. The probability that $k$ satisfies the above condition is $1/n$, as long as $\#\varphi \neq 0$. Assume that we are lucky and $k$ indeed satisfies the above condition. Next we choose a set $T$ of size $2^{k+1}$. If $2^{k-1} \leq \#\varphi \leq 2^k$, then $2(\#\varphi) \leq |T| \leq 4(\#\varphi)$, meaning $|T|$ is neither too big nor too small compared to $\#\varphi$. We then setup a universal family of hash functions $H$ from the set of all assignments $\{0,1\}^n$ to $T$, and we randomly pick a hash function $h$ from $H$ and an element $\alpha$ from $T$. We will show that, with non-trivial probability, there will be a *unique* satisfying truth assignment $x$ such that $h(x) = \alpha$. So, we consider the question: "Is there a truth assignment $t$ such that $\varphi(t) = 1$ and $h(t) = \alpha$". By Cook's theorem this can be converted to a SAT question $\varphi'$, and it will have a unique satisfying assignment iff there is a unique $t$ satisfying $\varphi$, and is mapped to $\alpha$ by $h$. If we are lucky in choosing the "correct" $k$, $h$ and $\alpha$, $\varphi'$ will have exactly one satisfying truth assignment.

**Algorithm for SAT:**
Input: A formula $\varphi$ over $n$ variables.

1. Choose a number $1 \leq k \leq n$ uniformly at random.

2. Let $T = \{0,1\}^{k+1}$, let $H$ be a universal family of hash functions from $\{0,1\}^n$ to $T$.

3. Choose $h \in H$ and $\alpha \in T$ uniformly at random.

4. Let $\varphi'$ be the Boolean formula from Cook's theorem, encoding the NP predicate "$(\exists t \in \{0,1\}^n)[(\varphi(t) = 1) \wedge (h(t) = \alpha)]$".

5. Run the procedure $A$ for USAT on $\varphi'$. *Accept* $\varphi$ iff $A(\varphi') =$ accept and the truth assignment extracted using $A$ via self-reducibility indeed satisfies $\varphi$.

It is clear that, if $\varphi$ is unsatisfiable then our algorithm will reject it with probability one. So, assume that $\varphi$ is satisfiable. Then over the random choices of $k, h$ and $\alpha$, we show that our algorithm accepts $\varphi$ with probability $\Omega(1/n)$. This probability can then be easily amplified, making it an RP algorithm.

First of all suppose $2^{k-1} \leq \#\varphi \leq 2^k$. The probability that $k$ satisfies the above condition is $1/n$. Under this assumption, we have $2(\#\varphi) \leq |T| \leq 4(\#\varphi)$. We say that two *satisfying* distinct truth assignments $a$ and $b$ (with $a \neq b$) collide under $h$, if $h(a) = h(b)$. Over the random choice $h$, let $C$ be the random variable that counts the number of collisions. We first show that the expected number of collisions $\mathrm{E}[C]$ is small. For any fixed $a \neq b$, the probability that $a$ and $b$ collide is $1/|T|$ (this follows from the definition of universal family of hash functions). For any pair of satisfying truth assignments $a \neq b$, let $X_{a,b}$ be a 0-1 random variable such that $X_{a,b} = 1$, if $a$ and $b$ collide under $h$ and $X_{a,b} = 0$, if they don't collide. Its expectation is $\mathrm{E}[X_{a,b}] = 1/|T|$. The number of collisions $C$ is the sum of over all $X_{a,b}$. So, expectation of $C$ is

$$\mathrm{E}[C] = \sum_{(a,b)} \mathrm{E}[X_{a,b}].$$

The summation above ranges over all distinct pairs of satisfying truth assignments. The number of such pairs is $\binom{\#\varphi}{2}$. Hence,

$$\mathrm{E}[C] = \binom{\#\varphi}{2} \frac{1}{|T|} \leq \frac{\#\varphi}{4}.$$

Using Markov's inequality, we have

$$\Pr\left[C \geq \frac{\#\varphi}{3}\right] \leq \frac{E[C]}{\#\varphi/3} \leq \frac{3}{4}.$$

So, with probability at least $1/4$, the number of collisions $C$ is at most $\#\varphi/3$. In general, if there are $c$ collisions, at most $2c$ satisfying truth assignments can participate in collisions. (This can be shown easily by the inductive argument: Take any such $x$ and pick any $y \neq x$ such that $h(x) = h(y)$. Now remove $x$ and $y$. There can be at most $c - 1$ collisions left among the remaining points, and thus at most $2c - 2$ points.) Thus, in our case, at most

$2/3 \cdot \#\varphi$ satisfying truth assignments can participate in some collision. It follows that at least $\#\varphi/3$ satisfying truth assignments are mapped to a unique image. Call these $\geq \#\varphi/3$ images in $T$ *good*: i.e., there is a unique satisfying assignment $a$ such that $h(a) = t$. Recall that $|T| \leq 4(\#\varphi)$. So, at least $1/12$ fraction of elements in $T$ are good. Therefore, with probability at least $1/12$, the randomly picked element $\alpha$ is good.

We can now lower bound the probability that $\varphi'$ has a unique satisfying truth assignment. Assume that $\varphi$ is satisfiable. Then, with probability $1/n$, the number $k$ chosen by the algorithm satisfies $2^{k-1} \leq \#\varphi \leq 2^k$. Assuming $k$ satisfies the above condition, with probability at least $1/4$, the number of collision $C$ is at most $\#\varphi/3$. Assuming $C \leq \#\varphi/3$, with probability at least $1/12$, the randomly chosen $\alpha$ is good. If $\alpha$ is good, then $\varphi'$ has exactly one satisfying truth assignment. Putting together,

$$\Pr_{k,h,\alpha}[\varphi' \text{ has a unique satisfying truth assignment}] \geq \frac{1}{n} \cdot \frac{1}{4} \cdot \frac{1}{12} = \frac{1}{48n}$$

As we noted already, if $\varphi$ is unsatisfiable, our algorithm has zero probability of accepting it. If $\varphi$ is satisfiable, our algorithm accepts it with probability at least $1/(48n)$. So our algorithm is an RP algorithm with success probability $1/(48n)$. We can amplify this success probability, as usual, by running the algorithm multiple times. For example, by running the algorithm $96n$ times, the success probability can be amplified to $1/2$. ♣

## 5.10 Efficient Amplification

Suppose we have some language $L$ in RP. Thus membership of $x$ in $L$ can be determined by running over witness strings $y \in \{0,1\}^n$, for some $n = |x|^{O(1)}$, and achieving

$$\begin{aligned} x \in L &\implies \Pr_y[D(x,y) = 1] \geq 1/2 \\ x \notin L &\implies \Pr_y[D(x,y) = 1] = 0. \end{aligned}$$

A naive method of amplification of this probability can be obtained by running independent $y_1, \ldots, y_k \in \{0,1\}^n$. Thus, with $kn$ random bits, we can achieve error probability $1/2^k$. A similar process with majority vote can be carried out for BPP langugaes, justified by the Chernoff bound. This probability is the ratio of the number of "bad" coin flips ($2^{k(n-1)}$) over the total number of coin flips ($2^{kn}$). Is there any way to achieve this reduction of error probability more efficiently?

### 5.10.1 Chor-Goldreich Generator

It turns out that there are several methods. Here is a first try. Let $\{h_s\}$ be a family of universal hash functions from $\{0,1\}^n \to \{0,1\}^n$. Now by randomly choosing $s$, we simply take the pairwise independent samples $y_i = h_s(i)$ as "witness" strings, for $i = 1, \ldots, k$. (We consider $i = 1, \ldots, k$ as embedded in $\{0,1\}^n$, as long as $k \le 2^n$.) For our RP langugae $L$, clearly if $x \notin L$ then all $D(x, y_i) = 0$. Suppose $x \in L$, we estimate the error probability that all $D(x, y_i) = 0$. We do this by using the Chebechev bound. Let $Z_i$ be the 0-1 r.v. such that $Z_i = 1$ iff $D(x, y_i) = 1$.

Theorem 5.23 *If $x \in L$, then*

$$\Pr_s[(\forall 1 \le i \le k) D(x, y_i) = 0] \le 1/k.$$

The proof is a simple application of the Chebechev bound. Let $x \in L$. Then $Z_i$ are pair-wise independent with expectation $\mu = \mathrm{E}[Z_i] = \Pr_y[D(x, y) = 1] \ge 1/2$, and as a 0-1 r.v. $Z_i$ has variance $\le 1/4$. Then

$$\Pr_s[(\forall 1 \le i \le k) D(x, y_i) = 0] = \Pr_s\left[\sum_{i=1}^{k} Z_i = 0\right] \le \Pr_s\left[\left|\sum_{i=1}^{k} Z_i - k\mu\right| \ge k/2\right] \le 1/k.$$

This is called the Chor-Goldreich generator, which achieves error $1/k$ with $2n$ bits. Of course one can not use this generator to achieve exponentially small error probability since this would require exponentially many evaluations of the hash function.

### 5.10.2 Hash Mixing Lemma and Nisan's Generator

The following generator of Nisan's was primarily devised for space bounded computations. But it can also be used for deterministic amplification.

Again let $\{h_s\}$ be a family of universal hash functions from $\{0,1\}^n \to \{0,1\}^n$. Pick uniformly and independently $s_1, \ldots, s_k$, and consider the following recursive definition of $G_i(y; s_1, \ldots, s_i)$, for $0 \le i \le k$. First $G_0(y) = y$, and for $i \ge 0$,

$$G_{i+1}(y; s_1, \ldots, s_{i+1}) = G_i(y; s_1, \ldots, s_i) \circ G_i(h_{s_{i+1}}(y); s_1, \ldots, s_i),$$

where $\circ$ denotes concatenation. Thus, $G_1(y; s_1) = y \circ h_{s_1}(y)$, $G_2(y; s_1, s_2) = y \circ h_{s_1}(y) \circ h_{s_2}(y) \circ h_{s_1}(h_{s_2}(y))$, etc. It may appear that in $G_1(y; s_1)$, it is not very clever since it uses $3n$ bits to produce only $2n$ (not even quite uniform) bits. The genius of this generator comes when you realize that this generator exponentially stretches its input seed. For $G_k$, it takes $(2k+1)n$ bits, and produces $2^k n$ bits. Also note that for any particular bit position, it is very simple to compute this bit in a random access manner, without recursion and without even having to compute the earlier bits sequentially. For $G_k$, the $i$ th block of $n$ bits can be directly obtained as follows. Let the $k$-bit binary representation of $i$ be $i_k i_{k-1} \ldots i_1$, then the $i$ th block of $n$ bits is $h_{s_1}^{i_1} h_{s_2}^{i_2} \ldots h_{s_k}^{i_k}(y)$, where $h_{s_j}^1$ denotes $h_{s_j}$ and $h_{s_j}^0$ denotes the identity function. With read-only access to the hash function seeds $s_1, s_2, \ldots, s_k$, this can be computed in space $O(k+n)$. These considerations will become important for its primary intended purpose for space bounded computations; but they are not crucial for our purpose of amplification here.

How good are these bits? The following lemma is the technical tool to address this question.

LEMMA 5.24 (HASH MIXING LEMMA) *Let $\epsilon = 2^{-n/3}$. Then for all subset $E \subseteq \{0,1\}^n \times \{0,1\}^n$, for all but an $\epsilon/4$ fraction of $s$,*

$$\mid \Pr_{y \in \{0,1\}^n}[y \circ h_s(y) \in E] - \mu[E] \mid < \epsilon,$$

*where, $\mu[E]$ is the probability measure of the set $E$, i.e., $\mu[E] = \Pr_{y,z \in \{0,1\}^n}[y \circ z \in E]$.*

*Proof.* We want to estimate the fraction $s$ such that it is "bad":

$$\mid \Pr_{y \in \{0,1\}^n}[y \circ h_s(y) \in E] - \mu[E] \mid \geq \epsilon. \tag{5.1}$$

For any fixed $y$ we define the indicator random variable (random over $s$)

$$Z_y^{h_s} = \begin{cases} 1 & \text{if } y \circ h_s(y) \in E \\ 0 & \text{otherwise.} \end{cases}$$

We can write $\Pr_{y \in \{0,1\}^n}[y \circ h_s(y) \in E]$ as a sum of pair-wise independent random variables (random over $s$)

$$\frac{1}{2^n} \sum_{y \in \{0,1\}^n} \mathbf{1}_{[y \circ h_s(y) \in E]} = \frac{1}{2^n} \sum_y Z_y^{h_s},$$

then it has expectation

$$\frac{1}{2^n} \sum_{y \in \{0,1\}^n} \mu'[E_y] = \mu[E],$$

where $E_y$ is the fibre-set over $y$, $E_y = \{z \mid y \circ z \in E\}$, and $\mu'[E_y]$ is the 1-dimensional probabiliy measure $\mu'[E_y] = |E_y|/2^n$.

Therefore we can use Chebechev inequality to estimate this deviation. Thus the "bad" event (5.1) over $s$ has probability at most

$$\begin{aligned} \frac{1}{\epsilon^2}\mathbf{Var}\left[\frac{1}{2^n}\sum_y Z_y^{h_s}\right] &= \frac{1}{\epsilon^2}\frac{1}{2^{2n}}\sum_y \mathbf{Var}[Z_y^{h_s}] & (5.2) \\ &\leq \frac{1}{\epsilon^2 2^n 4}, & (5.3) \end{aligned}$$

where the first equality uses the fact that $Z_y^{h_s}$ are pair-wise independent (over distinct $y$'s), and the second inequality follows from the trivial fact that a 0-1 random variable has variance at most 1/4.

Hence, for $\epsilon = 2^{-n/3}$, for all but a $\epsilon/4$ fraction of $s$,

$$\mid \Pr_{y \in \{0,1\}^n}[y \circ h_s(y) \in E] - \mu[E] \mid < \epsilon.$$

♣

We now return to our problem of deterministic amplification for RP languages. Suppose $x \in L$, and the witness set is $W_x \subseteq \{0,1\}^n$. Assume $n \geq 9$ and $\mu(\overline{W_x}) \leq 1/2$ is the error probability of a single random try.

Theorem 5.25 *If $x \in L$ then*

$$\Pr[G_k(y; s_1, \ldots, s_k) \subseteq \overline{W_x}] \leq (\mu(\overline{W_x}))^{2^k} + (k/4 + 2)\epsilon,$$

*where $\epsilon = 2^{-n/3}$.*

*Proof.* The intuitive idea is to use the $k\epsilon$ term to take care of the exceptional probability of "bad" hash functions $h_{s_i}$ in $k$ "rounds", and for "good" hash functions, the estimate should be as if it were uniform, with error term $O(\epsilon)$.

For $1 \leq i \leq k$, fixing any $s_1, \ldots, s_{i-1}$, define the set

$$A_i = \{y \mid G_i(y; s_1, \ldots, s_{i-1}) \subseteq \overline{W_x}\}$$

We will apply the Hash Mixing Lemma to the set $E_i = A_i \times A_i$. We say that $s_i$ is bad for $E_i$ (with respect to the fixed $s_1, \ldots, s_{i-1}$) if $s_i$ belongs to the $\epsilon/4$ fraction of exceptional $s$ in the Hash Mixing Lemma. Note that which $s_i$ is bad depends on $s_1, \ldots, s_{i-1}$. However, for any $s_1, \ldots, s_{i-1}$ there can be at most $\epsilon/4$ fraction of $s$ that can be bad, the rest are "good".

We will use the integration notation $\int$ over the probability measures on $s$ and $y$. Then

$$\begin{aligned} & \Pr[G_k(y; s_1, \ldots, s_k) \subseteq \overline{W_x}] && (5.4) \\ = \; & \int_{s_1 \text{ bad}} + \int_{s_1 \text{ good}} \int_{s_2 \text{ bad}} + \cdots + \int_{s_1 \text{ good}} \int_{s_2 \text{ good}} \cdots \int_{s_{k-1} \text{ good}} \int_{s_k \text{ bad}} && (5.5) \\ & + \int_{s_1 \text{ good}} \int_{s_2 \text{ good}} \cdots \int_{s_k \text{ good}} \int_{y}. && (5.6) \end{aligned}$$

The first $k$ terms are all $\leq \epsilon/4$. Note that for the $i$ th term, each $\int_{s_1}, \ldots, \int_{s_{i-1}}$ is over a probability measure.

Now we fix $s_1, \ldots, s_k$ where $s_i$ is good on $E_i$ for $s_1, \ldots s_{i-1}$, and consider

$$\int_y = \Pr_y[G_k(y; s_1, \ldots, s_k) \subseteq \overline{W_x}].$$

Write $\beta = \mu(\overline{W_x})$. Then $\beta \leq 1/2$. We will prove by induction that

$$\int_y \leq \beta^{2^k} + 2\epsilon.$$

Any upper bound on this probability becomes an upper bound of $\int_{s_1 \text{ good}} \int_{s_2 \text{ good}} \cdots \int_{s_k \text{ good}} \int_y$, as each $\int_{s_i}$ is over a probability measure.

For $k = 0$, trivially $\int_y = \beta$. For $k = 1$, by the Hash Mixing Lemma

$$\int_y \leq \Pr_{y,z}[G_1(y; s_1) \circ G_1(z; s_1) \subseteq \overline{W_x}] + \epsilon \leq \beta^2 + \epsilon.$$

Similarly, for $k = 2$,

$$\int_y \leq (\beta^2 + \epsilon)^2 + \epsilon \leq \beta^{2^2} + 2\epsilon,$$

since $2\beta^2 + \epsilon \leq 1$. For the general $G_{k+1}$, with $k \geq 2$,

$$\int_y \leq (\beta^{2^k} + 2\epsilon)^2 + \epsilon \leq \beta^{2^{k+1}} + 2\epsilon.$$

Here we used $n \geq 3$ and therefore $\epsilon \leq 1/8$.

This proves that

$$\Pr[G_k(y; s_1, \ldots, s_k) \subseteq \overline{W_x}] \leq (\mu(\overline{W_x}))^{2^k} + (k/4 + 2)\epsilon,$$

For $k \geq 3$, a cleaner expression holds:

$$\Pr[G_k(y; s_1, \ldots, s_k) \subseteq \overline{W_x}] \leq (\mu(\overline{W_x}))^{2^k} + k\epsilon.$$

♣

In terms of the number of random bits needed to achieve error probability $2^{-k}$, Nisan's generator uses $O(n \log k)$ bits.

### 5.10.3 Leftover Hash Lemma and Impagliazzo-Zuckerman Generator

Take $\ell$ and $k$ as integer parameters. (A reasonable choice will be $\ell \approx k \approx \sqrt{n}$ to balance out the bounds.) This generator produces $k + 1$ blocks of $n$-bits each, using a source of $3n + k\ell$ bits. Let $\{h_s\}$ be a family of universal hash functions from $\{0,1\}^n \to \{0,1\}^{n-\ell}$. The generator $G$ is defined from $\{0,1\}^{2n} \times \{0,1\}^n \times (\{0,1\}^\ell)^k \to (\{0,1\}^n)^{k+1}$ as follows:

$$G(s, Y_1, Z_1, \ldots, Z_k) = Y_1 \circ Y_2 \circ \cdots \circ Y_{k+1},$$

where $\circ$ denotes concatenation, $s \in \{0,1\}^{2n}$, $Y_1 \in \{0,1\}^n$, $Z_i \in \{0,1\}^\ell$, and

$$Y_{i+1} = h_s(Y_i) \circ Z_i,$$

for $1 \leq i \leq k$.

The way this should be read intuitively is this: We first take $Y_1$ as is. When we use $Y_1$ to test for membership of an RP language, not all of $n$-bits of randomness in $Y_1$ is used. So we should use a hash function to "extract" some "leftover" randomness, getting $h_s(Y_1)$. We supply some fresh true random bits $Z_1$ to make up $Y_2$. Then we repeat this process at $Y_2$, until $Y_k$.

How good is this generator? The technical tool to address this question is the following Leftover Hash Lemma. Before that we need definitions.

For any discrete probability distribution $P$ and $Q$ on a finite set $X$, the $\ell_1$-distance is defined as

$$||P - Q||_1 = \sum_{x \in X} |P_x - Q_x|,$$

where $P_x$ and $Q_x$ denote the probability of $x \in X$ under $P$ and $Q$ respectively. For any event $E \subseteq X$, the difference in probability $|P(E) - Q(E)| = |\sum_{x \in E} P_x - \sum_{x \in E} Q_x| \leq ||P - Q||_1$. In fact separating out those $P_x > Q_x$ from $P_x \leq Q_x$, it is easy to see that

$$||P - Q||_1 = 2 \max_E |P(E) - Q(E)|,$$

i.e., twice the maximum difference in probability. (This quantity $\max_E |P(E) - Q(E)|$ is called the *statistical distance* between $P$ and $Q$. However, some authors also call $||P - Q||_1$ their statistical distance.)

A distribution $P$ is caled $\epsilon$-uniform if $||P - U||_1 \leq \epsilon$, where $U$ is the uniform distribution on the same set (same support) as $P$.

Similarly we can define the $\ell_2$-distance

$$||P - Q||_2 = \sqrt{\sum_{x \in X} (P_x - Q_x)^2}.$$

The collision probability $c(P)$ is defined as

$$c(P) = \Pr_{x, x' \in X}[x = x'] = \sum_{x \in X} P_x^2,$$

where $x$ and $x'$ are independently taken according to $P$.

The next lemma is a simple application of the Cauchy-Schwarz inequality.

Lemma 5.26 *If $c(P) \leq \frac{1+\epsilon^2}{|X|}$, then $P$ on $X$ is $\epsilon$-uniform.*

*Proof.* By Cauchy-Schwarz,

$$||P-U||_1^2 \leq |X| \sum_{x \in X} (P_x - \frac{1}{|X|})^2 = |X| \left( c(P) - \frac{1}{|X|} \right) \leq \epsilon^2.$$

♣

In the next lemma we want to compute the collision probability of $s \circ h_s(y)$ when we have some restriction of $y$ (such as $y$ is taken from some (non)-witness set of some RP language.)

Lemma 5.27 *Let $s$ and $y$ be taken uniformly and independently from $\mathcal{S}$ and $Y' \subseteq Y$, and let $h_s$ be a hash function from $Y$ to $T$. Let $P$ be the probability distribution $s \circ h_s(y)$, and let $X = \mathcal{S} \times T$. Then*

$$c(P) = \frac{1 + \frac{|T|}{|Y'|}}{|X|}.$$

*Proof.* We simply compute, taking $s, s' \in \mathcal{S}$ and $y, y \in Y'$ uniformly and independently,

$$\begin{aligned} c(P) &= \Pr[(s = s') \wedge (h_s(y) = h_{s'}(y'))] \\ &= \frac{1}{|\mathcal{S}|} \Pr[h_s(y) = h_{s'}(y') | s = s'] \\ &= \frac{1}{|\mathcal{S}|} \left( \Pr[y = y'] + \Pr[y \neq y'] \cdot \Pr[h_s(y) = h_s(y') | y \neq y'] \right) \\ &\leq \frac{1}{|\mathcal{S}|} \left( \frac{1}{|Y'|} + 1 \cdot \frac{1}{|T|} \right) \\ &= \frac{1}{|X|} \cdot \left( 1 + \frac{|T|}{|Y'|} \right) \end{aligned}$$

♣

With the above notations, applying the previous two lemmas in sequence,

Lemma 5.28 (Leftover Hash Lemma) *The distribution $s \circ h_s(y)$ is $\epsilon$-uniform, where $\epsilon = \sqrt{|T|/|Y'|}$.*

Now we are ready to prove

Theorem 5.29 *For $x \in L$ so that $\mu(\overline{W_x}) \leq 1/2$,*

$$\Pr[G(s, Y_1, Z_1, \ldots, Z_k) \subseteq \overline{W_x}] \leq (\mu(\overline{W_x}))^{k+1} + 2^{-\ell/2}.$$

*Proof.* The intuitive idea of the proof is as follows. We first charge $\mu(\overline{W_x})$ for the first $Y_1 \in \overline{W_x}$. Then on the condition that $Y_1 \in \overline{W_x}$, the hashed value $h_s(Y_1)$ with appended uniform bits $Z_2$ makes a distribution quite close to uniform, and upon replacing $h_s(Y_1) \circ Z_2$ by $n$ true random bits $\hat{Y_2}$ we only pay a small price, and an inductive argument takes over.

Now we give the proof. Fix $\ell$, we induct on $k$. Let

$$e_k = \Pr[G(s, Y_1, Z_1, \ldots, Z_k) \subseteq \overline{W_x}],$$

where the probability is over all the unform and independent random choices of $s, Y_1, Z_1, \ldots, Z_k$. Clearly $e_0 = \mu(\overline{W_x})$. For $k \geq 1$,

$$e_k = \mu(\overline{W_x}) \Pr[G(s; Y_2, Z_2, \ldots, Z_k) \subseteq \overline{W_x} \mid Y_1 \in \overline{W_x}],$$

where $Y_2 = \langle h_s(Y_1) \circ Z_1 \rangle$. The probability is still over all $s, Y_1, Z_1, \ldots, Z_k$, except now we have the condition that $Y_1 \in \overline{W_x}$.

Thus we can consider $\overline{W_x}$ as our subset $Y'$ in Lemma 5.27, and estimate the statistical distance between the distributions

$$\langle s \circ Y_2 \circ Z_2 \circ \ldots \circ Z_k \rangle$$

and

$$\langle s \circ \hat{Y_2} \circ Z_2 \circ \ldots \circ Z_k \rangle$$

where $\hat{Y_2}$ is independent and uniform over $\{0,1\}^n$, the same length as $Y_2 = \langle h_s(Y_1) \circ Z_1 \rangle$. Let $\epsilon = \sqrt{2^{n-\ell}/(\mu(\overline{W_x})2^n)} = 1/\sqrt{2^\ell \mu(\overline{W_x})}$ as in Leftover Hash Lemma, then

$$\Pr[G(s, Y_2, Z_2, \ldots, Z_k) \subseteq \overline{W_x} \mid Y_1 \in \overline{W_x}] \leq \Pr[G(s, \hat{Y_2}, Z_2, \ldots, Z_k) \subseteq \overline{W_x}] + \epsilon/2,$$

as the probability over any event, such as the behavior under $G$, under these two distributions can only differ by at most $\epsilon/2$, half the $\ell_1$-norm distance.

It follows that,

$$e_k \leq \mu(\overline{W_x})(e_{k-1} + \epsilon/2).$$

We will inductively assume

$$e_{k-1} \leq (\mu(\overline{W_x}))^k + 2^{-(\ell+1)/2},$$

then we have

$$e_k \leq \mu(\overline{W_x}) \left( (\mu(\overline{W_x}))^k + 2^{-(\ell+1)/2} + \frac{1}{2\sqrt{2^\ell \mu(\overline{W_x})}} \right) \leq (\mu(\overline{W_x}))^{k+1} + 2^{-(\ell+1)/2}.$$

The last inequality follows from the fact that $\mu(\overline{W_x}) \leq 1/2$, and $\sqrt{\mu(\overline{W_x})/2^\ell} \leq 2^{-(\ell+1)/2}$.

♣

### 5.10.4 Expander Mixing Lemma

Let $G = (V, E)$ be a $d$-regular undirected graph with $n = |V|$ nodes. The adjacency matrix of $G$ is a symmetric 0-1 matrix defined as

$$M(i,j) = \begin{cases} 1 & \text{if } \{i,j\} \in E \\ 0 & \text{otherwise.} \end{cases}$$

Any real symmetric matrix has $n$ real eigenvalues $\lambda_0, \lambda_1, \ldots, \lambda_{n-1}$ (counting multiplicity) and $n$ corresponding orthogonal unit eigenvectors $v_0, v_1, \ldots, v_{n-1}$. Thus the inner product $\langle v_i, v_j \rangle = \delta_{ij}$, and $Mv_i = \lambda_i v_i$. It is clear that the all-1 vector is an eigenvector with eigenvalue $d$. Normalizing we let $v_0 = \frac{1}{\sqrt{n}}(1, 1, \ldots, 1)^T$, and $\lambda_0 = d$. Let

$$\lambda = \max_{1 \le i \le n-1} |\lambda_i|$$

denote the second largest eigenvalue in absolute value. The gap between $d$ and $\lambda$ controls the expansion property of $G$.

We can express every vector $v$ in terms of the eigenvectors $v = \sum_{0 \le i \le n-1} \alpha_i v_i$, then $Mv = \sum_{0 \le i \le n-1} \alpha_i \lambda_i v_i$.

For any two sets $A, B \subseteq V$, let $E(A, B)$ denote the set of ordered pairs of vertices which are also edges of $G$,

$$E(A, B) = \{(a, b) \mid a \in A, b \in B, \{a, b\} \in E\}.$$

What should the size of $E(A, B)$ be approximately? If we fix $A$, and randomly pick a subset $B$ of a certain cardinality $|B|$, then $|E(A, B)|$ is a sum of random variables $X_1 + \ldots + X_{|B|}$, where each $X_j$ is the number of ordered pairs of the form $(a, b_j)$, such that $\{a, b_j\} \in E$, where $a \in A$ and $b_j$ is the $j$ th element of the set $B$. These $X_j$ are not independent; but each $b_j$ is still uniformly distributed on $V$. Each $X_j$ can be further expressed as a 0-1 sum of $|A|$ many 0-1 random variables, $X_j = X_{1,j} + \ldots + X_{|A|,j}$, where each $X_{i,j}$ is the contribution of the $i$ th element $a_i$ of $A$ to $X_j$. Clearly then, the expectation $\mathbf{E}[X_{i,j}] = d/n$, and $|E(A, B)|$ has expectation $d|A||B|/n$.

The following lemma says $|E(A, B)|$ is close to this average, the extent to which this is close is controlled by $\lambda$.

Lemma 5.30 (Expander Mixing Lemma) *For all $A, B \subseteq V$,*

$$\left| |E(A, B)| - \frac{d|A||B|}{n} \right| \le \lambda \sqrt{|A||B|} \le \lambda n.$$

*Proof.* Express the chartecteristic functions $\chi_A$ and $\chi_B$ in terms of the eigenvectors,

$$\chi_A = \sum_{0 \le i \le n-1} \alpha_i v_i, \quad \text{and} \quad \chi_B = \sum_{0 \le i \le n-1} \beta_i v_i,$$

where $\alpha_i = \langle \chi_A, v_i \rangle$ and $\beta_i = \langle \chi_B, v_i \rangle$. Clearly $\alpha_0 = |A|/n$ and $\beta_0 = |B|/n$.

Our proof tries to separate out the "main term" and the rest in the sum

$$\begin{aligned}
|E(A,B)| &= \sum_{i \in A, j \in B} M(i,j) \\
&= \chi_A^T M \chi_B \\
&= \left( \sum_{0 \le i \le n-1} \alpha_i v_i \right) \cdot \left( \sum_{0 \le i \le n-1} \beta_i \lambda_i v_i \right) \\
&= \sum_{0 \le i \le n-1} \lambda_i \alpha_i \beta_i \\
&= \frac{d|A||B|}{n} + \sum_{1 \le i \le n-1} \lambda_i \alpha_i \beta_i
\end{aligned}$$

Thus,

$$\begin{aligned}
\left| |E(A,B)| - \frac{d|A||B|}{n} \right| &\le \lambda \sum_{1 \le i \le n-1} |\alpha_i \beta_i| \\
&\le \lambda ||\alpha||_2 ||\beta||_2 \\
&= \lambda ||\chi_A||_2 ||\chi_B||_2 \\
&= \lambda \sqrt{|A||B|} \\
&\le \lambda n
\end{aligned}$$

♣

There is an alternative way to state this result that brings home the similarity to the Hash Mixing Lemma. Consider either (1) taking a uniform $x \in V$ and then taking a uniform neighbor $e_s(x)$ (where $s$ picks one of $d$ values uniformly), or (2) taking two independent and uniform points $x$ and $y$ in $V$. Then for all $A, B \subseteq V$,

$$|\Pr_{x,s}[x \in A, e_s(x) \in B] - \Pr_{x,y}[x \in A, y \in B]| \le \lambda/d.$$

Note that the first term is $\frac{|E(A,B)|}{nd}$, and the second term is $\frac{|A||B|}{n^2}$, then

$$\left| \frac{|E(A,B)|}{nd} - \frac{|A||B|}{n^2} \right| = \frac{1}{nd} \left| |E(A,B)| - \frac{d|A||B|}{n} \right| \le \frac{\lambda}{d}.$$

Expander graphs are those with a large gap between $\lambda$ and $d = \lambda_0$. One can explicitly construct expander graph families with $n \to \infty$, and with constant $d$, and $\lambda \le 2\sqrt{d}$.

For all $n'$ and $d'$, one can construct expander graphs with $n' \le n \le 2n'$ and $d' \le d \le 2d'$, and achieves $\lambda \le d^{9/10}$.

A simple application is the following Karp-Pippenger-Sipser generator. Consider an expander graph with vertex set $\{0,1\}^n$, and degree $d$. Assume $\lambda \leq d^{9/10}$. With "seed" $z \in \{0,1\}^n$, $G(z)$ produces all $d$ neighbors of $z$: $G(z) = \{z_1, \ldots, z_d\}$.

THEOREM 5.31 *For $x \in L$ so that $\mu(\overline{W_x}) \leq 1/2$,*

$$\Pr[G(z) \subseteq \overline{W_x}] \leq 2\left(\frac{\lambda}{d}\right)^2.$$

*Proof.* Let $A$ be the set of vertices all of whose neighbors are in $\overline{W_x}$. Let $B = W_x$. Then $E(A,B) = \emptyset$. By Expander Mixing Lemma,

$$\left|0 - \frac{d|A||W_x|}{2^n}\right| \leq \lambda\sqrt{|A||W_x|}.$$

Since $|W_x| \geq 2^n/2$, we get

$$\frac{|A|}{2^n} \leq 2\left(\frac{\lambda}{d}\right)^2.$$

♣

### 5.10.5 Ajtai-Komlós-Szemerédi Generator

The idea of using expander graphs can be significantly furthered. The intuitive idea is that, after the initial random vertex, one can take a random walk on the expander graph. If the graph has vertex set $\{0,1\}^n$, and constant degree $d$, then each additional step of the random walk takes only $O(1)$ random bits. So if we take $k$ steps, we will use only $n+O(k)$ bits. In the meanwhile the names of the vertices the random walk visited form $k$ blocks of $n$-bits each. Conditional on the previous vertices being in (or not in) the witness set $W_x \subseteq \{0,1\}^n$, the properties of the expander graph ensure that the random walk tends to "disperse" quickly, and therefore these names of the vertices on the random walk in a way simulate true random samples.

Consider an expander graph with vertex set $\{0,1\}^n$, and constant degree $d$. We will assume $\lambda/d \leq 1/4$; this can be constructed. Take the seed bits $z \in \{0,1\}^n$, and $s_1, s_2, \ldots, s_k \in \{1, \ldots, d\}$. We take $y_0 = z$, and for $i \geq 1$, $y_i$ is the neighbor of $y_{i-1}$ indexed by $s_i$. Then

$$G(z; s_1, s_2, \ldots, s_k) = \{y_0, y_1, \ldots, y_k\}.$$

THEOREM 5.32 *For $x \in L$ so that $\mu(\overline{W_x}) \leq 1/2$,*

$$\Pr[G(z; s_1, s_2, \ldots, s_k) \subseteq \overline{W_x}] \leq 2^{\Theta(k)}.$$

*Proof.* The proof idea is to express the probability distributions after $i$ steps of the random walk in the spectra (eigenvalues and eigenvectors) of the expander graph, and analyze the probability (expressed in $\ell_1$-norm) in terms of $\ell_2$-norm.

Write $N = 2^n$. Choosing uniformly $z \in \{0,1\}^n$ gives the uniform distribution, which can be represented by a vector in $\mathbf{R}^N$ as $p_0 = (1/N, \ldots, 1/N)^T$. Let $\hat{M} = \frac{1}{N}M$ be the probability transition matrix of the Markov chain defined by the random walk, where $M$ is the adjacency matrix of $G$. If $p$ is a distribution on $G$, then after one step of the random walk, the distribution is $\hat{M}p$. Define $P = P_{\overline{W_x}}$ to be the projection matrix, which has a 1 in every diagonal entry corresponding to an element of $\overline{W_x}$, and 0 elsewhere else. Then

$$\Pr[y_1 \in \overline{W_x}] = ||P\hat{M}p_0||_1.$$

Denote by $w_i = \Pr[\{y_1, \ldots, y_i\} \subseteq \overline{W_x}]$. We show by induction that

$$w_k = ||(P\hat{M})^k p_0||_1.$$

Inductive Hypothesis: We assume the distribution of the $i$ th step $y_i$, conditional on $\{y_1, \ldots, y_{i-1}\} \subseteq \overline{W_x}$, is given by the vector

$$\frac{1}{w_{i-1}}\hat{M}(P\hat{M})^{i-1}p_0.$$

and

$$w_{i-1} = ||(P\hat{M})^{i-1}p_0||_1.$$

Then clearly

$$\Pr[\{y_i \in \overline{W_x} \mid \{y_1, \ldots, y_{i-1}\} \subseteq \overline{W_x}] = \frac{1}{w_{i-1}}||(P\hat{M})^i p_0||_1,$$

and

$$\begin{aligned} w_i &= \Pr[\{y_1, \ldots, y_i\} \subseteq \overline{W_x}] \\ &= \Pr[\{y_1, \ldots, y_{i-1}\} \subseteq \overline{W_x}] \cdot \Pr[\{y_i \in \overline{W_x} \mid \{y_1, \ldots, y_{i-1}\} \subseteq \overline{W_x}] \\ &= w_{i-1} \cdot \frac{1}{w_{i-1}}||(P\hat{M})^i p_0||_1 \\ &= ||(P\hat{M})^i p_0||_1. \end{aligned}$$

We can also write

$$\Pr[\{y_i \in \overline{W_x} \mid \{y_1, \ldots, y_{i-1}\} \subseteq \overline{W_x}] = \frac{w_i}{w_{i-1}}.$$

The distribution of $y_{i+1}$, conditional on $\{y_1, \ldots, y_i\} \subseteq \overline{W_x}$, can be represented by the vector

$$\frac{1}{w_i/w_{i-1}}\hat{M}P\left(\frac{1}{w_{i-1}}\hat{M}(P\hat{M})^{i-1}p_0\right) = \frac{1}{w_i}\hat{M}(P\hat{M})^i p_0.$$

This completes the induction.

(We can also show $\Pr[\{y_0, y_1, \ldots, y_k\} \subseteq \overline{W_x}] = ||(P\hat{M})^k \hat{M} p_0||_1$. But this only contributes another factor of 1/2.)

Now write any vector $v$ in terms of the eigenvectors $v_i$, $v = \sum_{0 \le i \le n-1} \alpha_i v_i$, then

$$\begin{aligned} ||P\hat{M}v||_2 &\le ||P\hat{M}\alpha_0 v_0||_2 + ||P\hat{M} \sum_{1 \le i \le n-1} \alpha_i v_i||_2 \\ &\le ||P\alpha_0 v_0||_2 + ||\hat{M} \sum_{1 \le i \le n-1} \alpha_i v_i||_2. \end{aligned}$$

Here we used the fact that $\hat{M}v_0 = v_0$, and the orthogonal projection matrix $P$ has $\ell_2$-norm $||P|| \le 1$. More concretely, for any vector $u$, $||Pu||_2^2 = \sum_{i \in \overline{W_x}} u_i^2 \le \sum_{1 \le i \le N} u_i^2 = ||u||_2^2$.

Recall $v_0 = \frac{1}{N}(1, \ldots, 1)^T$. Note that $||Pv_0||_2^2 = \frac{|\overline{W_x}|}{N} = \mu(\overline{W_x})$, and $\hat{M}v_i = (\lambda_i/d)v_i$, it follows that

$$\begin{aligned} ||P\hat{M}v||_2 &\le \sqrt{\mu(\overline{W_x})}|\alpha_0| + ||\sum_{1 \le i \le n-1} (\lambda_i/d)\alpha_i v_i||_2 \\ &= \sqrt{\mu(\overline{W_x})}|\alpha_0| + \sqrt{\sum_{1 \le i \le n-1} (\lambda_i/d)^2 |\alpha_i|^2} \\ &\le \sqrt{\mu(\overline{W_x})}||\alpha_0 v_0||_2 + (\lambda/d)||\sum_{1 \le i \le n-1} \alpha_i v_i||_2 \\ &\le \left(\sqrt{\mu(\overline{W_x})} + \frac{\lambda}{d}\right) ||v||_2, \end{aligned}$$

where the last inequality is because both $\alpha_0 v_0$ and $\sum_{1 \le i \le n-1} \alpha_i v_i$ are orthogonal projections of $v = \sum_{0 \le i \le n-1} \alpha_i v_i$.

Finally, by Cauchy-Schwarz,

$$\begin{aligned} w_k &= \Pr[\{y_1, \ldots, y_k\} \subseteq \overline{W_x}] \\ &= ||(P\hat{M})^k p_0||_1 \\ &\le \sqrt{N}||(P\hat{M})^k p_0||_2 \\ &\le \sqrt{N}\left(\sqrt{\mu(\overline{W_x})} + \frac{\lambda}{d}\right)^k ||p_0||_2 \\ &\le \sqrt{N}\left(\sqrt{1/2} + 1/4\right)^k \frac{1}{N} \\ &= 2^{-\Theta(k)} \end{aligned}$$

In order to achieve error probability $2^{-k}$, the AKS generator uses $n + O(k)$ true random bits.

# Chapter 6

# Hartmanis Conjectures

In this lecture, we discuss Berman-Hartmanis isomorphism conjecture.

## 6.1 Introduction

The Berman-Hartmanis conjecture, also known as the *isomorphism conjecture* is related to NP vs. P question. We start with a definition. Two languages $A$ and $B$ are said to be p-isomorphic (polynomial time isomorphic), if there is a polynomial computable, polynomial time invertible, 1-1 and onto reduction from $A$ to $B$. Formally, we need a function $f : \Sigma^* \mapsto \Sigma^*$, such that i) $f$ is 1-1 and onto, ii) $x \in A \iff f(x) \in B$, iii) $f$ polynomial time computable, iv) $f^{-1}$ is polynomial time computable. Berman and Hartmanis observed that all "known" NP-Complete problems are p-isomorphic to each other. Here, "known" refers to, for example, all problems in Garey and Johnson. We will get into to the issue of proving this claim, shortly. Based on this claim, they formulated the now-famous conjecture:

**Berman-Hartmanis isomorphism conjecture:** Any two NP-Complete sets are p-isomorphic to each other.
The most interesting aspect of the conjecture is that, if it is true then NP $\neq$ P. Because, if NP = P, then, even finite sets would be NP-Complete. But, a finite set cannot be isomorphic to a infinite set like SAT. The conjecture has been studied extensively in the past two decades. If the conjecture is true, then not even sparse sets can be NP-Complete, because SAT is an "exponentially" dense set and there cannot be a polynomial time isomorphism from a dense set to a sparse set. This raises an interesting issue of whether sparse sets can be NP-Complete. Mahaney's theorem (which we proved in an earlier lecture) shows that no sparse set can be NP-Complete, unless NP=P. The isomorphism conjecture remains open. Some recent evidence shows that it may be false. We will discuss this evidence at the end of the lecture.

We now return to the claim made by Berman and Hartmanis: all “known” NP-Complete problems are p-isomorphic to each other. The claim is based on their following theorem. Suppose there are polynomial time computable, polynomial time invertible, 1-1 and length increasing reductions from $A$ to $B$ and $B$ to $A$. Then, $A$ and $B$ are p-isomorphic to each other. (We will state the theorem more formally later.) They observe that such reductions exist for all “known” NP-Complete problems. Proof of the above theorem is based on a theorem due to Cantor. We prove Cantor's theorem first, and then continue our discussion of the isomorphism conjecture.

## 6.2 A Theorem of Cantor

Theorem 6.1 *If $A$ and $B$ are sets such that there exists a $1-1$ map from $A$ to $B$ and a $1-1$ map from $B$ to $A$, then there exists a $1-1$ correspondence (i.e., a map that is both $1-1$ and onto) between $A$ and $B$*

*Proof.*

Let $f$ be a $1-1$ map from $A$ to $B$, and $g$ be a $1-1$ map from $B$ to $A$.

If either $f$ or $g$ is also onto, then it is a $1-1$ correspondence. We will henceforth assume that neither $f$ nor $g$ is onto and prove that there still exists a $1-1$ correspondence.

Consider the following two sequences of subsets of $A$ and $B$: $A_0 = A$, $B_0 = B$, and the rest of the $A_i$ and $B_i$ are given by:

$$A_i = g(B_{i-1}) \qquad B_i = f(A_{i-1})$$

These two sequences are shown pictorially in figure 6.1.

The first thing to note about this sequence is that $A_i \subset A_{i-1}$ (i.e., $A_i$ is a proper subset of $A_{i-1}$) for all $i \in \mathbf{N}$, and the same is true for the $B_i$. To see this, observe first that it is clear from the definitions of $A_i$ and $B_i$ that $A_i \subseteq A_{i-1}$ and $B_i \subseteq B_{i-1}$. We will show (by induction on $i$) that successive sets are not equal.

Observe that $A_1 = g(B_0)$. Since $g$ is not onto, there exists an element in $A_1$ that is not in $A_0$. Similarly, $B_1 \subset B_0$.

Now, assuming that $B_i \subset B_{i-1}$, let $x$ be any element of $B_{i-1} - B_i$. Note first that $g(x) \in A_i$, since $A_i = g(B_{i-1})$. We claim that $g(x) \notin A_{i+1}$. If $g(x) \in A_{i+1}$, then $\exists y \in B_i$ such that $g(y) = g(x)$. Since $x \notin B_i$, this would imply that two distinct elements ($x$ and $y$) of $B$ are mapped to the same element of $A$, contradicting the stipulation that $g$ is $1-1$. This completes the induction and proves that $A_i \subset A_{i-1}$ for all $i \in \mathbf{N}$. A similar argument can be made for the $B_i$.

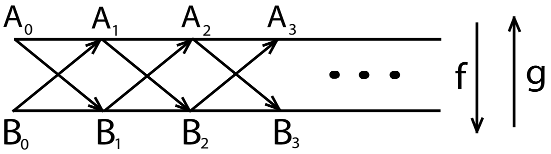


Figure 6.1: A pictorial representation of the sequences created by applying the functions $f(x)$ and $g(x)$. The $A_i$'s are listed by cardinality with the largest on the left. The same is true of the $B_i$'s.

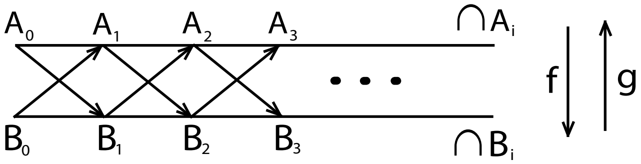


Figure 6.2: The pictorial description from 6.1 modified to include the sets $\bigcap_{i=0}^{\infty} A_i$ and $\bigcap_{i=0}^{\infty} B_i$. Again, the largest sets are on the left of the image while the smallest (i.e. $\bigcap_{i=0}^{\infty} A_i$ and $\bigcap_{i=0}^{\infty} B_i$) are on the right.

Now, with these facts in hand, we note that the set $A$ can be decomposed as follows:

$$A = (A_0 - A_1) \cup (A_1 - A_2) \cup \cdots \cup (A_0 \cap A_1 \cap A_2 \cap \cdots)$$

$$= \left( \bigcup_{i=0}^{\infty} (A_i - A_{i+1}) \right) \cup \left( \bigcap_{i=0}^{\infty} A_i \right)$$

Similarly,

$$B = \left( \bigcup_{i=0}^{\infty} (B_i - B_{i+1}) \right) \cup \left( \bigcap_{i=0}^{\infty} B_i \right)$$

Thus, for any element $x \in A$, either $x \in (A_i - A_{i+1})$ for some (unique) $i$, or $x \in A_i$ for all $i$ (see figure 6.2). We'll define a function $F : A \to B$, which will be a $1-1$ correspondence, as follows: if $x \in \bigcap_{i=0}^{\infty} A_i$, then $F(x) = f(x)$. Otherwise, $x \in (A_i - A_{i+1})$ for some $i$. In this case,

$$F(x) = \begin{cases} f(x) & \text{if } i \text{ is even} \\ g^{-1}(x) & \text{if } i \text{ is odd} \end{cases}$$

That is, an element $x$ of $A$ is mapped to $B$ by either $f$ or $g^{-1}$, depending on which subset(s) of $A$ it falls into. Note that for all $i > 0$, $A_i$ contains only elements that are in the image of $g$, so $g^{-1}(x)$ is well-defined in all cases where we use it. It remains for us to show that $F$ is $1-1$ and onto.

We start with $1-1$. We first consider an element $x \in \bigcap_{i=0}^{\infty} A_i$ and argue that no other element of $A$ maps to $F(x)$ under $F$. Here we have $F(x) = f(x)$. Moreover, note that $f(x) \in \bigcap_{i=0}^{\infty} B_i$. Consider any element $x' \in A$ with $x' \neq x$. If $x' \in \bigcap_{i=0}^{\infty} A_i$, $F(x') = f(x')$ and as $f$ is 1-1, $F(x) \neq F(x')$. If $x' \notin \bigcap_{i=0}^{\infty} A_i$, then $x'$ is mapped to some element in $B_i - B_{i-1}$, for some $i$. Hence, such an $x'$ cannot be mapped to an element in $\bigcap_{i=0}^{\infty} B_i$. So, $F(x) \neq F(x')$. Next consider an element $x \in (A_i - A_{i+1})$ for some $i$. We have two cases: if $i$ is even, then $f(x) \in (B_{i+1} - B_{i+2})$ – i.e., the largest $j$ for which $f(x) \in B_j$ is odd; if $i$ is odd, then $g^{-1}(x) \in B_{i-1} - B_i$ – i.e., the largest $j$ for which $g^{-1}(x) \in B_j$ is even. Thus, an element of $A$ mapped by $f$ never collides with an element mapped by $g^{-1}$. Since $f$ and $g$ are both $1-1$, elements mapped by $f$ never collide with each other, and likewise for $g^{-1}$. Thus, $F$ is $1-1$.

It may be obvious at this point that $F$ is also onto, but we will prove it for good measure. Let $y$ be an element of $B$. Suppose $y \in \bigcap_{i=0}^{\infty} B_i$. Clearly, $y \in B_1$ and hence, for some $x \in A$, $f(x) = y$. For any $i$, notice that $y \in B_i = f(A_{i-1})$ and $f$ is 1-1 and hence, $x \in A_{i-1}$. Thus, $x \in \bigcap_{i=0}^{\infty} A_i$, and hence, $F(x) = f(x) = y$. Suppose, $y \in B_i - B_{i+1}$ for some $i$, then we again look at two cases: if $i$ is even, then $y$ is mapped to by $g(y)$; if $i$ is odd, then $i$ is mapped to by $f^{-1}(y)$, which is guaranteed to exist, since every odd $B_i$ is the image under $f$ of some $A_j$. Thus, $F$ is $1-1$ and onto.

## 6.3 Myhill's Theorem

One interesting theorem that can be proven using a similar proof is Myhill's theorem.

Theorem 6.2 *Every* RE-*complete set has a 1-1 onto recursive, invertible map to every other* RE-*complete set.*

This theorem shows that essentially, there is only "one" RE-complete set. In other words, any two undecidable, r.e. sets are just computable renaming of each other. We will not prove the theorem in this class.

## 6.4 The Berman-Hartmanis Conjecture

Berman and Hartmanis conjecture that all NP -Complete problems are p-isomorphic to each other. The conjecture is based on the following theorem.

Theorem 6.3 *Let $L_1$ and $L_2$ be two sets such that there are functions $f : \Sigma^* \mapsto \Sigma^*$ and $g : \Sigma^* \mapsto \Sigma^*$ that satisfy*

1. *$f$ and $g$ are polynomial time computable.*

*2. $f$ and $g$ are 1-1.*

*3. $f$ and $g$ are length increasing. That is, for any $x$, $|f(x)| > |x|$ and $|g(x)| > |x|$.*

*4. $f$ is a reduction from $L_1$ to $L_2$ and $g$ is a reduction from $L_2$ to $L_1$.*

*5. $f$ and $g$ are polynomial time invertible. That is, there is a polynomial time algorithm that given $x$ either outputs the (unique) $y$ such that $f(y) = x$ or says no such $y$ exists. Similarly, such an algorithm exists to compute $g^{-1}$.*

*Then, $L_1$ and $L_2$ are p-isomorphic. That is, there is a polynomial time computable, polynomial time invertible reduction $F$ from $L_1$ to $L_2$ which is 1-1 and onto.*

*Proof.* We follow the proof of Cantor's theorem. Let $A_0 = B_0 = \Sigma^*$. Then, as in Cantor's proof, define

$$A_i = g(B_{i-1}) \text{ and } B_i = f(A_{i-1})$$

First of all notice that, a string $x$ of length $n$ cannot be in $A_{n+1}$, because both $f$ and $g$ are (strictly) length increasing. Thus, $\bigcap_{i=0}^{\infty} A_i$ and $\bigcap_{i=0}^{\infty} B_i$ are empty. Let $x$ be any string and $i$ be the (unique) integer such that $x \in A_i - A_{i+1}$. Because, $\bigcap_{i=0}^{\infty} A_i$ is empty, $i$ is well-defined. Now let,

$$F(x) = \begin{cases} f(x) & \text{if } i \text{ is even} \\ g^{-1}(x) & \text{if } i \text{ is odd} \end{cases}$$

That completes the definition of $F$. We need to argue that $F$ satisfies the required properties:

1. *$F$ is 1-1 and onto:* The proof is same as Cantor's proof.

2. *$F$ is a reduction from $L_1$ to $L_2$ :* For any $x$, $F(x)$ is either $f(x)$ or $g^{-1}(x)$. The claim follows from the fact that $f$ is a reduction from $L_1$ to $L_2$ and $g$ is a reduction from $L_2$ to $L_1$.

3. *$F$ is polynomial time computable :* Given an $x$, all we need to do is, determine whether we are in the even or odd case. After that, we need to either compute $f(x)$ or $g^{-1}(x)$ and both these can be done in polynomial time. Notice that $x \in A_1$ iff $g^{-1}(x)$ exists. Similarly, $x \in A_2$ iff $f^{-1}(g^{-1}(x))$ exists and $x \in A_3$ iff $g^{-1}(f^{-1}(g^{-1}(x)))$ exists. In general, to determine whether $x \in A_k$, we need simply apply $g^{-1}$ and $f^{-1}$ alternatively $k$ times and check whether that element exists or not. Now our goal is to find the unique $d$ such that $x \in A_d - A_{d+1}$. We already noted that $x \notin A_n$, where $n = |x|$. Hence, $d \leq n$. Recall our assumption that $f^{-1}$ and $g^{-1}$ are polynomial time computable. So, the idea is to start with $x$ and apply $g^{-1}$ and $f^{-1}$ alternatively: compute

$$x_0 = x, x_1 = g^{-1}(x), x_2 = f^{-1}(x_1), x_3 = g^{-1}(x_2), x_4 = f^{-1}(x) \ldots$$

Notice this, the strings in this sequence get shorter as we move on (i.e. $|x_{i+1}| < |x_i|$), because $f$ and $g$ are length increasing. After some steps, we will have to stop, becuase either $f^{-1}$ (or $g^{-1}$) does not exist or we end up with a string of length 1. Suppose we computed till $x_i$ succesfully and $x_{i+1}$ does not exist. Then, $x \in A_i$ but $x \notin A_{i+1}$, (i.e. $d = i$). As $f^{-1}$ and $g^{-1}$ are computable each step of this process can be carried out in polynomial time. As $d \leq n$, we will need at most $n$ steps.

Using this theorem, Berman and Hartmanis observed that any two "known" NP-Complete sets are p-isomorphic to each other. Because, for any "known" NP-Complete problem, the NP-Completeness reduction can easily be modified to make it 1-1, length increasing and polynomial time invertibe! As examples, we will consider SAT and Hamiltonican circuit.

Let $L$ be any lnaguage in NP and $f$ be the reduction from $L$ to SAT. We will construct a new reduction $f'$ from $L$ to SAT which is length increasing and 1-1. Let $x = a_1 a_2 \ldots a_n \in \{0,1\}^*$ be the input string of length $n$. Reduction $f'$ first computes $\varphi = f(x)$. Then construct a new formula $\varphi'$. $\varphi'$ uses all the boolean variables of $\varphi$ and $n$ new variables $z_1, z_2, \ldots, z_n$. New formula $\varphi' = \varphi \wedge (l_1 \vee l_2 \vee \ldots \vee l_n)$, where the lieteral $l_i$ is $z_i$, if $a_i = 1$ and $\overline{z_i}$ if $a_i = 0$. It is clear that $\varphi$ is satisfiable iff $\varphi'$ is satisfiable and hence, $f'$ is a reduction from $L$ to SAT. Moreover, $|\varphi'| > |x|$. Finally, given $\varphi'$, we can easily extract the string $x$. Thus, $f'$ is polynomial time invertible.

A similar trick can be used in case of Hamiltonican circuit (HC). Let $L$ be any language in NP and $f$ be a reduction from $L$ to HC. We construct a new reduction $f'$. Given input $x = a_1 a_2 \ldots a_n$, $f'$ first computes the graph $G = f(x)$. $f'$ will output a new graph $G'$. $G'$ is obtained by first adding $3n + 1$ new vertices $s_1, s_2, \ldots, s_{n+1}$, $u_1, u_2, \ldots, u_n$, and $v_1, v_2, \ldots, v_n$ to $G$. For each $1 \leq i \leq n$, if $a_i = 0$ then add the edges $s_i \rightarrow u_i \rightarrow v_i \rightarrow s_{i+1}$; if $a_i = 0$ then add the edges $s_i \rightarrow v_i \rightarrow u_i \rightarrow s_{i+1}$. Thus, we have created some "chain" of vertices, that depend on input $x$. Finally, take any edge $(s, t)$ in $G$ and "replace" it with the chain constructed from $x$. Namely, remove the egde $(s, t)$ and add edges $(x, s_1)$ and $(s_{n+1}, t)$. It is clear that $G$ has a Hamiltonican circuit iff $G'$ has one. Moreover, $|G'| > |x|$ and given $G'$, we can extract $x$ easily.

For all "known" NP-Complete problems, say those listed in Garey and Johnson, we can do such simple tricks to make the reduction length increasing and 1-1. Then, the above theorem implies that all known NP-Complete problems are p-isomorphic to each other. Based on this evidence, Berman and Hartmanis conjectured that all NP-Complete problems are p-isomorphic to each other. But, now a days, it is widely believed to be fasle. The Joseph-Young conjecture gives some evidence for this belief.

## 6.5 Joseph-Young Conjecture

The Joseph and Young conjecture states that there is an NP-Complete languages $A$ such that there is no polynomial time invertible reduction from SAT to $A$. Such a language would fail to satisfy the requirements of Theorem 6.3. The language $A$ is constructed based on the assumption that one-way functions exist. Loosely speaking, a function is one-way, if it is computable in polynomial time, but not invertible in polynomial time. It is not known whether one-way functions exist (existence of one-way functions imply NP $\neq$ P). But, there are some candidate functions believed to be one-way. One example is multiplication. Given two numbers $a$ and $b$, it is easy to compute their product $ab$. But the inverse of multiplication, namely, factoring is believed to be hard. Suppose we have a one-way function $f$. It is easy to show that for any 1-1 function $g$, the language $g$(SAT) is NP-Complete. In particular, $f(SAT)$ is NP-Complete. Now, the conjecture is that there is no invertible reduction from SAT to $f(SAT)$.

# Chapter 7

# Interactive Proof Systems

In Chapter 1, we defined a prover-verifier model for languages in NP. Let us take SAT as an example language in NP. Suppose the prover wants to prove that a given formula is satisfiable. He can do so by giving a satisfying truth assignment as a certificate. The verify can verify in polynomial time whether the assignmnet indeed satisifies the formula. Suppose the formula is actually unsasfiable. How can the prover provide a concise certificate that the formula is unsasfiable? In other words, is there a proof system for unsatisfiability? Next two chapters develop proof systems for languages beyond NP. In these proof systems, the prover and verifier have many rounds of interaction. We start with an example for an Interactive Proof system (**IP**).

## 7.1 Interactive Proofs – An Example

We explain IP with an example. We say that two graphs $G = (V, E)$ and $H = (V', E')$ are isomorphic, if relabeling $G$ would give $H$. Formally, $G$ and $H$ are isomorphic, if there is a one-one and onto function $\sigma : V \to V'$, such that $(u, v) \in E$ iff $(\sigma(u), \sigma(v)) \in E'$. If $G$ and $H$ are isomorphic, we write it as $G \cong H$. For a permutation $\sigma$ on $V$, we denote $\sigma(G) = (V, E_\sigma)$, where, $E_\sigma = \{(\sigma(u), \sigma(v))|(u, v) \in E\}$. Define the following two languages:-

$$GI = \{(G, H)|G \cong H\}$$

$$GNI = \{(G, H)|G \ncong H\}$$

It's easy to see that GI is in NP. We can simply guess a $\sigma$ and verify whether $\sigma(G) = H$. What about GNI? It's in co-NP, by definition. But what about a proof system for GNI? No concise proof for GNI is known. Here, we describe what's called an interactive proof for GNI.

An Interactive Proof system (IP) for $L$ consists of two players, prover and the verifier. The prover claims that a string $x$ is in $L$ and the verifier has to verify it. The prover has no computational limitations, he is all powerful, whereas, the verifier is a probabilistic polynomial time player. They can interact with each other. The verifier can toss some coins and ask the prover some question and prover answers it. This process of interaction can go on for some polynomial number of rounds. At the end, the verifier accepts or rejects the input. If $x \in L$, then we require that the verifier accept with probability $\geq 3/4$ and if $x \notin L$, then he accepts with probability $\leq 1/4$. As usual, the values 3/4 and 1/4 are arbitrary and can be amplified.

We now describe an IP for GNI. So, let $G$ and $H$ be the two input graphs and the prover claims that $G \ncong H$. The verifier tosses a coin. If HEAD, he chooses $G$ else chooses $H$. Let the chosen graph be $S$. Then, he chooses a random permutation $\sigma$ and sends $S_\sigma = \sigma(S)$ to the prover. The prover has to send the original coin toss of the verifier. If $G$ and $H$ were really non-isomorphic, then, $S_\sigma$ will be isomorphic to exactly one of $G$ or $H$ and the prover can find it easily and correctly guess the coin toss. On the other hand, if he were cheating and $G \cong H$, $S_\sigma$ be isomorphic to both $G$ and $H$. So the prover, can at most make a wild guess at what the coin toss was. He will have probability only 1/2 to succeed. We can run this process a polynomial number of times to make the probability of cheating to be exponentially small.

It's easy to see that $NP \subseteq IP$. The verifier keeps quiet! On input $x$, the prover sends him the certificate that $x \in L$. The verifier verifies whether it's a valid certificate.

## 7.2 Arthur-Merlin Games

Arthur-Merlin games are restricted versions of IP. In AM we allow only one round of interaction betwen the prover and verifier. Here we call the prover as Merlin and the verfier as Arthur. Merlin is an all powerful magician, whereas, Arthur is a humble probabilistic polynomial time monarch! Merlin wants to prove that a string $x$ is in $L$. As in IP, he does this by interacting with Arthur.

**Definition (AM):** $L \in AM$ iff $\exists$ a polynomial time predicate $P$ such that

$$x \in L \Rightarrow Pr_y[\exists z, P(x,y,z) = 1] \geq \frac{3}{4}$$

$$x \notin L \Rightarrow Pr_y[\exists z, P(x,y,z) = 1] \leq \frac{1}{4}$$

The strings $y$ and $z$ are polynomially bounded in length of $x$.

**Remarks:** We can see that $AM \subseteq IP$. AM is a restricted version of IP. Arthur simply tosses coins and chooses a random string $y$ and sends it to Merlin. Merlin sends back a certificate $z$. Arthur, evaluates the polynomial predicate $P(x,y,z)$. If $x \in L$, for most coin tosses $y$ of Arthur, Merlin will have some valid proof $z$. On the other hand, if $x \notin L$, for most $y$, Merlin will not have a valid proof $z$.

As usual, we can amplify the success probabilities. Arthur, chooses many random strings, $y_1, y_2, \dots y_r$ and sends them all to Merlin. Merlin replies back with $z_1, z_2, \dots z_r$. The predicate $P$ is modified to $P'$ which would simulate $P(x, y_1, z_1), \dots P(x, y_r, z_r)$. Then take the majority. We can use Chernoff bounds to show that the success probability is amplified exponentially in $r$. Thus, we can replace $\frac{3}{4}$ by $1 - \frac{1}{2^n}$, and $\frac{1}{4}$ by $\frac{1}{2^{poly(n)}}$.

Now suppose we have a university president who wants no error. He wants probabilities 1 and 0, respectively. For example, if $x \notin L$, then $\forall y \forall z$ it is false. Indeed, we can consider, four possibilities here.

$$(1.)\ \begin{cases} x \in L & \text{then for most } y\text{, there exists } z \text{ that makes it true} \\ x \notin L & \text{then for most } y\text{, for all } z \text{ it is untrue} \end{cases}$$

$$(2.)\ \begin{cases} x \in L & \text{then for all } y\text{, there exists } z \text{ that makes it true} \\ x \notin L & \text{then for most } y\text{, for all } z \text{ it is untrue} \end{cases}$$

$$(3.)\ \begin{cases} x \in L & \text{then for all } y\text{, there exists } z \text{ that makes it true} \\ x \notin L & \text{then for all } y\text{, for all } z \text{ it is untrue} \end{cases}$$

$$(4.)\ \begin{cases} x \in L & \text{then for most } y\text{, there exists } z \text{ that makes it true} \\ x \notin L & \text{then for all } y\text{, for all } z \text{ it is untrue} \end{cases}$$

Statement (1.) is simply the definition of AM. Statements (3.) and (4.) are both cases which fall into NP. Statement (2) defines, what we call $AM_1$, one sided error version of $AM$. We discuss it shortly.

We note that we can always transform $\exists z_{|z| \leq n^{O(1)}} P(x, y, z) = 1$ to a satisfiability issue by using Cook's theorem. Therefore, the general form of AM is:

For $x \in L$, then for most $y$, we get $\varphi_{x,y} \in$ SAT.

For $x \notin L$, then for most $y$, we get $\varphi_{x,y} \notin$ SAT.
where the map $x, y \mapsto \varphi_{x,y}(z)$ is computable in polynomial time.

We can think of AM as follows: We think of AM by writing AM = BP $\cdot$ $NP$.

We define AM$_1$ to be the statement of (2.). Formally,

**Definition ($AM_1$)**: $L \in AM_1$ iff $\exists$ a polynomial time predicate $P$ such that

$$x \in L \Rightarrow Pr_y[\exists z, P(x, y, z) = 1] = 1$$

$$x \notin L \Rightarrow Pr_y[\exists z, P(x, y, z) = 1] \leq \frac{1}{4}$$

$AM_1$ can be thought of as $co{-}RP.NP$. It is immediate that AM$_1 \subseteq$ AM. We have the following surprising theorem. We now prove that $AM = AM_1$. Equivalently, BP.NP=coRP.NP.

**Theorem:** $AM = AM_1$.
**Proof:** Let $L$ be in $AM$ and $A$ be the underlying polynomial predicate. We amplify the success probability such that,

$$x \in L \Longrightarrow P_{y \in \{0,1\}^m}[\exists z \in \{0,1\}^r, A(x, y, z)] \geq 1 - \frac{1}{m}$$

$$x \notin L \Longrightarrow P_y[\exists z, A(x,y,z)] \leq \frac{1}{m}$$

Recall that a set $E \in \{0,1\}^m$ is called thin if $|E| \leq 2^m/m$. $E$ is called fat if $E^c$ is thin. A set may be neither thin nor fat. Let $k \sim \log m$.

We already proved the following lemma about fat and thin sets:-
**Lemma:** If $E$ is fat then,

$$Pr_{s_1,s_2,\ldots s_m \in \{0,1\}^m}[\bigcap_{i=1}^{m}(E \oplus s_i) \neq \phi] = 1$$

If $E$ is thin, then,

$$Pr_{s_1,s_2,\ldots s_m}[\bigcap_{i=1}^{m} E \oplus s_i \neq \phi] \leq \frac{1}{2^{m(k-1)}}$$

Fix an input $x$. Let $E_x = \{y|\exists z, A(x,y,z) = 1\}$. If $x \in L$, then, $E$ is fat. If $x \notin L$, $E$ is thin. We shall construct a new one sided error protocol for $L$ as follows:- Suppose Merlin claims $x \in L$. He will prove this claim by showing that $E_x$ is fat. To do this, Arthur will give him set of random shift strings $s_1, s_2, \ldots s_m$. Merlin will produce a $y \in \bigcap_{i=1}^{m} E \oplus s_i$. By above lemma, for any set of random strings, such a $y$ exists. But, Merlin has to prove that $y$, indeed, has this property. Equivalently, for $1 \leq i \leq m$, $y \oplus s_i \in E$. To prove this, he produces, $z_1, z_2, \ldots z_m$ such that, $A(x, y \oplus s_i, z_i) = 1$. On the other hand, if $x \notin L$, by the second part of the above lemma, the probability that such a $y$ exists is very small.

Formally, we now present a polynomial predicate $A'$ as required by the definition of $AM_1$. $A'(x, s, z')$ takes three arguments, where $x$ is the input string, $s =< s_1, s_2, \ldots s_m >$ will be a string of length $m^2$; $z' =< y', z_1, z_2, \ldots z_m >$ will be a string of length $m + mr$, where $r$ is the length of $z$ in the original AM protocol.

$$A'(x, < s_1, s_2, \ldots s_m >, < y', z_1, z_2, \ldots z_m >) = \bigwedge_{i=1}^{m}(A(x, y' \oplus s_i, z_i)$$

As explained already, with the help of the "fatness" lemma, we can easily verify that $A'$ has the required properties. ♣

We now compare IP with AM. One main difference is that, in IP, the coin tosses of Arthur are *private*, whereas, in AM, they are *public*. In IP, Merlin cannot see the coin tosses of Arthur, whereas, in AM he *can* see them. (In our definition of AM, Merlin can see the "y" and then say his proof "z".) Nothing is hidden from Merlin. To explain this better, recall our IP protocol for Graph non-isomorphism.

Suppose $G_1 \ncong G_2$. The original interactive proof system worked as follows:- the verifier takes $b \in_R \{0,1\}$ and $\sigma \in_R S_n$. Then, sends over $\sigma(G_b)$ to the prover (and hides his coin toss). Then the prover sends back $b'$ and claims that $G_{b'} \cong \sigma(G_b)$. The verifier then verifies this. We can't use this protocol straightaway in the Arthur-Merlin setup. In AM, Merlin invites Arthur to send the bits, i.e., $\sigma$ and $b$. Now Merlin can easily cheat Arthur.

But, we can design a clever AM protocol for GNI. Play this game. Aut($G$) is defined to be the set of all graphs isomorphic to $G$. Look at $\mathrm{Aut}(G_0) \cup \mathrm{Aut}(G_1)$. This set is large, in fact much larger than one of Aut($G_0$) or Aut($G_1$). If $G_0 \cong G_1$ then these sets are the same. Otherwise, we have twice the size for this larger set. Merlin tells us which one of the sets is larger. Merlin engages in a proof of the size of these.

## 7.3 Merlin–Arthur Games

MA is similar to AM, except that Merlin moves first. He presents a proof to Arthur(a probabilistic polynomial time player), who verfies whether the proof is valid.

**Definition (MA):** $L$ is in MA if, there exists a polynomial time computable predicate $A(.,.,.)$ such that,

$$x \in L \Longrightarrow \exists z Pr_y[A(x,y,z) = 1] \geq \frac{3}{4}$$

$$x \in L \Longrightarrow \forall z Pr_y[A(x,y,z) = 1] \leq \frac{1}{4}$$

where $|y|$ and $|z|$ are polynomially bounded in $|x|$.

**Remarks:** Here, if $x \in L$, a single proof "$z$" should work for most of the coin tosses "$y$" of Arthur. If $x \notin L$, no matter what proof Merlin presents, Arthur is unlikely to accept it. MA is like N.BPP. Similar to NP, but the verfication procedure is in in BPP. The proof "z" is a *publishable proof.* You can write it down and then check it. It's like a relaxed way of proving.

Again, we can amplify this from $\frac{3}{4}$ to $1 - 2^{-q(n)}$, for any polynomial $q(n)$.

**Proposition:** $BPP \subseteq MA$.
**Proof:** Arthur simply ignores Merlin, and just tosses some coins and simulats the BPP algorithm. As we remarked, $MA$ is like $N.BPP$, so it's quite obvious that $BPP \subseteq N.BPP = MA$. ♣

It's easy to see that,
**Proposition:** $NP \subseteq MA$. ♣

As in AM, we can define the one sided error version of MA, namely $MA_1$:-
**Definition ($MA_1$):** $L$ is in MA if, there exists a polynomial time computable predicate $A(.,.,.)$ such that,

$$x \in L \Longrightarrow \exists z Pr_y[A(x,y,z) = 1] = 1$$

$$x \in L \Longrightarrow \forall z Pr_y[A(x,y,z) = 1] \leq \frac{1}{4}$$

where $|y|$ and $|z|$ are polynomially bounded in $|x|$.

We saw that $AM = AM_1$. We can prove a similar result for $MA$:-

**Theorem:** $MA = MA_1$. Proof is left as a homework. ♣

$MA$ is like $N.BPP$ and $MA_1$ is like $N.coRP$. We have shown above that $N.BPP = N.coRP$. Can we employ similar techniques to show that BPP ⊆ coRP? Unfortunately, we can't. Because we do not allow the intervening of the prover. In the proof that MA ⊆ MA$_1$ we allowed the prover to give the proof and the shifts, so we did have the intervening of the prover.

We now establish a link between $MA$ and $AM$. We show that $MA \subseteq AM$. Let $L \in MA$ via a polynomial predicate $A(.,.,.)$ that meets the requirements mentioned in the definition of $MA$. One naive attempt to prove $L \in AM$ would be to use the same predicate $A$ for the AM protocol. This would work, when $x \in L$. As per $MA$, there is some proof $z_0$ that works for most of Arthur's coin tosses $y$. In AM, irrespective of Arthur's coin tosses, Merlin can give this same proof $z_0$. This would satisfy, $Pr_y[A(x,y,z_0) = 1] \geq 3/4$. But, this idea fails in case $x \notin L$. $A$ satisfies the condition that, for any $z$, only a small fraction $y$ would work. But, for each $y$, some $z$ may work. So, Merlin can cheat Arthur when $x \notin L$. Our proof will make use of this naive idea, but, by amplifying sufficiently, the success probability of $MA$, we avoid the pitfall described above.

**Theorem:** $MA \subseteq AM$.

**Proof:** Let $\in MA$. As we know that $MA = MA_1$, $L \in MA_1$. Furthermore, we can amplify the success probability so that, we have a polynomial predicate $A(.,.,.)$ that satifies,

$$x \in L \Longleftrightarrow \exists z\, [Pr_y[A(x,y,z) = 1] = 1]$$

$$x \notin L \Longleftrightarrow \forall z \left[Pr_y[A(x,y,z) = 1] \leq \frac{1}{2^{m+1}}\right]$$

where, $m = |z|$, and $m$ is polynomial in $|x|$.

Now we can use the same predicate $A$ for our AM protocol. The main point is that, when $x \notin L$, for any $z$, most of the $y$'s do not work. By a counting argument, we see that, for half of the $y$'s, no $z$ works:-

$$Pr_y[\exists z, A(x,y,z) = 1] \leq \sum_{z \in \{0,1\}^m} [Pr_y[A(x,y,z) = 1] \leq \sum_{z \in \{0,1\}^m} \frac{1}{2^{m+1}} = \frac{1}{2}$$

On the other hand, when $x \in L$, there is some $z_0$, that satisfies, $Pr_y[A(x,y,z_0) = 1] = 1$. This $z_0$ can serve as a proof for Merlin, for all the coin tosses of Arthur. We have shown that,

$$x \in L \Longleftrightarrow Pr_y[\exists z, A(x,y,z) = 1] = 1$$

$$x \notin L \Longleftrightarrow Pr_y[\exists z, A(x,y,z) = 1] \leq \frac{1}{2}$$

We can reduce the probability of getting cheated, from 1/2 to smaller value, by repeatedly running, $A$. ♣

## 7.4 AM With Multiple Rounds

In AM and MA, we had two rounds of interaction, one by Arthur, and one by Merlin. We can generalize to include more rounds of interactions. For example, we have MAM. Merlin moves first, then Arthur moves, then Merlin.

**Definition (MAM):**

$$x \in L \iff \exists z_1 \left[ Pr_y[\exists z_2, A(x, y, z_1, z_2)] \geq \frac{3}{4} \right]$$

$$x \in L \iff \forall z_1 \left[ Pr_y[\exists z_2, A(x, y, z_1, z_2)] \geq \frac{3}{4} \right]$$

We can think of MAM in two ways. One way is MAM=N.AM, just like NP, but the verfication procedure is an AM predicate (instead of a polynomial predicate. To get the second way, first think of MA as MA(P). The undelying predicate is a polynomial predicate. Then, MAM is MA(NP). In other words, take the original definition of MA, and replace the phrase, "where $A$ is a polynomial predicate" by "where $A$ is a NP predicate".

As with MA and AM, we can also have a one sided error version of MAM, named $MAM_1$. We can prove the following theorem, using, essentially, the same technique we used to prove that $AM = AM_1$.

**Theorem :** $MAM = MAM_1$. ♣

Then, we proceed as in the proof of $MA \subseteq AM$, to show that $MAM \subseteq AMM$. But, what is AMM? This means, that Arthur picks the random string $y_1$. Then, Merlin gives a proof $z_1$. Then, again Merlin gives a proof $z_2$. But, Merlin can combine these two moves into a single move and give $z_1 \cdot z_2$. Thus, AMM is same as AM. In other words, we extend the proof $MA \subseteq AM$ to show that $MAM = MA(NP) \subseteq AM(NP)$. But what is AM(NP)? Observe that it is same as AM. We have proved that:-

**Theorem:** $MAM = AM$. ♣

We can generalize further to involve more rounds mixing M's and A's. For example, MAAAMMAAAMM. But we can certainly, replace a streak of A's by a single A and a streak of M's by a single M. In the above example, we have MAAAMMAAAMM⊆ MAMAM. Just like MAM is same as AMM, we can make MAMAM to be AMMAM, by changing the starting MA to be AM. Now again compress the "MM" into "M" to get AMAM. We can continue this process of turning "MAM" into "AM" then, compressing streaks of M's into a single M (and streak of A's into single A) to get AM. Define, AM[k] to be AM with k rounds of interaction. We have "shown" that,

**Theorem:** AM[k]=AM. ♣

A formal proof can be given by by using induction. The above theorem is valid only when $k$ is a constant. What if the number of rounds is dependent on the input length? Meaning, if the input is $x$, we allow $poly(|x|)$ rounds of intreaction. Define AM[poly] to be such a class. Then, it's known that

**Theorem:** AM[poly]=PSPACE. ♣

We have proved that

$$MA_1 = MA \subseteq AM_1 = AM = MAM$$

In fact, we showed that, any finite (fixed number of) iterations of M and A is $\subseteq$ AM.

## 7.5 AM and Other Complexity Classes

We now study the relationship of AM with respect to other classical complexity classes. Recall that, $L$ is said to be in $\Sigma_2^p$, if there exists a polynomial predicate $D(.,.,.)$ such that,

$$x \in L \Longleftrightarrow \exists y \forall z [D(x,y,z) = 1]$$

where $|y|$ and $|z|$ are polynomially bounded in $|x|$. A language $L$ is said to be in $\Pi_2^P$, if its compliment $L^c$ is in $\Sigma_2^P$.

We know that AM is contained in $\Pi_2^P$. This is easy to see that $\text{AM}_1 \in \Pi_2^P$. $L$ is in $\Pi_2^P$, if there is a polynomial predicate $A$ such that,

$$x \in L \Longleftrightarrow \forall y \exists z [A(x,y,z) = 0]$$

$$x \notin L \Longleftrightarrow \exists y \forall z [A(x,y,z) = 1]$$

Verify that, $\text{AM}_1 \subseteq \Pi_2^P$. If $x \in L$, for all coin tosses $y$ of Arthur, Merlin has some proof $z$. Whereas, if $x \notin L$, we have some $y$ (in fact, most of the $y$), for which, all proofs of Merlin are invalid. As $\text{AM}_1 = \text{AM}$, we have proved the theorem,

**Theorem:** AM $\subseteq \Pi_2^p$ ♣

We know that $NP \subseteq$ AM. We also know that graph non-isomorphism is in AM. Can we generalize this to prove that, $coNP \subseteq$ AM? But it is unlikely, as shown by the following theorem.

**Theorem:** If $coNP \subseteq$ AM, the polynomial hierarchy (PH) collapses to AM.
**Proof:** The proof is a simple application of our theorem that $MAM =$ AM. From the definition of $\Sigma_2^P$, we see that, $\Sigma_2^P = N.(coNP)$. As we assumed that, $coNP \subseteq$ AM, we have $\Sigma_2^P \subseteq N.(\text{AM})$. We already observed that $N.\text{AM} = MAM$ and that $MAM =$ AM. Thus, $\Sigma_2^P \subseteq$ AM. As, AM $\subseteq \Pi_2^P$, we have that $\Sigma_2^P = \Pi_2^P =$ AM, which implies that $PH = AM$. ♣

We now prove that $MA \subseteq ZPP^{NP}$. We use ideas from our proof that $BPP \subseteq ZPP^{NP}$.

**Theorem:** $MA \subseteq ZPP^{NP}$.
**Proof:** Let $L \in MA$. There exists a polynomial predicate $D$, such that,

$$x \in L \Longleftrightarrow \exists y \, [Pr_z[D(x,y,z)=1]] = 1$$

$$x \notin L \Longleftrightarrow \forall y \, [Pr_z[D(x,y,z)=1]] \leq \frac{1}{m}$$

where, $|y| = |z| = m$, and $m$ is a polynomial in $n = |x|$. Such an amplification can easily be achieved.

Recall that, a set $E \subseteq \{0,1\}^m$ is called thin, if $|E| < 2^m/m$. $E$ is called fat, if $E^c$ is thin. That is, $|E| \geq (1 - 1/m)2^m$. Call $E$ to be perfect if E consists of all strings in $\{0,1\}^m$.

Recall our proof of $BPP \subseteq ZPP^N P$. There given input $x$, we tested, whether the witness set $E_x$, is fat or thin, in a zero error manner, using a SAT oracle. We are going to do the same, with little more sophistication.

Fix an input $x$. We say that $y$ beats $z$ if $D(x,y,z) = 1$. For a $y$, let $E_y$ be the set of $z$'s that $y$ beats. The main observation is that, if $x \in L$, there there is some $y$, such that, $E_y$ is perfect. If $x \notin L$, then for all $y$, $E_y$ is thin. These are the two conditions that our ZPP machine will test (with the help of an oracle.)

We shall first try to find whether there is a $y$ with perfect $E_y$. We pick, at random some polynomial $z_i$'s. Then ask the oracle, if there is some $y$ that beats all these $z_i$. If the answer is 'no', we can safely reject $x$. But, if the answer is 'yes' we can't be sure and we proceed. We then use self-reducibility to find $y_0$ that beats all these $z_i$. We then ask the oracle, if this $y_0$ beats all z's. If so we can safely accept $x$. If not, we next try to check whether $E_y$ is fat or thin. To do that wee pick $m$ random strings $s_1, \ldots s_m$. And ask the oracle, if $E_{y_0}$, when shifted by these strings will cover the all the $z$'s. Verify that this is an NP question. If $E_{y_0}$ is fat we have a high probability that the answer would be 'yes'. On the other hand, if $E_{y_0}$ is thin, the probability is zero. We also know that, if $x$ is not in $L$, for all $y_0$, $E_{y_0}$ is thin. Thus, if we get the answer 'yes' from the oracle, we can safely accept. If we get the answer 'no', we are in trouble. Three cases are possible here:-

- Case 1: $x \in L$, but the oracle gave us a $y_0$ with a thin $E_{y_0}$;
- Case 2: $x \in L$, the oracle gave us a a good $y_0$, but our shift strings were not good enough to cover (a low probability event).
- Case 3: $x \notin L$, and so the $y_0$ is thin.

So, if we get the answer 'no', we give up. We see that case 2 is a low probability event. Cases 1 and 2, can happen only when there is a thin $y_0$, that can beat our random strings $\{z_i\}$. But $E_{y_0}$ being thin, how can $y_0$ beat so many $z_i$? We argue this formally later.

Formally, the $ZPP$ machine $M$, given SAT oracle works as follows:- Let $x$ be the input string.

1. Pick $m^2$ strings $z_1 \dots z_{m^2}$ at random from $\{0,1\}^m$.

2. Ask the oracle if there is a $y$ such that for all $i$, $D(x, y, z) = 1$. If the answer is 'no', then REJECT. If 'yes', use self-reducibility to get a $y_0$ that beats all these $z_i$.

3. Choose $m$ strings $s_1, \dots s_m$, at random from $\{0,1\}^m$.

4. Ask the oracle, whether $E_{y_0}$ when shifted by these random strings will cover all the $z$'s. That is, ask if the following predicate is true or not

$$\forall z \left[ z \in \bigcup_{i=1}^{m} E_{y_0} \oplus s_i \right]$$

If the above predicate is true, then ACCEPT. If not, output "?".

**Correctness:** In step 2, when we reject, we have that for any $y$, it fails to beat at least one $z_i$. So, there cannot be any perfect $y$. We are correct in rejecting $x$. In step 4, suppose we ACCEPT. We claim that we are again correct in accepting $x$. By contradiction, suppose $x \notin L$. Then every $y$ is bad. For a bad $y$, $E_y < 2^m/m$. So shifting by $m$ strings cannot cover all of $\{0,1\}^m$.

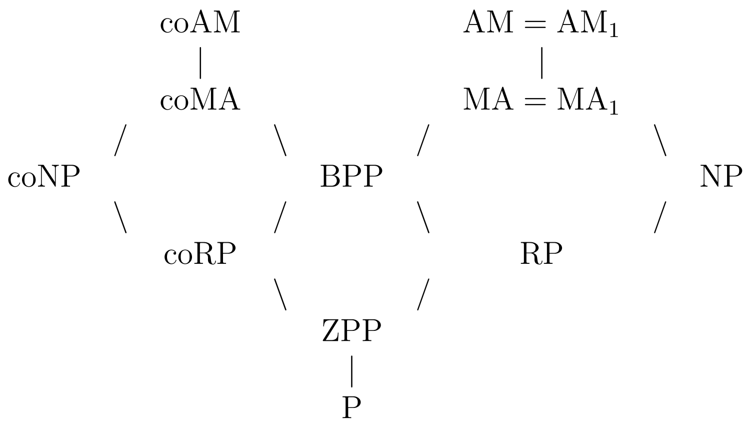
Now we argue that the probability that we output "?" is low. First let's bound the probability that we will have a non-good $y_0$ at the end of step 2. Fix a $y_0$ which is not good. That is $Pr_z[y_0 \text{ beats } z] \leq (1 - 1/m)$. We picked up $m^2$ strings at random. The probability that $y_0$ beats all these strings is $\leq (1 - 1/m)^{m^2} \sim e^{-m}$. There are at most $2^m$ such $y$'s. So, the probability that there exists some $y$ which is not good, but beats all these $m^2$ strings is $\leq 2^m/e^m$. This probability is exponentially small. So, with high probability, there are no non-good witnesses that can beat all the $m^2$ strings. (So, if $x \notin L$, we would stop at step 2, with high probability).

Assume that we proceeded to step 4, and $y_0$ is good. Then, with high probability, the shifting by $m$ strings would cover all of $\{0,1\}^m$. This we proved already in proving that $BPP \in ZPP^{NP}$. Recall the lemma that if $S \in \{0,1\}^m$ has cardinality at least $2^m(1 - 1/m)$, then,

$$Pr_{s_1 \dots s_m \in \{0,1\}^m} \left[ \bigcup_{i=1}^{m} S \oplus s_i = \{0,1\}^m \right] \geq 1 - \frac{1}{2^{m(k-1)}}$$

where $k \sim \log m$. So the probability that we output "?" is bounded by $(2^m/e^m) + (2^{-m(k-1})$. ♣

The structure among the complexity classes, so far is:- /* A BETTER PICTURE WITH ZPP$^{\text{NP}}$ TO BE DRAWN */



## 7.6 LFKN Protocol for Permanent

We start the discussion by reviewing the definition of permanent of a matrix. Let $M = (a_{ij})$ be a $n \times n$ matrix where each entry is $n$ bits long. We define,

$$\text{Perm}(M) = \sum_{\sigma=(\sigma_1,\ldots,\sigma_n)\in S_n} a_{1\sigma_1} \cdots a_{n\sigma_n}$$

where $S_n$ is the set of all permutations over the set $\{1,\ldots n\}$.

There is alternative way to express the permanent. It is similar to the way we usually compute the determinent, except that we do not have those minus signs:

$$\text{Perm}(M) = \sum_{i=1}^{n} a_{1i} \cdot \text{Perm}(M_{1i})$$

where $M_{1i}$ is the $(n-1) \times (n-1)$ submatrix obtained by deleting row 1, and column $i$. The proof of equivalence of the two definitions is that every term appears exactly once in both expressions.

We are now ready to discuss the interactive proof system, The input is a $n \times n$ matrix $M$, with each entry $n$-bits. Since permanent has $n!$ terms, $|\text{Perm}M| \leq 2^{n^{O(1)}}$. The prover $P$ first chooses a huge prime $p > 2^{n^{O(1)}}$ and sends it to the verifier $V$. Using the recent polynomial time AKS algorithm for primality testing, the verifier can check that $p$ is indeed a prime number. The prover next sends the $\text{Perm}(M)$ modulo $p$. As $p$ is large enough, taking modulo $p$ is fine. Hereafter, all the actions take place modulo $p$.

So the prover has produced a number $\alpha$ with the claim that $\text{Perm}(M) = \alpha \bmod p$. If prover's claim is true, we want the verifier to get convinced with probability 1. If not, the verifier has to be convinced with only a small probability.

The protocol revolves around the following polynomial. Define

$$F(x) = \sum_{i=1}^{n} \frac{(x-1)(x-2)\cdots\widehat{(x-i)}\cdots(x-n)}{(i-1)(i-2)\cdots\widehat{(i-i)}\cdots(i-n)} M_{1i}.$$

The above notation means that in the $i^{th}$ term of the summation, $(x - i)$ is missing in the numerator and $(i - i)$ is missing in the denominator. The coffiecients of the polynomial are $(n - 1) \times (n - 1)$ matrices. The polynomial has the following important property.

If we set $x = i$, of the $n$ terms in the summation, except the $i^{th}$ term, all the other terms evaluate to 0. The $i^{th}$ term evaluates to $M_{1i}$. Thus we have $F(1) = M_{11}$, and in general, for $1 \leq i \leq n$, $F(i) = M_{1i}$. $F(x)$ has been constructed using the matrix analog of the well-known Lagrange interpolation The polynomial can be viewed in two ways. It is a polynomial whose coefficients are $(n-1) \times (n-1)$ matrices. It is also a $(n-1) \times (n-1)$ matrix whose entries are polynomials (in $x$) of degree $n - 1$. We will view $F(x)$ mainly using the second method.

Think of $F(x)$ as a $(n - 1) \times (n - 1)$ matrix whose entries are polynomials of degree $(n - 1)$. What is the permanent of $F(x)$? It is a polynomial of degree $\leq (n - 1)^2$. Denote this polynomial as $g(x) = \text{Perm}(F(x))$. The polynomial $g(x)$ has many useful properties. First of all it can used to calculate the permanent of the minors $M_{1i}$ easily:

$$\text{Perm}[M_{1i}] = \text{Perm}[F(i)] = Perm[F(x)]|_{x=i} = g(i).$$

Secondly, we can calculate the permanent of $M$:

$$\text{Perm}[M] = \sum_{i=1}^{n} a_{1i} \cdot \text{Perm}[M_{1i}] = \sum_{i=1}^{n} a_{1i} \cdot g(i).$$

The beauty of the polynomial $g(x)$ is that it can do more! We saw that, for any $1 \leq i \leq n$, $g(i) = \text{Perm}(M_{1i})$. But, in fact, $g(x)$ can be used to compute the permanent of many other matrices! Choose any $r \in Z/p$. Evaluate $F$ at $r$ to obtain the $(n - 1) \times (n - 1)$ matrix $N_r = F(r)$. If $1 \leq r \leq n$, $N_r$ is nothing but $M_{ir}$ and for other values, $N_r$ is some matrix! The main idea of the proof is that $g(r) = \text{Perm}(N_r)$, for any $r \in Z/p$. Thus $g$ can be used to compute permanent of some $p$ many matrices (and $p >> n$). Still $g$ is very short: it is only polynomially long in $n$. Now let us get back to the discussion of the interactive proof system and put our observations into use.

So the prover has sent a value $\alpha$ with claim that $\alpha = \text{Perm}(M)$. Next he sends a polynomial $g'(x)$, with the implicit claim that $g'(x) = g(x)$ (of couse, he could be lying!). The verifier first checks whether these two claims are consistent: check if

$$\alpha = \sum_{i=1}^{n} a_{1i} \cdot g'(i).$$

If the above test fails, verifier halts and rejects the prover's claim. Suppose the above test passed.

We can make a simple observation:

$$\alpha \neq \text{Perm}(M) \Longrightarrow g \neq g'.$$

Even if the prover is a liar, he has to be consistent with his lies! If he lies and gives $\alpha \neq \text{Perm}[M]$, he has to lie again and give a $g'$ different from $g$. Otherwise, the above consistency check will fail and verifier will catch him!

The next step is the heart of the proof. The verifier computes the polynomial $F(x)$. He can compute $F(x)$ easily, without the help of the prover. Next the verifier chooses a random number $r$ from $Z/p$ and computes $N_r = F(r)$. As we saw already, permanent of $N_r$ can be computed

$$\text{Perm}(N_r) = \text{Perm}[F(r)] = g(r).$$

Of course, the verifier cannot compute the polynomial $g$. But, he has the $g'$ given by the prover, who claimed that $g' = g$. So the verifier simply computes $g'(r)$ and assumes that $g'(r) = \text{Perm}(N_r)$.

There are two possible cases to analyze: $g' = g$ and $g' \neq g$. If $g' = g$, then clearly, $g'(r) = \text{Perm}(N_r)$. On the other hand, suppose $g' \neq g$. What is the probability (over the random choice of $r$) that $g'(r) = \text{Perm}(N_r)$? We know that $g$ and $g'$ are both polynomials of degree $(n-1)^2$. So they can agree on at most $(n-1)^2$ many choices of $x \in Z/p$. As $r$ was a random value chosen from $Z/p$,

$$\Pr_{r \in Z/p}[g'(r) = g(r)] \leq \frac{(n-1)^2}{p}.$$

We chose $p$ to be exponential in $n$. So except for a (exponentially) small probability, if $g' \neq g$, then $g'(r) \neq \text{Perm}(N_r)$. Let us summarize the discussion so far: if $\alpha \neq Perm[M]$, then $g' \neq g$ and hence, with probability exponentially close to 1, $g'(r) \neq \text{Perm}(N_r)$. But, by claiming that $g = g'$, the prover has, implicitely, claimed that $\text{Perm}(N_r) = g'(r)$.

The situation the same as the one with which the story started, but it is ”smaller”. Initially, the provr claimed that $\alpha = \text{Perm}(M)$, where $M$ was a $n \times n$ matrix. Now, he has claimed that $g'(r) = \text{Perm}(N_r)$, where $N$ is a $(n-1) \times (n-1)$ matrix. And if the prover' first claim was a lie, then the second claim is also a lie, with very high probability! How can the verifier validate the claim? Simply, by repeating the same process!

After $(n-1)$ rounds, the two players would be left with a matix of size $1 \times 1$, say $(x)$, and the prover would have (implicitely) claimed its permanent to be some $\beta$. The verifier can esily check this: Is $\text{Perm}[(x)] = x = \beta$? If not he rejects the original claim of the prover that $\alpha = Perm(M)$. If $\alpha \neq \text{Perm}(M)$, then probability that $\beta \neq \text{Perm}[(x)]$ is at least $1 - n \cdot (n-1)^2/p$. (The factor $n$ comes from the $n$ rounds). We conclude that a honest prover will win with probability 1. And a dishonest prover will win with probability exponentially small.

**Exact protocol:** $M$ is the $n \times n$ input matrix. The prover claims that $Perm(M) = \alpha$. The first thing prover does is to choose a prime number $p \sim 2^{n^3}$, (note: $p >> Perm(M)$). He sends it to the prover along with a proof that it's prime (using Pratt's theorem that Primes$\in$ NP). All the computation hence forth take place modulo $p$.

1. Prover's claim is that $Perm(M) = \alpha$, where $M = (a_{ij})$ is a $n \times n$ matrix.

2. If $M = (x)$ is a $1 \times 1$ matrix, the verifier checks $\alpha = x$. If they do not match he *rejects* else *accepts*.

3. The verifier computes

$$F(x) = \sum_{i=1}^{n} \frac{(x-1)(x-2)\cdots\widehat{(x-i)}\cdots(x-n)}{(i-1)(i-2)\cdots\widehat{(i-i)}\cdots(i-n)} M_{1i}.$$

4. Prover sends a polynomial $g'(x)$ degree $(n-1)^2$ (with the implicite claim that $g'(x) = \mathrm{Perm}[F(x)]$).

5. Verifier checks

$$\alpha = \sum_{i=1}^{n} a_{1i} \cdot g'(i).$$

   If they do not match *reject*.

6. Verifier picks $r \in_R Z/p$ and sends it to prover.

7. Verifier computes $N_r = F(r)$, a $(n-1) \times (n-1)$ matrix.

8. Verifier sets $M \longleftarrow N_r$, $n \longleftarrow n-1$ and $\alpha \longleftarrow g'(r)$.

9. Go back to step 1.

If the $\alpha = Perm(M)$, the prover can provide correct polynomials $g$ and the verifier will accept with probability 1. Suppose $\alpha \neq \mathrm{Perm}(M)$. The verifier will reject in step 4 of some round. If that does not happen, to make verifier accept in the end in step 1, the verifier has to be lucky in at least one round, where the random $r$ satisfies $g'(r) = g(r)$. Probability for this to happen in any particular round is $\leq (n-1)^2/p$. So probability for it to happen in one of the $n$ rounds is $\leq n(n-1)^2/p$. As $p \sim 2^{n^3}$, this error probability is exponentially small. ♣

# Chapter 8

# IP=PSAPCE

# Chapter 9

# Derandomization

**Chapter Outline:** Derandomization intro. Psudo random generators. Next-bit prediction and pseudo-random equivalence. Oneway functions. Weak-oneway implies strong oneway. Goldreich Levin hardcore bit. Constructing pseudo random generators using GL-hardcore bit.

AN INTRO TO DERANDOMIZATION IS REQUIRED HERE.

## 9.1 Pseudorandom Generators

Informally, a pseudorandom generator $G$ is a polynomial time algorithm, that stretches a short random string called seed into a long pseudorandom string. $G$ is a successful, if its output cannot be distinguished from truly random strings, by any efficient adversary. In other words, the output of $G$ is computationally indistinguishable from truly random strings. We formalize the notion of computational indistinguishability in two models, and show their equivalence.

A *generator* is a polynomial-time computable, deterministic function $G : \{0,1\}^n \to \{0,1\}^{l(n)}$ where $l(n) > n$. An *adversary* is a probabilistic polynomial time Turing machine.

**Pseudorandom Generators:** We call the first model of formalism as pseudorandom model. Here, we adopt the following convention. An adversary $A$, given an input string, outputs 1, if it it thinks its input is pseudorandom, and outputs if it thinks its input is truly random. With respect to a generator $G$, we define the the success of an adversary $A$, at a length $n$ as:-

$$\delta_n^A = |\mathrm{Pr}_{x\in\{0,1\}^n}[A(G(x)) = 1] - \mathrm{Pr}_{y\in\{0,1\}^{l(n)}}[A(y) = 1]|$$

So $\delta_n^A$ is larger if the adversary is distinguishing well.

**Definition (Pseudorandom Generator):** We say that $G$ beats $A$ if $\forall$ polynomials $p(\cdot)$, $\forall$ sufficiently large $n$, we have $\delta_n^A < \frac{1}{p(n)}$. $G$ is defined to be a pseudorandom generator if $G$

beats every adversary.

**Remark:** By the above definition, $A$ beats $G$ if there exists $p(\cdot)$ such that for infinitely many $n$ we have $\delta_n^A > \frac{1}{p(n)}$.

**Next bit unpredictability:** In this second model, adversary $A$ is given first $i$ bits of the output of the generator. It has to predict the $(i+1)^{th}$ bit. For an adversary $A$, at a length, the success is measured as,

$$\beta_n^A = \Pr_{\substack{x \sim U_n \\ i \in_R [0..l(n)-1]}} [A(G(x)_{1,\ldots,i}) = G(x)_{i+1}] - \frac{1}{2}$$

Thus, we take average over all $i$. Any adversary can easily get a success of 1/2 by simply guessing. So, we have subtracted 1/2, to normalize.

**Definition (Next-bit unpredictable generator):** We say that $G$ beats $A$ if $\forall p(\cdot)$, $\forall$ sufficiently large $n$, we have $\beta n^A < \frac{1}{p(n)}$. $G$ is defined to be a next-bit unpredictable if $G$ beats every adversary.

The first model is due to Yao and the second is due to Blum and Micali. The equivalence was shown by Yao.

**Theorem (Yao):** $G$ is pseudorandom if and only if $G$ is next-bit predictable.
**Proof:** Equivalently, $G$ is not pseudorandom if and only if $G$ is not next-bit predicatable. So, we show that, some adversary beats $G$ in the pseudorandom model if and only if some adversary beats $G$ in the next bit model.

**Part I:** Suppose $A$ is an adversary in the next bit model. We want to find an adversary $A'$ that performs equally good in the pseudorandom model.

Fix $n$. $A'$ takes as input a $l(n)$ bit string and has to predict whether its truly random or pseudorandom. $A'$ uses it $A$ as an oracle. $A'$ on input $y_1, \ldots y_{l(n)}$, does the following:-

1. First choose a random $i \in [0..l(n)-1]$.
2. If $A(y_1, \ldots, y_i) = y_{i+1}$ then output 1, i.e., guess pseudorandom. Otherwise output 0, i.e., guess truly random.

Suppose the success of $A$, at length $n$ is, $\beta_n^A$.

$$Pr_{x \sim U_n}[A'(G(x)) = 1] = Pr_{\substack{x \sim U_n \\ i \in_R [0..l(n)-1]}} [A(G(x)_{1,\ldots,i}) = G(x)_{i+1}] = \beta_n^A + \frac{1}{2}$$

If the input to $A'$ is truly random, irrespective of the $i$ chosen by $A'$, $A$ can output $y_{i+1}$ with probability only 1/2. So,

$$Pr_{y \in \{0,1\}^{l(n)}}[A(y) = 1] = \frac{1}{2}$$

Thus,

$$\delta_n^{A'} = Pr_{x \in \{0,1\}^n}[A(G(x)) = 1] - Pr_{y \in \{0,1\}^{l(n)}}[A(y) = 1] = \beta_n^A + \frac{1}{2} - \frac{1}{2} = \beta_n^A$$

Thus, for all $n$, $\delta_n^{A'} = \beta_n^A$. If $A$ can beat $G$ in next-bit model, $A'$ can beat $G$ in the pseudorandom model.

**Part II:** Now suppose that $A$ is an adversary for $G$ in the pseudorandom random model, with success $\delta_n^A$ at length n. We construct an adversary $A'$ in the next bit model.

Fix $n$. Let the input to $A'$ be $b_1, \ldots, b_i$. $A'$ has to predict the next bit $b_{i+1}$. $A'$ does the following:-

1. Choose $l(n) - i$ random bits $r_{i+1}, \ldots, r_{l(n)}$ at random.
2. Compute $a = A(b_1, \ldots, b_i, r_{i+1}, \ldots, r_{l(n)})$.
3. If $a = 1$ output $r_{i+1}$, else output $\overline{r_{i+1}}$.

The intuition is that $A$ outputs 1 more often when given all pseudorandom bits. And outputs 0 more often when given all truly random bits. We input $A$ with a mix of $i$ pseudorandom bits and $(l(n) - i)$ truly random bits. $b_{i+1}$ is either $r_{i+1}$ or its compliment. If $r_{i+1} = b_{i+1}$ then $A$ gets one more pseudorandom random bit and is more likely to output 1. To put it in a reverse Bayesian way, if $A$ outputs 1, we are more likely to have $r_{i+1} = b_{i+1}$. To get a formal proof, we first quantify the behavior of $A$ when given a mix of pseudorandom and truly random bits. Suppose we input $A$ with $i$ pseudorandom bits and $l(n) - i$ truly random bits. The probability that $A$ outputs 1, for such a mix, is denoted by $p_i$. Formally,

$$p_i = Pr_{\substack{x \sim U_n \\ r_{i+1}, \ldots, r_{l(n)} \in_R \{0,1\}}} [A(G(x)_{1,\ldots,i} \circ r_{i+1}, \ldots, r_{l(n)}) = 1$$

First observe that,

$$p_0 = Pr_{y \in_R \{0,1\}^{l(n)}}[A(y) = 1]$$
$$p_{l(n)} = Pr_{x \in_R \{0,1\}^n}[A(G(x)) = 1]$$

Thus, $p_{l(n)} - p_0 = \delta_n^A$.

We next compute the success of $A'$ in terms of $\delta_n^A$. For any $0 \leq i < l(n)$, we denote the success of $A'$ as $\beta_{n,i}^{A'}$. We let $w = b_1, \ldots b_i, r_{i+1}, \ldots, r_{l(n)}$. Then,

$$\begin{aligned}
\beta_{n,i}^{A'} &= Pr[A'(b_1, \ldots b_i) = b_{i+1}] \\
&= Pr[A(w) = 1 \text{ and } r_{i+1} = b_{i+1}] + Pr[A(w) = 0 \text{ and } r_{i+1} \neq b_{i+1}] \\
&= Pr[r_{i+1} = b_{i+1}] \cdot Pr[A(w) = 1 | r_{i+1} = b_{i+1}] + Pr[r_{i+1} \neq b_{i+1} \cdot Pr[A(w) = 0 | r_{i+1} \neq b_{i+1}] \\
&= \frac{1}{2} \left( Pr[A(w) = 1 | r_{i+1} = b_{i+1}] + Pr[A(w) = 0 | r_{i+1} \neq b_{i+1}] \right)
\end{aligned}$$

The first quantity inside the parenthesis, is easy to compute. When $r_{i+1} = b_{i+1}$, we are giving one more pseudorandom bit to $A$. So,

$$Pr[A(w) = 1 | r_{i+1} = b_{i+1}] = p_{i+1}$$

. To compute the second quantity:-

$$\begin{aligned} p_i &= Pr[A(b_1, \ldots, b_i, r_{i+1}, \ldots, r_{l(n)}) = 1] \\ &= Pr[r_{i+1} = b_{i+1}] \times Pr[A(w) = 1 | r_{i+1} = b_{i+1}] + Pr[r_{i+1} \neq b_{i+1} \times Pr[A(w) = 1 | r_{i+1} \neq b_{i+1}] \\ &= \frac{1}{2} p_{i+1} + \frac{1}{2} Pr[A(w) = 1 | r_{i+1} \neq b_{i+1}] \end{aligned}$$

So,

$$Pr[A(w) = 0 | r_{i+1} \neq b_{i+1}] = p_{i+1} - 2p_i + 1$$

Substituting in the equation for $\beta_{n,i}^{A'}$, we get,

$$\beta_{n,i}^{A'} = \frac{1}{2} \left( p_{i+1} + p_{i+1} - 2p_i + 1 \right) = \frac{1}{2} + p_{i+1} - p_i$$

Now we can compute the success $\beta_n^{A'}$ of $A'$ when $i$ is chosen at random.

$$\begin{aligned} \beta_n^{A'} &= \sum_{i=0}^{l(n)} \frac{1}{l(n)} \beta_{n,i}^{A'} \\ &= \sum_{i=0}^{l(n)} \frac{1}{l(n)} \left( \frac{1}{2} + p_{i+1} - p_i \right) \\ &= \frac{1}{2} + \frac{1}{l(n)} \sum_{i=0}^{l(n)} p_{i+1} - p_i \\ &= \frac{1}{2} + \frac{1}{l(n)} (p_n - p_0) \\ &= \frac{1}{2} + \frac{\delta_n^A}{l(n)} \end{aligned}$$

The success probability of $A'$ is polynomially related to that of $A$. So, if $A$ beats $G$ in the pseudorandom model, $A'$ beats $G$ in the next-bit model. ♣

## 9.2 One-way Functions

In this lecture, we define weak and strong one-way functions. Informally, a function $f$ is one-way, if it's easy to compute $f(x)$ given $x$, but hard to compute $x$ given $f(x)$. The notion of weak one-way requires that any probabilistic polynomial time algorithm that tries to invert $f$, fails for at least a significant fraction of inputs. The notion of strong one-way requires that any probabilistic polynomial time algorithm fails to invert $f$, except for an insignificant fraction of inputs. We make these notions precise and then show that existence of weak one-way function implies the existence of strong one-way functions.

**Definition (weak one way function):** A function $f : \{0,1\}^n \to \{0,1\}^{p(n)}$ is called weak one way function, if there exist an $l > 0$ such that, for all probabilistic polynomial time algorithms $A$, for sufficiently large $n$, $A$ fails to invert $f$ fails in, at least, $1/n^l$ fraction of the inputs. Formally,

$$Pr_{x \sim U_n}\left[f(A(f(x)) \neq f(x)\right] \geq \frac{1}{n^l}$$

**Definition (strong one way function):** A function $f : \{0,1\}^n \to \{0,1\}^p(n)$, is called strong one way function, if for all probabilistic polynomial time algorithm $A$, for all polynomials $p$, for sufficiently large $n$,

$$Pr_{x \sim U_n}\left[f(A(f(x)) = f(x)\right] \leq \frac{1}{p(n)}$$

In other words, $A$ can invert $f$ in only a negligible fraction of inputs.

**Theorem (Yao):** Strong one-way functions exist if weak one-way functions exist. **Proof:** Assume that $f : \{0,1\}^n \to \{0,1\}^{p(n)}$ is a weak one-way function. We construct a strong one-way function $F : \{0,1\}^{m \times n} \to \{0,1\}^{m \times p(n)}$. $m$ is a parameter polynomial in $n$ and we will fix it later. $F$ is defined to be

$$F(x_1, x_2 \ldots x_m) =< f(x_1), f(x_2), \ldots f(x_m) >$$

The claim is that $F$ is strongly one-way. The intuition is as follows:- Suppose an algorithm tries to invert $F$ in the naive way, by inverting each component of its input. It will have a success probability of $\left(1 - \frac{1}{n^l}\right)$ in inverting each individual component. So it will have a success probability $\leq \left(1 - \frac{1}{n^l}\right)^m$ in inverting $F$. If we choose, $m = n^{l+1}$, this probability will be $\leq e^{-m}$. We next prove the claim formally.

By contradiction, assume that $F$ is not strong one-way. Then there is an algorithm $A$ can invert $F$ on $\geq e^{-n} + \frac{1}{n^k}$ fraction of its inputs, for some $k$. We want to invert $f$ on $> 1 - \frac{1}{n^l}$ fraction of our $2^n$ inputs.

Our algorithm $A'$ to invert $f$ is the following: Upon input $y = f(x)$, first randomly pick $i \in_R \{1, \ldots, m\}$. Then, randomly (uniformly) pick $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_m \in \{0,1\}^n$. Construct,

$$Y = \langle f(x_1), \ldots, f(x_{i-1}), y, f(x_{i+1}), \ldots, f(y_m) \rangle$$

where $y = f(x)$ is in the $i$th position. Ask $A$ to invert $Y$. Let it's output be $X' = (x'_1, x'_2, \ldots x'_m)$. If $y = f(x'_i)$, output $x'_i$. If $A'$ succeeded in inverting $F$ at $Y$, we would have succeeded in inverting $y$. We repeat the process some polynomial (to be fixed later) number of steps, to enhance our success probability. If we are unsuccessful in all these polynomial number of trials, we give up.

We now argue that our algorithm inverts $f$ in more than $1/n^l$ fraction of its inputs. Define a bipartite graph of $2^n$ vertices on one side and $2^{m \cdot n}$ vertices on the other side. Strings in $\{0,1\}^n$ are the vertices in the left side $L$ and strings in $\{0,1\}^{m \cdot n}$ are the vertices on the right side $R$. Add an edge between $x$ and $(x_1, \ldots, x_m)$ if $x = x_i$, for some $i$. The graph is a multigraph (meaning, there may be more than one edge between a pair of vertices). For a node $x$, we let $N(x)$ denote the multi-set of its neighbors. Let $G \subseteq R$ are the points at which $A$ can invert $F$.

Now let's review our algorithm $A'$. Given $f(x)$, $A'$ (implicitly) chooses a random element $r \in_R N(x)$, then computes $Y = F(r)$ and then asks $A$ to find $r$. $A'$ wants to find $x$=$i^{th}$ component of $r$. If $N(x) \cap G$ is large, then with high probability $Y$ will be in $G$. If $Y \in G$, then $A'$ will invert $Y$ successfully and hence $A'$ will invert $f(x)$ successfully. We shall argue that for "large" fraction of $x$, $N(x) \cap G$ is "large".

To make the above argument formal, let's first state some simple facts. $|L| = 2^n$. $|R| = 2^{m \cdot n}$. The degree of each node in $R$ is $m$. The degree of each node in $L$ is $m2^{n(m-1)}$. Our bipartite graph is an "expander", in the following sense.

A bipartite graph has $(\epsilon, \delta)$-expansion property, if for all $S \subseteq L$, with $|S| \geq \epsilon|L|$, we have that $|N(S)| \geq (1-\delta)|R|$. In other words, for any small set in $L$, its neighbor set is large. Our graph has good expansion properties.

Let $S \subseteq L$ be of size $|S| \geq \epsilon|L|$. What's the size of $N(S)$? Pick a random $Y \in R$. Let $Y = (x_1, x_2, \ldots x_m)$. What's the probability that $Y \notin N(S)$? So,

$$Pr[Y \notin N(S)] = Pr[x_1 \notin S, \ldots x_m \notin S] \leq (1-\epsilon)^m$$

We want this probability to exponentially small. So, choose $m = n/\epsilon$. Then,

$$Pr[Y \notin S] \leq (1-\epsilon)^m \leq e^{-n}$$

And

$$|N(S)| \geq (1 - e^{-n})|R|$$

We shall fix the parameters now. Fix:-

$$\epsilon_0 = \frac{1}{n^l}; m = \frac{n}{\epsilon_0} = n^{l+1}; \delta_0 = e^{-n}$$

Thus, our graph has $(\epsilon_0, \delta_0)$-expansion property.

We assumed that the adversary $A$ invert $F$ in $e^{-n} + \frac{1}{n^k}$ fraction of the inputs. Let $\alpha = \frac{1}{n^k}$. Then $|G| \geq (\delta_0 + \alpha)|R|$.

Let $S \subseteq L$, be any set with $|S| \geq \epsilon_0|L|$. As our graph has $(\epsilon_0, \delta_0)$-expansion property, $|N(S)| \geq (1-\delta_0)|R|$. So,

$$|G \cap N(S)| = |G| - |G \cap \overline{N(S)}| \geq |G| - |\overline{N(S)}| \geq (\alpha + \delta_0)|R| - \delta_0|R| = \alpha|R|$$

So $|G \cap N(S)| \geq \alpha|R|$. The left degree of the graph is $m2^{n(m-1)}$. So, the total number of edges incident on $S$ is $|S|m2^{n(m-1)}$. How many of these are good? Meaning, how many have the other vertex in $G$? From each one of these vertices in $|G \cap N(S)|$, at least one edge goes to $S$. Thus, the number of good edges incident on $S$ is at least $\alpha|R|$. What's the average number of good edges incident on $S$?

$$\frac{\#\text{good edges}}{\text{total \# of edges}} \geq \frac{\alpha|R|}{Sm2^{n(m-1)}} = \frac{\alpha 2^{n \cdot m}}{Sm2^{n(m-1)}} \geq \frac{\alpha}{m}\frac{2^{nm}}{2^n 2^{n(m-1)}} = \frac{\alpha}{m}$$

So, for any set $S$ of size at least $\epsilon_0|L|$, the fraction good edges is more than $\frac{\alpha}{m}$.

Now consider the set,

$$S = \left\{ x \in L \;\middle|\; \frac{|\{(x,g) \mid g \in G\}|}{|N(x)|} < \frac{\alpha}{m} \right\}$$

For this set, the fraction of good edges is $< \frac{\alpha}{m}$. So its size should be $< \epsilon_0|L|$. So, the set

$$D = \left\{ x \in L \;\middle|\; \frac{|\{(x,g) \mid g \in G\}|}{\deg(x) = |N(x)|} \geq \frac{\alpha}{m} \right\}$$

has cardinality $|D| \geq (1-\epsilon_0)|L|$.

We are given $f(x)$ as input to invert. We randomly pick a neighbor $r$ of $x$, compute $Y = F(r)$, use $A$ to invert $Y$ and get $x$. Suppose $x \in D$. Then, with probability $\geq \frac{\alpha}{m}$, $r$ will be in $G$ and we will be successful. If we repeat the process $\frac{m}{\alpha}$, we will be successful. The main point is that $\alpha$ was assumed to be $1/n^k$, a inverse polynomial. So $m/\alpha$ is polynomial. As $|D| \geq (1-\epsilon_0)|R|$, $Pr_x[x \in D] \geq (1-\epsilon_0) = 1 - \frac{1}{n^l}$. Thus, we will invert $f$ with probability $\geq 1 - \frac{1}{n^l}$. This contradicts the assumption that $f$ is $1/n^l$-hard. ♣

## 9.3 Goldreich-Levin Hardcore Bit

Let $f$ be a strong one-way permutation. A hardcore predicate for $f$ is a boolean function $g$ such that, it's hard to compute $g(X,Y)$, given $f(X)$ and $Y$, Formally, for any probabilistic polynomial time algorithm, for all polynomials $p$, for sufficiently large $n$,

$$Pr_{x,y \in_R \{0,1\}^n}[A(f(x),y) = g(x,y)] \leq \frac{1}{2} + \frac{1}{p(n)}$$

We show that such predicates can be constructed for any strong one-way permutation $f$. The construction is due to Goldreich and Levin. Define $g(X,Y)$ to be the inner product $X \cdot Y$ of $X$ and $Y$. That is, if $X = x_1x_2\ldots x_n$ and $Y = y_1y_2\ldots y_n$, then, $X \cdot Y = \sum x_iy_i$, where the summation is over mod 2.

We claim that $g$ is a hardcore predicate for $f$. On the contrary, suppose there is an algorithm $A$ that can compute $g(X,Y)$, given $f(X)$ and $Y$, with probability $\frac{1}{2} + \frac{1}{n^c}$. We show that, using $A$ as a black-box, $f$ can be inverted with high probability.

For two string $X = x_1 \ldots x_n$ and $Y = y_1 \ldots y_n$, we define $X+Y$ to be the bit-wise XOR of $X$ and $Y$. That is, $X+Y = z_1 \ldots z_n$, where $z_i = x_i + y_i \pmod 2$ . Define $\widehat{X} = \bar{x_1}x_2\ldots x_n$, the string $X$ with first bit flipped. We have $X \cdot (Y+Z) = X \cdot Y + X \cdot Z$. So, $X \cdot Y + X \cdot \widehat{Y} = x_1$. This last property plays an important role in our construction. Our goal is to compute $x_1$ correctly, with high probability. Then use the same procedure to compute other bits of $X$, there by inverting $f$.

Let $m = O(\log n)$. We will fix its exact value later. Choose $m$ strings $Y_1, Y_2, \ldots Y_m \in \{0,1\}^n$ at random. Let $\epsilon_1, \epsilon_2, \ldots, \epsilon_m$ be some fixed $m$ bits, not all 0. Define $Y_{\epsilon_1\epsilon_2\ldots\epsilon_m} = \sum \epsilon_i Y_i$. The above one is a random variable. It will be uniformly distributed over $\{0,1\}^n$. Take any $\delta_1\delta_2\ldots\delta_m \in \{0,1\}^m - \{0\}^m$, such that $\epsilon_1 \ldots \epsilon_m \neq \delta_1 \ldots \delta_m$. For some $i$, $\epsilon_i = 1$ and $\delta_i = 0$ (or vice versa). $Y_i$ is chosen uniformly from $\{0,1\}^m$ and it contributes only to $Y_{\epsilon_1\ldots\epsilon_m}$. So the random variables $Y_{\epsilon_1\ldots\epsilon_m}$ and $Y_{\delta_1\ldots\delta_m}$ are independent. Thus the set of $2^m - 1$ random variables $\{Y_{\epsilon_1\ldots\epsilon_m}\}$, for $\epsilon_1 \ldots \epsilon_m \in \{0,1\}^m - \{0\}^m$, are pairwise independent.

Now we are ready to present our construction. We first choose $Y_1, Y_2, \ldots, Y_m \in_R \{0,1\}^n$. Then guess the values of $X \cdot Y_1, X \cdot Y_2, \ldots, X \cdot Y_m$ to be $b_1, b_2, \ldots, b_m$. Assume that we guessed correctly! Then, for any $\epsilon_1, \ldots, \epsilon_m$ we can compute $X \cdot Y_{\epsilon_1\epsilon_2\ldots\epsilon_m}$ correctly:-

$$X \cdot Y_{\epsilon_1\epsilon_2\ldots\epsilon_m} = X \cdot (\epsilon_1 Y_1 + \cdots + \epsilon_m Y_m) = \sum \epsilon_i (X \cdot Y_i) = \sum \epsilon_i b_i$$

Fix some $\epsilon_1, \ldots, \epsilon_m$ and compute $X \cdot Y_{\epsilon_1\epsilon_2\ldots\epsilon_m}$ as above. Then use the black-box $A$ to compute $X \cdot \widehat{Y_{\epsilon_1\epsilon_2\ldots\epsilon_m}}$. Assume that $A$ computed this correctly. We can now compute the first bit $x_1$ of $X$, $x_1 = X \cdot Y_{\epsilon_1\epsilon_2\ldots\epsilon_m} + X \cdot \widehat{Y_{\epsilon_1\epsilon_2\ldots\epsilon_m}}$. (Recall that for any $Y$, $x_1 = X \cdot Y + X \cdot \widehat{Y}$.) How to make the correct guesses for $b_1, \ldots, b_m$? As $m = O(\log n)$, total number of possible values for $b_1 \ldots b_m$ is polynomial in $n$. So we can run the above procedure for every possible sequence of guesses. And one of the guesses would be correct. To make the black-box work well with high probability, we run the above procedure over all possible (polynomial number of) $\epsilon_1, \ldots \epsilon_m$. Then, we take the majority. We would get $x_1$ this way. Similarly, compute other bits. Using Chebyshev inequality, we shall show that with high probability (over the random choices for $Y_1, Y_2, \ldots, Y_m$), we would have got the correct $x_1, x_2, \ldots, x_m$. We can verify whether we are correct by checking $f(x_1 x_2 \ldots x_m) = f(X)$. If we are not correct, we run the experiment again, by choosing at random another set of $Y_1, Y_2, \ldots, Y_m$.

We next present the algorithm in full. The input is $f(X)$. It tries to compute $X$.

```
Choose $Y_1, Y_2, \ldots, Y_m \in_R \{0,1\}^n$.
For all $2^m$ choices of $b_1 \ldots b_m \in \{0,1\}^m$ do:-
     For $1 \le k \le n$ do:-
          For each $\epsilon_1 \ldots \epsilon_m \in \{0,1\}^m - \{0\}^m$ do:-
               Use $A$ to compute the the vote
                         $V_{\epsilon_1,\ldots,\epsilon_m} = \sum_{i=1}^{m} \epsilon_i b_i + A[f(X), Y_{\epsilon_1\epsilon_2\ldots\epsilon_m} + 0^{k-1}10^{n-k}]$
          Set $x_k$ to be the majority of the above $2^m - 1$ votes.
     If $f(x_1, \ldots, x_k) = f(X)$, output $x_1 \ldots x_k$ and EXIT.
```

The above procedure runs in polynomial time, as $m = O(\log n)$. In the above procedure, as we run through all possible $b_1, \ldots, b_m$, in some iteration we would achieve $\forall i[b_i = X \cdot Y_i]$. Consider that particular iteration. What's the chance that we would get $x_1$ correctly? This depends upon the what's the majority of the votes. Each $\epsilon_1, \ldots, \epsilon_m$ casts a vote for the value of $x_1$. The vote $V_{\epsilon_1,\ldots,\epsilon_m}$ will be correct, if $A$ succeeds in computing $X \cdot \widehat{Y_{\epsilon_1\epsilon_2\ldots\epsilon_m}}$ correctly. We already observed that the $2^m - 1$ random variables $Y_{\epsilon_1\epsilon_2\ldots\epsilon_m}$ are uniformly distributed over $\{0,1\}^n$ and they are pairwise independent. So, by our assumption that $A$ can compute the inner product with probability $\geq \frac{1}{2} + \frac{1}{n^c}$, each vote will be correct with at least that much probability. We now apply Chebyshev over these $2^m - 1$ votes. Let $\chi$ be a random variable that counts the number of correct votes among these $2^m - 1$ votes. $\chi$ is a sum of $2^m - 1$ pairwise independent 0-1 random variables. Each of these individual random variable has a success probability, at least, $\frac{1}{2} + \frac{1}{n^c}$. Expectation of $\chi$ is

$$E[\chi] \geq (2^m - 1)\left(\frac{1}{2} + \frac{1}{n^c}\right)$$

Due to pairwise independence, var $\chi$ is the sum of variances of the individual 0-1 random variables. The variance of any 0-1 random variable is at most 1. So, var $\chi \leq 2^m - 1$. By Chebyshev,

$$\begin{aligned}
Pr[\text{ majority of } 2^m - 1 \text{ votes wrong }] &= Pr\left[\chi < \frac{2^m - 1}{2}\right] \\
&\leq Pr\left[|\chi - E[\chi]| > \frac{2^m - 1}{n^c}\right] \\
&\leq \frac{\text{var } \chi}{\left(\frac{1}{n^c}(2^m - 1)\right)^2} \\
&= \frac{(2^m - 1)n^{2c}}{(2^m - 1)^2} \\
&= \frac{n^{2c}}{2^m - 1}
\end{aligned}$$

Take $m \sim (2c \log n + 2)$. Then the probability that $x_1$ is not correct is at most $\frac{1}{n^2}$. The probability that any one of $x_1, x_2, \ldots, x_n$ is not correct is at most $\frac{1}{n}$. By repeating the above experiment, we can improve our success probability.

## 9.4 Construction of Pseudorandom Generators

Show how to construct PRG using gl-hardcore bit.

# Chapter 10

# Computing With Circuits

For the next few lectures, we will deal with circuit complexity. First we will concentrate on small depth circuits. As the depth of a circuit corresponds to parallel computation time, small depth circuit complexity captures efficient parallel computation. Before we prove circuit lower bounds, which show cannot be computed by small depth circuits, we should gain some appreciation of what can be done by these circuits. So we first exhibit some computational power of these circuits. We start with one of the simplest computations: integer addition.

## 10.1 Binary Addition

Given two binary numbers, $a = a_{n-1} \dots a_1 a_0$ and $b = b_n b_{n-1} \dots b_1 b_0$, we can add the two using the *school method* – adding each column from right (the least significant bit) to left (the most significant bit), with carry bit along the way. In other words, $r = a + b$, where

$$\begin{array}{rrccc} & a_{n-1} & \dots & a_1 & a_0 \\ + & b_{n-1} & \dots & b_1 & b_0 \\ \hline r_n & r_{n-1} & \dots & r_1 & r_0 \end{array}$$

can be accomplished by first computing $r_0 = a_0 \oplus b_0$ ($\oplus$ is exclusive or, which can be further written out in terms of $\vee$ and $\wedge$) and computing a carry bit, $c_1 = a_0 \wedge b_0$. Then, we can compute $r_1 = a_1 \oplus b_1 \oplus c_1$ and $c_2 = (c_1 \wedge (a_1 \vee b_1)) \vee (a_1 \wedge b_1)$, and in general, for $k \geq 1$ we have

$$\begin{aligned} r_k &= a_k \oplus b_k \oplus c_k \\ c_k &= (c_{k-1} \wedge (a_k \vee b_k)) \vee (a_k \wedge b_k) \end{aligned}$$

Certainly, the above operation can be done in polynomial time. The main question is, can we do it faster in parallel? Note that the computation expressed above is sequential. Before computing $r_k$, one needs to compute all the previous output bits. Of course, one can always express the bits $r_k$ in CNF (or DNF) form. Such a circuit will have depth just 2 and hence, could add fast in parallel. But this argument via CNF (or DNF) would give exponential size circuit in general.

It turns out that there is indeed a way to do this in fewer levels (i.e. low depth or fast parallel time) and polynomial size. This is usually called *Carry-Look-Ahead.* In computing $r_k = a_k \oplus b_k \oplus c_k$, the main problem is computing $c_k$ and what we would like is to be able to compute the carry bit $c_k$ in one step for any $k$. Notice that for $c_1$ to be 1, we must have $a_0 = b_0 = 1$. In general, for $c_k$ to be 1, a carry bit has to be generated at some position $i < k$, and should be propagated all the way through to position $k$. To be precise,

$$c_k = \exists i(0 \leq i < k) \left[a_i \wedge b_i \bigwedge \forall j(i < j < k)(a_j \vee b_j)\right]$$

which can be rewritten as

$$c_k = \bigvee_{0 \leq i < k} \left[a_i \wedge b_i \wedge \bigwedge_{i<j<k} (a_j \vee b_j)\right]$$

Notice that this computation can be done in constant depth since nothing in it depends on previous results. To do the above computation in constant depth, of course, we would need gates with arbitrary fan-in.

## 10.2 NC and AC Classes

We next define the classes $\mathrm{AC}^i$ and $\mathrm{NC}^i$. First, we define $\mathrm{AC}^0$.

**Definition.** $\mathrm{AC}^0$ is the set of languages that have constant depth, polynomial size circuits (unbounded fan-in). Formally, $L \in \mathrm{AC}^0$ if for every $n$, there exists a circuit $C_n$ with unbounded fan-in gates AND, OR and NOT, such that

1. $\forall x \in \{0,1\}^n, C_n(x) = L(x)$
2. $|C_n| < n^{O(1)}$
3. $\mathrm{depth}(C_n) = O(1)$

We showed that addition is in $\mathrm{AC}^0$ (formally, we should define a language version of addition, for example, the $i$th bit of the sum). We say that a language is in *uniform* $\mathrm{AC}^0$ if the *circuit* $C_n$ is computable in log-space. Meaning, there is a Turing machine, which given $1^n$ as input, outputs the circuit $C_n$ and the machine runs in space $O(\log n)$. Next, we generalize the notion to define $\mathrm{AC}^i$.

**Definition.** $\mathrm{AC}^i$ is the set of languages $L$ such that for every $n$, there exists a circuit $C_n$ with unbounded fan-in gates AND, OR and NOT, such that

1. $\forall x \in \{0,1\}^n, C_n(x) = L(x)$
2. $|C_n| < n^{O(1)}$
3. $\text{depth}(C_n) = O[(\log n)^i]$

AC$^i$ allows circuits to use gates with unbounded fan-in. The class NC$^i$ is similar to AC$^i$, except that only bounded fan-in gates are allowed by NC$^i$. More precisely, we require that each boolean $\vee$ and $\wedge$ is binary. The following containments (for uniform NC and AC classes) are easy to see:

$$\text{AC}^0 \subseteq \text{NC}^1 \subseteq \text{AC}^1 \subseteq \text{NC}^2 \subseteq \ldots \subseteq \text{P}$$

As with other hierarchies in complexity theory, we know very little about whether this hierarchy is strict or if it collapses. One thing we know is that AC$^0 \neq$ NC$^1$. This is because, one can do parity in NC$^1$, but not in AC$^0$. It is easy to see that we do parity in NC$^1$, by divide and conquer. Later, we will show that parity cannot be done in AC$^0$.

## 10.3 Multiplication

Now that we have shown binary addition to be in AC$^0$, we turn our attention to binary multiplication. Again, the *school method* involves multiplying one of the numbers by each of the digits of the other number and adding all the results to get the final result. But since we are doing binary multiplication, each of these digits is either a 1 or a 0. Thus, we either write the number (shifted appropriately) or not, and add all of these (at most $n$) numbers together. Thus, multiplication is nothing more than the addition of these $n$ numbers, each of at most $2n-1$ bits, as shown in the example below.

$$\begin{array}{ccccccc}
 & & & 1 & 0 & 0 & 1 \\
\times & & & 1 & 1 & 0 & 1 \\
\hline
 & & & 1 & 0 & 0 & 1 \\
 & 1 & 0 & 0 & 1 & & \\
1 & 0 & 0 & 1 & & & \\
\hline
1 & 1 & 1 & 0 & 1 & 0 & 1
\end{array}$$

So the question is how to add $m$ binary numbers, each of at most $m$ bits. To add these numbers quickly, we introduce a new trick called 3-2 trick. This trick is to take 3 numbers $a, b, c$, and output 2 new numbers $d, e$, such that the sum of the 2 new numbers is the same as the sum of the 3 input numbers: $a + b + c = d + e$. To do this, we imagine doing an addition at every bit position $i$ separately. Given $a_i, b_i$ and $c_i$, we can produce the sum as two bits, $uv \in \{00, 01, 10, 11\}$. The school method would have set the leading bit $u$ as the carry bit and propogated onward. Instead we write down $v$ as the $i$th bit of $d$, and $u$ as the $i+1$th bit of $e$, where $d$ and $e$ are the two output numbers. More precisely, for $0 \leq i < n$ in parallel, we take the three bits $a_i$, $b_i$, and $c_i$, and set $d_i = a_i \oplus b_i \oplus c_i$ and $e_{i+1} = (a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i)$.

Now we have converted the addition of three numbers to the addition of two numbers in constant depth with bounded fan-in circuit (which are sometimes called NC$^0$).

To make use of this for the addition of $m$ numbers, we apply divide and conquer. After $O(\log m)$ parallel rounds of the 3-2 tricks, we are left with two numbers to add, at which point we add them using the carry-look-ahead method. Thus, the parallel time (depth) to add the $m$ $m$-bit numbers can be done in NC$^1$.

Other operations that are also in NC$^1$ are integer division and inner product. Using the Chinese Remainder Theorem, it can be shown that division is in NC$^1$. (For division, the uniformity of the circuit had been a tricky issue. For many years one can only achieve P-uniformity; but recently it has been shown that it can be done in logspace-uniform NC$^1$.)

## 10.4 Inner Product, Matrix Powers and Triangular Linear Systems

We now turn our attention to matrix computations and solving linear systems of the form $Ax = b$. First, consider multiplying two $n \times n$ matrices $A$ and $B$. Each entry of the output matrix is an inner product of a row of $A$ and a column of $B$. All these can be carried out in parallel. Each inner product involves $n$ multiplications and then summing up the products. The multiplications can be carried out in parallel in NC$^1$. Summing up the $n$ products can be done in a "binary tree fashion" with $\log n$ levels. Thus matrix multiplication can be done in NC$^1$. How about computing $A^m$? Using repeated squaring, it is clear we need only $O(\log m)$ levels of matrix squaring and multiplying. In particular, for a $n \times n$ matrix $A$ with each entry an $O(n)$-bit integer, $A^{O(n)}$ can be computed in NC$^2$.

Our next goal is to solve linear equations. Recall that a linear system is of the form $Ax = b$ where $A$ is a matrix of coefficients, $b$ is a vector of constants and $x$ is a vector of unknowns. To simplify our discussion, suppose that $A$ is a non-singular $n \times n$ matrix.

We first consider the simplest case. Suppose $A$ is lower triangular with unit diagonals; that is, $A$ is of the form

$$A = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ a_{2,1} & 1 & 0 & \dots & 0 \\ a_{3,1} & a_{3,2} & 1 & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & 0 \\ a_{n,1} & a_{n,2} & a_{n,3} & \dots & a_{n,n} \end{pmatrix}$$

We first write $A = I - \Delta$ where $\Delta$ is strictly lowe triangular (i.e. has zeros along the diagonal as well as above the diagonal). Note that $\Delta^2$ has zeros in not only the upper triangular and diagonal entries, but also the diagonal just above the main diagonal. $\Delta$ is called a *nilpotent* matrix, in that for some $m$, $\Delta^m = 0$. In fact $\Delta^n = 0$. By analogy of geometric sereis, we write

$$(I - \Delta)^{-1} = \sum_{i=0}^{\infty} \Delta^i$$

As $\Delta^n = 0$, only the first $n$ terms of the above series are non-zero. So,

$$(I - \Delta)^{-1} = I + \Delta + \Delta^2 + \ldots \Delta^{n-1}$$

Of course, so far this argument is dubious, since it is arrived at by analogy. This argument can be made rigorous. However we will be theoretical computer scientist for once, a.k.a. mathematicians in a hurry, and justify this simply by verifying that the finite sum exhibited indeed is the inverse of $I - \Delta$, QED. Thus, if our linear system is such that $A$ is lower triangular, (as $A$ is non-singular, by dividing out the diagonal entries, we can assume it has unit diagonals), we can compute the inverse of $A$ by evaluating the above finite series. This involves only matrix multiplication and addition. Finally, the solution is given by $x = A^{-1}b$. This entire process is certainly in NC$^2$.

This process works great for lower triangular matrices, but we want to solve *any* linear system. We can do that using Gaussian elimination, which is certainly not parallel as each step depends on what was done in the previous step. We want to somehow reduce the problem of inverting general matrices to the problem of inverting lower triangular matrices, in parallel efficiently. We do that next.

## 10.5 Determinant and Linear Equations

We show how to solve general linear equations in NC$^2$.

The goal is to solve a given system of linear equations: $Ax = b$, where $A$ is a non-singular $n \times n$ matrix, $b$ an $n$ dimensional column vector and $x = (x_1, x_2, \ldots, x_n)$ is the vector of unknowns. Our aim is to compute the determinant of $A$ in NC$^2$. Once we do that, the solution $x$ can be found by using Cramer's rule: $x_i = \det(A_i)/\det(A)$, where $A_i$ is the matrix obtained by replacing the $i^{th}$ column of $A$ by the column vector $b$.

### 10.5.1 Trace of a matrix

*Trace* of an $n \times n$ matrix $A$, denoted $\mathrm{Tr}(A)$, is the sum of its diagonal entries. That is,

$$\mathrm{Tr}(A) = \sum_{i=0}^{n} a_{ii}$$

We next state some properties of trace.

Proposition 10.1 *For two $n \times n$ matrices $A$ and $B$, $\mathrm{Tr}(AB) = \mathrm{Tr}(BA)$.*

*Proof.*

$$\mathrm{Tr}(AB) = \sum_i \sum_k a_{ik} b_{ki} = \sum_k \sum_i a_{ik} b_{ki} = \sum_k \sum_i b_{ki} a_{ik} = \mathrm{Tr}(BA)$$

♣

The following lemma is well-known for a spectural decomposition of a matrix. More precise information of the matrix $J$ (called Jordan normal form) is known, but we will only need this form. We will use the lemma to establish more properties of trace and eigenvalues.

Lemma 10.2 *Let $A$ be an $n \times n$ matrix. There exist $n \times n$ matrices $T$ and $J$ such that $A = T^{-1}JT$, where $J$ is an upper triangular matrix whose $n$ diagonal entries are the $n$ eigenvalues of $A$.*

Proposition 10.3 *Let $A$ be an $n \times n$ matrix and $\lambda_1, \lambda_2, \ldots, \lambda_n$ be its eigenvalues. Then, for $k \geq 0$,*

$$\mathrm{Tr}(A^k) = \sum_{i=1}^{n} \lambda_i^k.$$

*Proof.* Let $A = T^{-1}JT$. Then we have

$$A^k = (T^{-1}JT)^k = T^{-1}J^kT$$

So,

$$\mathrm{Tr}(A^k) = \mathrm{Tr}(T^{-1}J^kT) = \mathrm{Tr}(T^{-1}TJ^k) = \mathrm{Tr}(J^k)$$

The second equality follows from Proposition 10.1. Note that $J$ is an upper triangular matrix whose $n$ diagonal elements are $\lambda_1, \lambda_2, \ldots, \lambda_n$. Hence, the diagonal elements of $J^k$ are $\lambda_1^k, \lambda_2^k, \ldots, \lambda_n^k$. So,

$$\mathrm{Tr}(J^k) = \sum_{i=1}^{n} \lambda_i^k$$

♣

Next, we can express the determinant of a matrix as the product of its eigenvalues.

Proposition 10.4 *For an $n \times n$ matrix $A$ with eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$, we have*

$$\det(A) = \prod_{i=1}^{n} \lambda_i$$

*Proof.* Consider the decomposition $A = T^{-1}JT$. Then,

$$\det(A) = \det(T^{-1}JT) = \det(J) = \prod_{i=1}^{n} \lambda_i$$

♣

From last section, for $1 \leq k \leq n$, we know how to compute $A^k$ in NC$^2$. Hence, we can also compute $\mathrm{Tr}(A^k) = \sum_{i=1}^{n} \lambda^k$. Notice that we have computed all these sums without explicitly computing $\lambda_1, \lambda_2, \ldots, \lambda_n$. Our aim is to compute $\det(A)$, the product of eigenvalues. One could explicitly compute these eigenvalues and then take their product. But, computing eigenvalues is at least as hard as computing the determinant. Instead, we will directly compute their product using the known traces $\mathrm{Tr}(A^k)$, $1 \leq k \leq n$.

Think of the eigenvalues as unknowns or variables. Then, $\mathrm{Tr}(A^k)$ and $\det(A)$ are polynomials over these variables. These polynomials are *symmetric polynomials* and we can express the polynomial $\det(A)$ in terms of the polynomials $\mathrm{Tr}(A^k)$. Towards that end, we shall diverge a little bit and study the symmetric polynomials.

### 10.5.2 Symmetric polynomials

We will focus on polynomials over $n$ variables $x_1, x_2, \ldots, x_n$. A polynomial over these variables is symmetric, if renaming the variables does not change the polynomial. Formally,

Definition 10.5 (Symmetric polynomial) *A polynomial $p$ on $x_1, x_2, \ldots, x_n$ is* symmetric *if $\forall \sigma \in S_n$ (i.e., for all possible permutations of 1 through $n$), $p(x_1, \ldots, x_n) = p(x_{\sigma_1}, x_{\sigma_2}, \ldots, x_{\sigma_n})$.*

Two families of symmetric polynomials are important for our purpose here: Newton's symmetric polynomials and elementary symmetric polynomials. We first give Newton's symmetric polynomials. For $0 \leq k \leq n$, the $k^{th}$ Newton's symmetric polynomial is given by

$$s_k = \sum_{i=1}^{n} x_i^k$$

It is clear that these polynomials are symmetric. We could define $s_k$ for $k > n$, but we will only need $0 \leq k \leq n$. Note that $s_0 = n$.

We now give the elementary symmetric polynomials. For $1 \leq k \leq n$, the $k^{th}$ elementary symmetric polynomial is given by

$$p_k = \sum_{1 \leq i_1 < i_2 < \ldots < i_k \leq n} x_{i_1} x_{i_2} \ldots x_{i_k}$$

Since every choice of $k$ variables appears as a product exactly once, these polynomials are symmetric. It is convenient to define $p_0 = 1$. Note that $p_n = \prod_i x_i$.

(Notice the correspondence between these polynomials and trace and determinant we discussed before. In particular, $\mathrm{Tr}(A^k) = \sum_{i=1}^n \lambda_i^k$ is the $k^{th}$ Newton's symmetric polynomial, whereas, $\det(A) = \prod_{i=1}^n \lambda_i$ is the $n^{th}$ elementary symmetric polynomial.)

One can express the elementary symmetric polynomials in terms of Newton's polynomials. For example,

$$p_1 = \sum_{i=1}^n x_i = s_1.$$

Similarly, since $s_2 = \sum_{i=1}^n x_i^2$,

$$p_2 = \sum_{1\le i<j\le n} x_i x_j.$$

Thus,

$$s_1^2 = \left(\sum x_i\right)^2 = s_2 + 2p_2.$$

Therefore,

$$p_2 = \frac{1}{2}(s_1^2 - s_2).$$

In fact any symmetric polynomial can be expressed as a polynomial of Newton's polynomials as well as expressed as a polynomial of elementary symmetric polynomials:

Theorem 10.6 (Fundamental theorem of symmetric polynomials) *Every polynomial on $x_1, \ldots, x_n$ is symmetric iff it can be expressed as a polynomial of $s_1, \ldots, s_n$ (as well as $p_1, \ldots, p_n$).*

We can use this theorem to relate the elementary symmetric polynomials and Newton's polynomials. But, we need more specific formulas relating the two families of polynomials. So, next we state and prove a relationship between these two families that is useful in computing determinants in NC$^2$.

Theorem 10.7 *For $1 \le k \le n$,*

$$p_k = \frac{1}{k}(p_{k-1}\cdot s_1 - p_{k-2}\cdot s_2 + \cdots \pm p_0 \cdot s_k)$$

*Proof.* For $k, m \ge 0$, define the polynomial:

$$f_k^m = \sum_{1\le i_1<\ldots<i_k\le n j\notin\{i_1,\ldots,i_k\}} x_{i_1}x_{i_2}\ldots x_{i_k}x_j^m$$

How many terms are there to sum over? Note that there are two distinct parts, the product $x_{i_1}x_{i_2}\ldots x_{i_k}$ and $x_j^m$. There are $\binom{n}{k}\cdot(n-k)$ terms. Clearly, the boundary cases are $f_0^m = \sum_{j=1}^n x_j^m = s_m$ and $f_k^0 = (n-k)\cdot p_k$.

We now derive a formula for $p_k \cdot s_m$, when $k \geq 1$ and $m \geq 0$:

$$\begin{aligned} p_k \cdot s_m &= \left(\sum_{1 \leq i_1 < \ldots < i_k \leq n} x_{i_1} x_{i_2} \ldots x_{i_k}\right) \cdot \left(\sum_{j=1}^{n} x_j^m\right) \\ &= \sum_{1 \leq i_1 < \ldots < i_k \leq n, j \notin \{i_1, \ldots, i_k\}} x_{i_1} x_{i_2} \ldots x_{i_k} x_j^m + \sum_{1 \leq i_1 < \ldots < i_k \leq n, j \in \{i_1, \ldots, i_k\}} x_{i_1} x_{i_2} \ldots x_{i_k} x_j^m \end{aligned}$$

The first sum is $f_k^m$. What about the second sum? It's a bit tricker, but a moment reflection convinces us that it is nothing but $f_{k-1}^{m+1}$, as each term in $f_{k-1}^{m+1}$ occurs eactly once in the second sum. So, for $k, m \geq 1$, we have

$$p_k \cdot s_m = f_k^m + f_{k-1}^{m+1}$$

For $k = 0$, since $p_0 = 1$, $p_0 \cdot s_m = s_m = f_0^m$.

Now consider the alternating sum:

$$\begin{aligned} & p_k \cdot s_0 - p_{k-1} \cdot s_1 + p_{k-2} \cdot s_2 - \ldots + (-1)^k p_0 \cdot s_k \\ = \; & (f_k^0 + f_{k-1}^1) - (f_{k-1}^1 + f_{k-2}^2) + \ldots + (-1)^{k-1}(f_1^{k-1} + f_0^k) + (-1)^k f_0^k \end{aligned}$$

This is a telescoping sum, where all the terms cancel out, except for $f_k^0$. By definition, $f_k^0 = (n-k)p_k$. Hence,

$$(n-k)p_k = p_k \cdot s_0 - p_{k-1} \cdot s_1 + p_{k-2} \cdot s_2 + \cdots + (-1)^k p_0 \cdot s_k$$

Note that $s_0 = n$ and $p_0 = 1$. So,

$$(-1)^{k-1} p_1 s_{k-1} + \cdots + p_{k-2} s_2 - p_{k-1} s_1 + k p_k = (-1)^{k-1} s_k.$$

♣

### 10.5.3 Csanky's Algorithm for Determinant

Recall that we are interested in computing the determinant of a given non-singular $n \times n$ matrix $A$. Let its eigenvalues be $\lambda_1, \lambda_2, \ldots, \lambda_n$. We can compute $S_k = \mathrm{Tr}(A^k)$ in $\mathrm{NC}^2$. Our goal is to compute,

$$\det(A) = P_n = \lambda_1 \lambda_2 \ldots \lambda_n$$

Think of the eigenvalues as variables or unknowns. Then, $S_k$ and $P_k$ can be viewed as polynomials. For each $1 \leq k \leq n$, $S_k$ is nothing but the $k^{th}$ Newton's symmetric polynomial and $P_k$ is nothing but the $k^{th}$ elementary symmetric polynomial. We have these equalities relating $P_k$ and $S_k$. Now, the idea is to treat $\{P_k\}$ as the unknowns and solve for $\{P_k\}$ as a system of linear equations. We can write

$$(-1)^{k-1} S_{k-1} P_1 + \cdots + S_2 P_{k-2} - S_1 P_{k-1} + k P_k = (-1)^{k-1} S_k.$$

Recall that we started off with the problem of solving linear system of equations. Have we made any progress? Yes! The crucial point is that, now the system is lower triangular! To be more pedantic, let us write the above system in matrix form:

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ -\frac{S_1}{2} & 1 & 0 & \cdots & 0 \\ \frac{S_2}{3} & -\frac{S_1}{3} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ \frac{(-1)^{n-1}S_{n-1}}{n} & \frac{(-1)^{n-2}S_{n-2}}{n} & \frac{(-1)^{n-3}S_{n-3}}{n} & \cdots & 1 \end{bmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ \vdots \\ P_n \end{pmatrix} = \begin{pmatrix} S_1 \\ -\frac{S_2}{2} \\ +\frac{S_3}{3} \\ \vdots \\ \frac{(-1)^{n-1}S_n}{n} \end{pmatrix}$$

We already saw how to solve system of linear equations whose coefficient matrix is lower triangular in $\mathrm{NC}^2$. Using that method, we can solve the above system of equations and compute $P_1, P_2, \ldots, P_n$. Finally, $P_n = \det(A)$ is the required answer.

# Chapter 11

# Circuit Lower Bounds

Chapter Outline: Razborov-Smolensky, Decision trees, Switching Lemma, Parity lower bounds, Non-approximability results for parity

## 11.1 Historical Notes

Discuss history of circuit lower bounds. Furst-Saxe-Sipser, Yao, Cai, Hastad.

## 11.2 Razborov-Smolensky Theorem

Constant depth size $2^{n^\epsilon}$ boolean circuits together with $\mathrm{Mod}\,3$ gates cannot compute parity.

We will prove the above result in next two sections. We first show that constant depth circuit can be approximated by a low degree polynomial. Then we show that such polynomials cannot approximate parity function.

### 11.2.1 Approximating Constant Depth Circuits by Low Degree Polynomials

In this section, we show that constant depth circuits can be approximated by low degree polynomials over GF(3).

Theorem 11.1 *We can approximate the function $F(x_1, x_2, \ldots, x_n) = x_1 \vee x_2 \vee \ldots \vee x_n$ with a low degree polynomial. More precisely, for any positive integer $k$, there is a degree $\leq 2k$ polynomial $P$ in $x_1, \ldots, x_n$, over GF(3) that agrees with $y$ on at least $2^n(1 - (\frac{1}{3})^k)$ of* *assignments. That is,*

$$Pr_{\sigma\in\{0,1\}^n}[P(\sigma) = F(\sigma)] \geq 1 - \left(\frac{1}{3}\right)^k$$

*The claim is also valid for the* $\bigwedge, \neg and \text{Mod}_3$ *functions.*

*Proof.* We use the probabilistic method to prove the lemma. Fix an assignment $\sigma_1 = (c_1, c_2, \ldots, c_n) \in \{0,1\}^n$. Choose $a_1, a_2, \ldots, a_n \in GF(3)$ uniformly at random. Consider the polynomial $\hat{F} = (\sum_{i=1}^n a_i x_i)^2)$. Note that, as we are working in GF(3)={-1,0,1}, $\hat{F}$ evaluates to 0 or 1 for any input. What is the probability that $\hat{F}(\sigma_1) = F(\sigma_1)$? If $F(\sigma_1)$ evaluates to 0, then $\hat{F}$ also evaluates to 0 (regardless of the random values $a_i$). On the other hand, suppose $F(\sigma_1) = 1$. Let $j$ be the largest index such that $c_j = 1$. Then

$$Pr[\hat{F}(\sigma_1) = 1] = Pr\left[\left(\sum_{i=1}^{n} a_i c_i\right)^2 = 1\right] = Pr\left[\left(\sum_{i=1}^{j} a_i c_i\right)^2 = 1\right] = Pr\left[\sum_{i=1}^{j} a_i c_i \neq 0\right]$$

The last equality is due to the fact that in GF(3), $x^2 = 1 \iff x \neq 0$. We evaluate the last quantity by conditioning on the first $j-1$ terms of the summation. These could sum up to 0,1 or -1. But regardless this value, we have 2/3 chance for the summation to be non-zero, because $c_j = 1$ and $a_j$ is chosen at random from GF(3). To conclude, for any fixed $\sigma_1 \in \{0,1\}^n$,

$$Pr_{a_1,a_2,\ldots,a_n\in GF(3)}[\hat{F}(\sigma_1) = F(\sigma_1)] \geq \frac{2}{3}$$

There are $3^n$ ways to choose the coefficients $a_i$'s of the polynomial and there are $2^n$ assignments for the variables $x_1, x_2, \ldots, x_n$. Consider the following table in which each row represents one possible vector of coefficients and each column represents a possible assignment.

| | Assignment of the $x_i$ | | | | |
|---|---|---|---|---|---|
| | $\sigma_1$ | $\sigma_2$ | $\sigma_3$ | $\cdots$ | $\sigma_{2^n}$ |
| $a^{(1)}$ | | | | | |
| $a^{(2)}$ | | | | | |
| $a^{(3)}$ | | | | | |
| $\vdots$ | | | | | |
| $a^{(3^n)}$ | | | | | |

Consider an assignment $\sigma_j = c_1, c_2, \ldots, c_n$ and a coefficient vector $a^{(i)} = (a_1, a_2, \ldots, a_n)$. Place a 1 in the cell $(a^{(i)}, \sigma_j)$ in the above table, if $(\sum_{j=1}^n a_i c^2_{)} = F(c_1, c_2, \ldots, c_n$, and place a 0 otherwise. By our probability calculation above, any column of the table has 1 in at least 2/3 of the cells. Thus, at least 2/3 of the entries in the entire table are 1. It follows that there is some row that has a 1 in at least 2/3 of its cells. Let this row be $r_1$ and let $\hat{F}_1$ be the polynomial corresponding to this row, i.e. $\hat{F}_1(x_1, x_2, \ldots, x_n) = \sum_{i=1}^n a_i^{(r_1)} x_i$. If we choose an assignment $\sigma$ at random, we have

$$Pr_{\sigma\in\{0,1\}^n}[\hat{F}_1(\sigma) = F(\sigma)] \geq \frac{2}{3}$$

Now ignore all the columns in which the row $r_1$ has 1. At most 1/3 of the columns remain. All these columns have 1 in at least 2/3 of their cells. Again by the same argument, there is some other row $r_2$ that has 1 in at least 2/3 of these remaining columns. We let $F_2$ be the polynomial corresponding to row $r_2$. Next ignore these columns. At most $(1/3)^2$ columns would remain. We continue like this for $k$ iterations. At most $(1/3)^k$ columns would remain. We would have constructed polynomials $\hat{F}_1, \hat{F}_2, \ldots, \hat{F}_k$.

Being the OR function, $F$ evaluates to 0 for only one assignment, namely $\sigma_0 = (x_1 = 0, x_2 = 0, \ldots, x_n = 0)$. All our $k$ polynomials evaluate to 0 on this assignment and for any other input they all evaluate to 0 or 1. Consider the columns (or assignments) that were ignored. These are "covered"by our polynomials in the sense that if $F(\sigma) = 1$ then at least one of these polynomials $F_i$ also evaluates to 1 on $\sigma$. So if we take OR of our polynomials $\hat{F} = \hat{F}_1 \vee \hat{F}_2 \vee \ldots \vee \hat{F}_k$, then for any "covered" assignment $\sigma$, $\hat{F}(\sigma) = F(\sigma)$. But $1 - (1/3)^k$ of the assignments are covered. So

$$Pr_{\sigma}[\hat{F}(\sigma) = F(\sigma)] \geq 1 - \left(\frac{1}{3}\right)^k$$

Only one problem remains. How do we OR two polynomials to get another polynomial? We use the trick mentioned in the beginning. We let $\hat{F}_1 \vee \hat{F}_2 = 1 - (1 - \hat{F}_1)(1 - \hat{F}_2)$. So we take OR of all the $k$ polynomials this way and let the required polynomial P to be $\hat{F}_1 \vee \hat{F}_2 \vee \ldots \vee \hat{F}_k$. What is the degree of P? Each $\hat{F}_i$ has degree 2 (because of the squaring). Thus $P$ has degree at most $2k$.

Thus we have constructed a polynomial of degree at most $2k$ that agrees with the OR function $F$ on at least $1 - (1/3)^k$ assignments. Same way we can approximate AND and NOT functions.

Next we show that any depth $d$ circuit $C$ of size $s$ over $n$ inputs, can be approximated by a low degree polynomials. We use the above lemma to approximate each of the $s$ gates by polynomials of degree $2k$. We construct a polynomial to approximate the circuit by combining these polynomials. For example, consider a two input $\vee$–gate $g$ that receives its inputs from two other gates $g_1$ and $g_2$ (which are at lower depth). The polynomial for gate $g$ constructed by Lemma 11.1 be $P(y_1, y_2)$. We substitute the polynomials for gates $g_1$ and $g_2$, say $F_{g_1}$ and $F_{g_2}$ for $y_1$ and $y_2$ in $P$ to get the polynomial $F_g$ for gate $g$. This way, we start at the input level and move upto the output level of the circuit to get the polynomial $F_C$ to approximate the circuit. As the depth of the circuit is $d$ and the polynomials for the gates (from Lemma 11.1 are of degree $2k$), the degree of $F_C$ would be $(2k)^d$.

Choose an input $x_1, x_2, \ldots, x_n \in \{0, 1\}$ at random. What is the probability that $F_C$ does not agree with circuit $C$ on this input? $F_C$ will fail to agree with the circuit $C$ only when at least one of the $s$ polynomials (of the gates) fails. By the guarantee given by Lemma 11.1, for any specific gate, this happens with probability at most $\frac{s}{3^k}$. Thus probability that at least one gate fails is at most $\frac{1}{3^k}$. Thus,

$$Pr_{x_1,x_2,\ldots,x_n \in \{0,1\}}[F_C(x_1,x_2,\ldots,x_n) = C(x_1,x_2,\ldots,x_n)] \geq 1 - \frac{s}{3^k}$$

We have shown that

Theorem 11.2 *For any circuit $C$ of depth $d$ and size $s$ on $n$ boolean variables, for any integer $k$, there is a polynomials $F(x_1,x_2,\ldots,x_n)$ of degree at most $(2k)^d$ over GF(3) whose value is equal to the output of the $C$ on at least $2^n(1-s/3^k)$ inputs. The circuit is allowed to use AND, OR, NOT and* $\text{Mod}_3$ *gates.*

### 11.2.2 Low Degree Polynomials Cannot Approximate Parity

## 11.3 Switching Lemma and Parity Lower Bounds

### 11.3.1 Decision Trees

Decision trees are used to represent boolean functions.

Definition 11.3 (Decision trees) *A* decision tree *is a rooted binary tree $t$ with every leaf labeled as 0 or 1. Every internal node is labeled with an $x_i$. The two edges from an internal node are labeled with* 0 *and* 1 *respectively. We do not disallow variables to repeat along a path from the root to the leaf.*

Definition 11.4 (Decision tree depth) *The* depth *of a decision tree is defined to be the number of edges in the longest path from the root to a leaf.*

Figure 11.1 shows an example of a decision tree.

A decision tree $T$ defines a boolean function as follows:

1. if depth$(T) = 0$, then it is a constant function (either 0 or 1)
2. if depth$(T) = d+1$ $(d \geq 0)$: inductively assume that the left subtree and the right subtree respectively define a boolean function. Call the boolean function defined by the left subtree $G$ and the right subtree as $H$. Also assume that the root is labeled as $x_i$ and has the left edge labeled 0 and the right edge labeled 1. Then,

$$F(x_1,x_2,\ldots,x_n) = \begin{cases} G(x_1,x_2,\ldots,x_n) & \text{if } x_i = 0 \\ H(x_1,x_2,\ldots,x_n) & \text{if } x_i = 1 \end{cases}$$

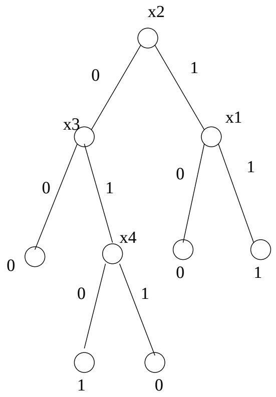


Figure 11.1: An example of a decision tree

Both $G$ and $H$ are also functions in $x_i$, but if we disallow $x_i$ from repeating along any path from root to leaf in the decision tree, we can leave it out of the functions $G$ and $H$. Then instead, we can write $G$ as a function of $x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$, which is often written as $G(x_1, x_2, \ldots \hat{x_i}, \ldots, x_n)$

Hence, the function $F$ is in fact:

$$F = (G \wedge \bar{x_i}) \vee (H \wedge x_i)$$

We can instead choose to write $G$ as $F|_{x_i=0}$ and $H$ as $F|_{x_i=1}$ and rewrite the above equation as:

$$F = (F|_{x_i=0} \wedge \bar{x_i}) \vee (F|_{x_i=1} \wedge x_i)$$

This expansion gives us a way to write the formula in DNF (i.e., $\vee$ of $\wedge$'s). If $F|_{x_i=0}$ and $F|_{x_i=1}$ can be written as a DNF where every conjunct has size at most $t$, then we get a formula $F$ in DNF in which every conjunct is of size at most $t+1$.

Definition 11.5 (Decision tree complexity) *For boolean function $f$, the decision tree complexity, $DC(f)$, is the minimum depth of the decision trees that compute $f$.*

Lemma 11.6 $D\left(f\right) = D\left(\bar{f}\right)$

Proof: The set of boolean decision trees that compute $f$ are in 1-1 correspondence with those that compute $\bar{f}$. (Just flip the bits at the leaves of the tree.) ♣

As demonstrated in the previous section, a boolean function computed by a decision tree of depth $d$ can be expressed as a boolean formula in disjunctive normal form (DNF), where each conjunct has size at most $d$. Using that idea, we get the following corollary.

Corollary 11.7 *A boolean function $f$ and its complement $\bar{f}$ can be expressed in DNF in which every conjunct has size $\leq DC(f)$. So $f$ can be written as a DNF formula with conjuncts of size $\leq DC(f)$, and as a CNF formula with disjuncts of size $\leq DC(f)$.*

The above mentioned special case of DNF where every conjunct is bounded in size will be useful in our discussion. The following definition spells this out.

Definition 11.8 ($t$-AND-OR) *Boolean formula $G$ is a $t$-AND-OR if $G = G_1 \wedge G_2 \wedge \cdots \wedge G_w$, where each $G_i$ is the OR of at most $t$ literals. So $G_i = \tilde{x}_1 \vee \tilde{x}_2 \vee \cdots \vee \tilde{x}_s$, for $s \leq t$, and $\tilde{x}_i \in \{x_i, \bar{x}_i\}$. (Likewise, a $t$-OR-AND is a formula in disjunctive normal form with at most $t$ terms in each conjunct.)*

### 11.3.2 Random Restrictions

Definition 11.9 (Restriction) *A restriction $\rho$ is a partial assignment to variables of boolean formula $G$. Specifically, it is a mapping $\rho : \{1, 2, \cdots, n\} \mapsto \{0, 1, *\}$. The restriction of $G$ by $\rho$, written $G|_\rho$, is the boolean function obtained by setting $x_i$ to $\rho(i)$ if $\rho(i) \in \{0, 1\}$ and leaving $x_i$ as a variable otherwise.*

A random $p$-restriction is a restriction $\rho$ where, for each $i$, one independently assigns $\rho(i)$ such that:

$$\begin{aligned} \Pr[\rho(i) = *] = p, \quad \text{and} & \\ \Pr[\rho(i) = 0] = \Pr[\rho(i) = 1] &= \frac{1-p}{2} \end{aligned}$$

### 11.3.3 Proving Circuit Lower Bounds for Parity: Overview

Here we will present a high level overview of proving lower bounds relating depth and size of circuits computing parity. A formal proof will be presented in subsequent lectures.

Before proceeding, let us review some conventions. Suppose we have a circuit that uses negation gates. By using de Morgan's law, we can push down the negation gates until they reach the input level. Thus, any boolean circuit can be transformed to an equivalent one such that all the negation gates occur at the bottom level (i.e. any negation gate takes it input only from variables). This transformation can be accomplished increasing the size at most twice the original and without any increase in depth. We assume that the input consists of both the variables and their negations so that size of a circuit refers to the number of AND and OR gates. We do not restrict fan-in or fan-out of the gates. So, without loss of generality, we can assume that the circuit is "leveled": a circuit of depth $d$ is made of $d$ levels so that all gates in a level are of same type and edges are only between adjacent levels. Thus, AND and OR gates will alternate across levels. From now on, we consider only circuits with above properties. By convention, output level (made of just one gate) is numbered 1 and the bottom level (where gates take their input from variables) is numbered $d$.

Now we are ready to state the the lower bound theorem.

THEOREM 11.10 *For sufficiently large $n$,* $\text{Parity}_n$ *cannot be computed by a depth-$d$ circuit of size $\leq 2^{cn^{1/d}}$, where $c > 0$ is some constant.*

One can prove the theorem with $c = 0.143781$. Here we will prove a weaker version where $c \approx 0.1$. The main ingredient of the proof is the switching lemma, which is as follows.

LEMMA 11.11 *[Switching Lemma] Let $G$ be a $t$-And-Or formula. Let $\rho$ be a random $p$-restriction. Then, for all $\Delta \geq 0$,*

$$\Pr[DC(G|_\rho) > \Delta] \leq (5pt)^\Delta. \tag{11.1}$$

Assuming the switching lemma, we sketch a proof of Theorem 11.10. Set $p = \gamma_0/t$, where $\gamma_0 > 0$ is a suitable constant fixed later. We will apply the lemma with $\Delta = t$. Then, the lemma says that when we apply a random $p$-restriction to a $t$-AND-OR formula $G$, with probability at least $1 - (5\gamma_0)^t$, the resultant formula $G|_\rho$ has DC $\leq t$. As we noted before, a formula with DC $\leq t$, can be expressed as a $t$-OR-AND formula. Thus, with high probability, the $t$-AND-OR formula $G|_\rho$ can be "switched" into an equivalent $t$-OR-AND formula. Using this ability to "switch", we can prove Theorem 11.10.

We first consider some restricted type of circuits. A circuit $C$ of depth $d$ is said to be of type $C^d(s,t)$ if it satisfies the two conditions: i) the gates at the bottom-most level (i.e. input level) have fan-in $\leq t$; ii) number of gates at levels above the bottom level $\leq s$. Thus, we have a bottom fan-in condition (bfi), which restricts fan-in of the bottom level gates. The gates above the bottom level (or the "internal" gates) are allowed to have any fan-in. At the end, we will relax these conditions and prove the theorem for any $d$-depth circuit (this part is easy).

Assume, without loss of generality that, the gates at the bottom level are OR gates (if not consider $\neg C$). Each gate at one level above ($(d-1)^{th}$ level) is an AND gate which gets its input from the OR gates at level $d$. Think of each such AND gates and along with the OR gates as a small circuit. Any such small circuit $sc$ is computing a $t$-AND-OR formula, because of the bottom fan in condition. Choose a random $p$-restriction $\rho$ and consider the circuit $C|_\rho$. By the switching lemma, with probability $\geq 1-(5\gamma_0)^t$, the formula computed by the small circuit $sc$ (in $C|_\rho$) has DC $\leq t$. As we discussed, this formula is (also) a $t$-OR-AND formula. Thus, we can *replace* the AND-OR circuit $sc$ by an OR-AND circuit, where the AND gates have fan-in $\leq t$. Suppose there are $s_{d-1}$ AND gates at level $d-1$. Assume that we were lucky and our random restriction $\rho$ is good so that we are able to replace all these $s_{d-1}$ AND-OR circuits by $s_{d-1}$ OR-AND circuits with bottom fan-in $\leq t$. The resultant circuit is made of OR gates in level $d-1$, OR gates is level $d-2$ and AND gates in level $d$. We can now *merge* the levels $d-1$ and $d-2$ in $C|_\rho$ to get a circuit that has only $d-1$ levels. There are a few things to note. The new circuit may have more gates at its bottom level than the original circuit. But, number of internal gates in the new circuits is $s - s_{d-1} \leq s$. Thus, the number of internal gates didn't increase. Next, bottom fan-in of the new circuit is $\leq t$. The new circuit has only $d-1$ levels. To summarize, the new circuit is of type $C^{d-1}(s,t)$ and is equivalent to $C|_\rho$. Of course, the new circuit is not computing the same function as the original one. But, this is not an issue when it comes to the parity function. We will elaborate this shortly.

Now the idea is to apply the above process of switching $d-2$ times: we pick a random $p$-restriction $\rho_1$ and apply it to $C$, then pick a random $p$-restriction $\rho_2$ and apply it to $C|_{\rho_1}$ and so on. Suppose we were lucky in picking all these $d-2$ random restrictions. Then, we will be left with a circuit $C'$ of type $C^2(1,t)$.

Suppose the original circuit $C$ computes parity on $n$ bits. What does the new circuit $C'$ compute? After applying all the $d-2$ random restrictions, some of the $n$ variables would have been assigned 1 or 0 and the rest would remain as variables (i.e assigned *). Let the number of variables remaining be $N$ (which is a random variable). Observe that $C'$ computes the parity on these $N$ variables.

Suppose all the $d-2$ random restrictions were good and suppose $N > t$. Then, $C'$ is a circuit of type $C^2(1,t)$ that computes parity on $N > t$ variables. That is impossible. ($C'$ computes a formula $G_1 \wedge G_2 \wedge \ldots \wedge G_w$, where each $G_i$ is an OR of $\leq t$ literals. In particular, $G_1$ has some $a \leq t < N$ literals. Set all these literals to 0. Then output of $C'$ is 0. But, since $N > a$, $G_1$ does not include some variable $x$. Set all the variables except those in $G_1$ and $x$ to be 0. Now, set $x$ appropriately, so that the parity is 1. Thus, $C'$ does not compute parity of $N$ variables).

The rest of proof is to show that with non-zero probability two properties are satisfied: i) the $d-2$ random restrictions are good (i.e. they allow us to "switch" $t$-AND-OR circuits to $t$-OR-AND circuits in all instances) and ii) $N > t$. We first note that instead of picking $d-2$ random restrictions, we could equivalently pick just one random restriction. Suppose we have a formula $F$ we apply a random $p_1$-restriction followed by a random $p_2$-restriction. Observe that, equivalently, we can apply a random $(p_1 \cdot p_2)$-restriction. (Distribution of the boolean functions obtained in the two cases will be the same). In our scenario, instead of applying $d-2$ random $p$-restriction one by one, we can equivalently apply a single random $p^{d-2}$-restriction.

We have a circuit $C$ of type $C^d(s,t)$ and a random $p^{d-2}$-restriction $\rho$. We need an estimate on the probability that $C|_\rho$ can be transformed (by switching) into a $C^2(1,t)$ circuit. As $C$ has $s$ internal gates, we will need do $s$ many "switches". The probability of failing in any one of these instances is bounded $\leq (5pt)^\Delta = (5\gamma_0)^t$. Thus, probability that we fail to convert $C|_\rho$ into an equivalent $C^2(1,t)$ circuit is $\leq s \cdot (5\gamma_0)^t$. Formally,

$$\Pr[C|_\rho \text{ is not equivalent to a } C^2(1,t) \text{ circuit}] \leq s \cdot (5\gamma_0)^t \tag{11.2}$$

Now we estimate the random variable $N$, the number of input variables of $C|_\rho$. $E[N] = n \cdot p^{d-2} = n \cdot (\gamma_0/t)^{d-2}$.

We will fix $\gamma_0$ and $t$ to obtain a bound on $E[N]$ and the RHS of ( 11.2). Set $\gamma_0 = 1/10$ and $t = \gamma_0 n^{1/(d-1)}$ Then,

$$\Pr[C|_\rho \text{ is not equivalent to a } C^2(1,t) \text{ circuit}] \leq s \cdot (5\gamma_0)^t = s \cdot 2^{-(0.1)n^{\frac{1}{(d-1)}}}.$$

We have $E[N] = n^{1/(d-1)}$. Then, using Chernoff bound, we get

$$\Pr[N < t] = \Pr[N < \gamma_0 \cdot E[N]] < e^{-\frac{(1-\gamma_0)^2}{2} \cdot n^{\frac{1}{d-1}}} < e^{-(0.4)n^{\frac{1}{d-1}}}$$

Let $c < 0.1$ be a constant. For sufficiently large $n$, if $s \leq 2^{-c \cdot n^{1/(d-1)}}$, then

$$s \cdot 2^{-(0.1)n^{\frac{1}{(d-1)}}} + e^{-(0.4)n^{\frac{1}{d-1}}} < 1.$$

Let us summarize. Let $s < 2^{-c \cdot n^{1/(d-1)}}$ and $C$ be a circuit of type $C^d(s,t)$ that computes parity of $n$ bits. Then, with non-zero probability, $C|_\rho$ can be transoformed into a $C^2(1,t)$ circuit that computes parity on $N > t$ bits. In particular, there exists a random restriction $\rho_0$ that satisfies both these properties. But, a circuit of type $C^2(1,t)$ cannot compute parity on $> t$ bits. A contradiction. We have proved the following theorem.

Theorem 11.12 *For $s \leq 2^{-c \cdot n^{1/(d-1)}}$, circuits of type $C^d(s,t)$ cannot compute parity on $n$ bits. Here, $c$ can be any constant* $< 0.1$.

It is now easy to prove Theorem 11.10. Observe that a circuit of size $s$ and depth $d$ can be viewed as a circuit of type $C^{d+1}(s,1)$. Then use Theorem 11.12. Later, we will prove a better version of the switching lemma, do a careful analysis and improve the contant $c$ from 0.1 to 0.143781.

### 11.3.4 Switching Lemma: Proof

In this lecture, we prove the switching lemma.

Lemma 11.13 *Let $G$ be a $t$-And-Or formula $G_1 \wedge G_2 \wedge \ldots \wedge G_w$. Let $\rho$ be a random $p$-restriction. Then, for all $\Delta \geq 0$,*

$$\Pr[\ \mathrm{DC}(G|_\rho) \geq \Delta\ ] \ \leq\ (5pt)^\Delta. \tag{11.3}$$

LEMMA 11.14 *Let $G$ be a t-And-Or formula $G_1 \wedge G_2 \wedge \ldots \wedge G_w$. For any $\beta$, $0 < \beta < t$, let $\rho$ be a random p-restriction, where $p = \frac{\beta}{t-\beta}$, and let $\alpha = \beta/ln\left[\frac{1+\sqrt{1+4e^{\beta}}}{2}\right]$. Then for all $\Delta \geq 0$, we have*

$$\Pr[DC(G|\rho) \geq \Delta] \leq \alpha^{\Delta}.$$

*Proof.* We prove these two lemmas together. We will prove the following stronger claim:

For *any* boolean function $F$,

$$\Pr[\ \mathrm{DC}(G|_\rho) \geq \Delta \mid F|_\rho \equiv 1\ ] \ \leq\ \alpha^{\Delta}, \tag{11.4}$$

where $\alpha$ will be set to $5pt$. (If the condition is not satisfied, the conditional probability is defined to be 0.) Lemma 11.13 follows by taking $F$ to be the constant function 1.

The intuition is as follows. We will assign some random restriction $\rho$ with parameter $p$. Note that $p$ is $\ll t$, the bfi. Thus, one expect for a conjunct, say $G_1$, the number of $*$ left in $G_1$ after this $\rho$ is 0, i.e., all the variables are assigned one way or another. Since we assign it with $\pm$ with equal probability, and since $G_1$ is an OR, we expect one of the variables is assigned in the right way so that $G_1 \equiv 1$. Thus, $G_1 \equiv 1$ is a low information event. We will "carry" this information along and proceed to $G_2$. It is the "rare" event that some $x_i$ in $G_1$ which gets a $*$ that we want to charge a price (in probability) of $p$.

As we move along in this process, we will be carrying more and more events of the form: over a collection subsets of literals, each subset has at least one literal which gets assigned in a particular way (0 or 1). The general form of such a condition is no less generic than saying a certain AND of OR's which is evaluated to true. But that is nothing but the condition that an arbitrary boolean function being true under the restriction. This is the "information" accumulated so far in the process of dealing with $G_i$ one by one.

Note that, at least intuitively, given some arbitrary boolean function $F$ being true under the restriction $\rho$, (i.e., $F|_\rho \equiv 1$), does not reduce the probability $p$ that any variable receiving $*$. Now we prove the theorem formally.

The statement (11.4) is trivially true for $\Delta = 0$, since the RHS becomes 1 in this case. Thus, we may assume $\Delta > 0$. We prove (11.4) by induction on $w$, the number of clauses conjuncted to form G. The base case of $w = 0$ is trivial, since $G \equiv 1$ by definition and the statement holds since the LHS is 0. By induction, assume that (11.4) holds for all conjuncts of up to $w - 1$ clauses.

Let $G$ be a $t$-And-Or formula $G_1 \wedge G_2 \wedge \ldots \wedge G_w$. Let $G' = G_2 \wedge \ldots \wedge G_w$. Let $\rho$ be a random $p$-restriction. By renaming literals, we can assume WLOG that $G_1 = \bigvee_{i \in T} x_i$ for some $T$ with $|T| \leq t$.

To prove the claim, we consider two cases $G_1|_\rho \equiv 1$ and $G_1|_\rho \not\equiv 1$. The case of $G_1|_\rho \equiv 1$ is easy. $G_1|_\rho \equiv 1$ implies that $G|_\rho \equiv G'|_\rho$. So, by induction

$$\Pr[\ \mathrm{DC}(G|_\rho) \geq \Delta \mid F|_\rho \equiv 1, G_1|_\rho \equiv 1\ ] \ =\ \Pr[\ \mathrm{DC}(G'|_\rho) \geq \Delta \mid (F \wedge G_1)|_\rho \equiv 1\ ] \ \leq\ \alpha^{\Delta}.$$

Rest of the proof deals with the case $G_1|_\rho \not\equiv 1$. That is, we want to prove the following statement as well.

$$\Pr[\ \mathrm{DC}(G|_\rho) \geq \Delta \mid F|_\rho \equiv 1, G_1|_\rho \not\equiv 1\ ] \ \leq\ \alpha^\Delta \tag{11.5}$$

We have renamed the variables so that $G_1 = \bigvee_{i \in T} x_i$. Then $G_1|_\rho \not\equiv 1$ means that for each $i \in T$, $\rho(i) = 0$ or $*$. For each $\rho$ such that $G_1|_\rho \not\equiv 1$, consider the set of all $i \in T$ such that $\rho(i) = *$. Moreover, if $\rho(i) = 0$ for all $i \in T$, then $G_1|_\rho \equiv 0$ and hence, $G|_\rho \equiv 0$ implying $DC(G) = 0$. Thus, it is not the case that $\rho(i) = 0$ for all $i \in T$. So,

$$\begin{aligned} &\{\rho\ :\ F|_\rho \equiv 1, G_1|_\rho \not\equiv 1, \mathrm{DC}(G|_\rho) \geq \Delta\} \\ &= \bigcup_{\emptyset \neq Y \subseteq T} \{\rho\ :\ \rho(Y) = *, \rho(T - Y) = 0, F|_\rho \equiv 1, \mathrm{DC}(G|_\rho) \geq \Delta\,\}. \end{aligned} \tag{11.6}$$

Let

$$a_Y \ = \ \Pr[\ \rho(Y) = * \wedge \rho(T - Y) = 0 \mid F|_\rho \equiv 1 \wedge G_1|_\rho \not\equiv 1\ ]. \tag{11.7}$$

So, (11.8) can be written as

$$\begin{aligned} &\Pr[\ \mathrm{DC}(G|_\rho) \geq \Delta \mid F|_\rho \equiv 1, G_1|_\rho \not\equiv 1\ ] \\ &= \sum_{\emptyset \neq Y \subseteq T} a_Y \cdot \Pr[\ \mathrm{DC}(G|_\rho) \geq \Delta \mid F|_\rho \equiv 1 \wedge G_1|_\rho \not\equiv 1 \wedge \rho(Y) = * \wedge \rho(T - Y) = 0] \\ &= \sum_{\emptyset \neq Y \subseteq T} a_Y \cdot \Pr[\ \mathrm{DC}(G|_\rho) \geq \Delta \mid F|_\rho \equiv 1 \wedge \rho(Y) = * \wedge \rho(T - Y) = 0] \end{aligned} \tag{11.8}$$

We will bound this sum from above by considering the individual terms. First consider the case of a fixed $|Y| < \Delta$.

Fix $\rho$ such that $\mathrm{DC}(G|_\rho) \geq \Delta$ and $F|_\rho \equiv 1 \wedge \rho(Y) = * \wedge \rho(T - Y) = 0$. We claim that there exists $\sigma_Y : Y \to \{0, 1\}$, such that $\mathrm{DC}(G|_\rho|_{\sigma_Y}) \geq \Delta - |Y|$.

Suppose not. Then for all $\sigma_Y : Y \to \{0, 1\}$, $\mathrm{DC}(G|_\rho|_{\sigma_Y}) < \Delta - |Y|$. So we can construct a decision tree for $G|_\rho$ as follows. Start with a full binary tree of depth $|Y|$ in which each level determines the value of $x_i$ for some $i \in Y$. Now let each leaf of this full binary tree, which represents a restriction $\sigma_Y$, be the root of a decision tree for $G|_\rho|_{\sigma_Y}$ of depth $\leq \Delta - |Y|$. The result is a decision tree for $G|_\rho$ of depth $\leq \Delta$. So $\mathrm{DC}(G|_\rho) < \Delta$, which is a contradiction.

We conclude from this contradiction that such a $\sigma_Y$ exists. We also note that $\sigma_Y \neq 0^Y$. For if $\sigma_Y = 0^Y$, then $G_1|_\rho|_{0^Y} \equiv 0$, so $G|_\rho|_{0^Y} \equiv 0$, so $\mathrm{DC}(G|_\rho|_{0^Y}) = 0$.

Since $\sigma_Y \neq 0^Y$, we have $G_1|_\rho|_{\sigma_Y} \equiv 1$, so $G|_\rho|_{\sigma_Y} \equiv G'|_\rho|_{\sigma_Y}$ and $\mathrm{DC}(G'|_\rho|_{\sigma_Y}) = \mathrm{DC}(G|_\rho|_{\sigma_Y}) \geq \Delta - |Y|$.

Now, again treating $\rho$ as a random $p$-restriction, we have

$$\Pr[\ \mathrm{DC}(G|_\rho) \geq \Delta \mid F|_\rho \equiv 1 \wedge \rho(Y) = * \wedge \rho(T - Y) = 0]$$

$$\begin{array}{ll}\leq & \Pr[\exists \sigma_Y : Y \to \{0,1\},\ \mathrm{DC}(G'|_\rho|_{\sigma_Y}) \geq \Delta - |Y| \mid F|_\rho \equiv 1 \wedge \rho(Y) = * \wedge \rho(T-Y) = 0] \\ \leq & \displaystyle\sum_{\substack{\sigma_Y : Y \to \{0,1\} \\ \sigma_Y \neq 0^Y}} \Pr[\ \mathrm{DC}(G'|_\rho|_{\sigma_Y}) \geq \Delta - |Y| \mid F|_\rho \equiv 1 \wedge \rho(Y) = * \wedge \rho(T-Y) = 0]\end{array} \tag{11.9}$$

$G'$ is a formula with $w-1$ clauses. We would like to invoke the induction hypothesis to bound the above sum. But, the condtion "$F|_\rho \equiv 1 \wedge \rho(Y) = * \wedge \rho(T-Y) = 0$" does not fit the induction hypothesis. We can overcome this as follows. Suppose $\rho(Y) = * \wedge \rho(T-Y) = 0$. Then, $F|_\rho \equiv 1$ iff for any (partial) truth assignment $\tau_Y : Y \to \{0,1\}$, we have $F|_\rho|\tau_Y \equiv 1$. To express this formally, set

$$\begin{array}{lcl} 0^{T-Y} & = & \text{the all 0 assignment on } T-Y, \\ \widetilde{F} & = & \bigwedge_{\tau_Y : Y \to \{0,1\}} F|_{0^{T-Y}}|_{\tau_Y} \ \text{ and} \\ \widetilde{\rho} & = & \rho \text{ restricted to the complement of } T. \end{array}$$

Then whenever $\rho(Y) = * \wedge \rho(T-Y) = 0$, as in the condition on the probabilities in (11.9),

$$\widetilde{F}|_{\widetilde{\rho}} \equiv 1 \iff F|_\rho \equiv 1$$

Again, suppose $\rho(Y) = *$ and $\rho(T-Y) = 0$. Then, for any $\sigma_Y : Y \to \{0,1\}$, we have $G'|_\rho|\sigma_Y$ is equivalent to $(G'|_{0^{T-Y}}|_{\sigma_Y})|_{\widetilde{\rho}}$. Thus,

$$\begin{array}{l} \Pr[\ \mathrm{DC}(G'|_\rho|_{\sigma_Y}) \geq \Delta - |Y| \mid F|_\rho \equiv 1 \wedge \rho(Y) = * \wedge \rho(T-Y) = 0] \\ \quad \leq \ \Pr[\ \mathrm{DC}((G'|_{0^{T-Y}}|_{\sigma_Y})|_{\widetilde{\rho}}) \geq \Delta - |Y| \mid \widetilde{F}|_{\widetilde{\rho}} \equiv 1] \end{array}$$

As $G'|_{0^{T-Y}}|_{\sigma_Y}$ has (at most) $w-1$ clauses, we can apply induction hypothesis and get

$$\Pr[\ \mathrm{DC}((G'|_{0^{T-Y}}|_{\sigma_Y})|_{\widetilde{\rho}}) \geq \Delta - |Y| \mid \widetilde{F}|_{\widetilde{\rho}} \equiv 1] \leq \alpha^{\Delta - |Y|}$$

So (11.9) becomes

$$\Pr[\ \mathrm{DC}(G|_\rho) \geq \Delta \mid F|_\rho \equiv 1 \wedge \rho(Y) = * \wedge \rho(T-Y) = 0] \ \leq \ (2^{|Y|} - 1)\alpha^{\Delta - |Y|} \tag{11.10}$$

We have proved the above bound for $|Y| < \Delta$. Now consider the case $|Y| \geq \Delta$. Here (11.10) holds trivially, because $|Y| \geq \Delta > 0$ and $\alpha < 1$ imply $(2^{|Y|} - 1)\alpha^{\Delta - |Y|} \geq 1$.

So (11.8) becomes

$$\Pr[\ \mathrm{DC}(G|_\rho) \geq \Delta \mid F|_\rho \equiv 1, G_1|_\rho \not\equiv 1\ ] \ \leq \ \sum_{\emptyset \neq Y \subseteq T} a_Y \cdot (2^{|Y|} - 1)\alpha^{\Delta - |Y|} \tag{11.11}$$

Let

$$b_Y \ = \ \Pr[\ \rho(Y) = * \mid F|_\rho \equiv 1 \wedge G_1|_\rho \not\equiv 1\ ]. \tag{11.12}$$

Clearly $b_Y \geq a_Y$. Also, we see intuitively that $Z \subseteq T$, $b_Z = \Pr[\ \rho(Z) = * \mid F|_\rho \equiv 1 \wedge G_1|_\rho \not\equiv 1\ ] \leq q^{|Z|}$, where $q = p/(p + \frac{1-p}{2})$ is the probability of $\rho$ assigning any variable to be $*$ given that it is assigned to be either $*$ or 0. For we already saw that $G_1|_\rho \not\equiv 1$ means that each variable in $Z$ is assigned either 0 or $*$. The additional condition that $F|_\rho \equiv 1$ can only decrease the probability that some variable is assigned a $*$. We will give a rigorous proof that $b_Z \leq q^{|Z|}$ at the end of these notes.

It follows that

$$\begin{aligned}
\Pr[\ \mathrm{DC}(G|_\rho) \geq \Delta \mid F|_\rho \equiv 1, G_1|_\rho \not\equiv 1\ ] &\leq \sum_{\emptyset \neq Y \subseteq T} a_Y \cdot (2^{|Y|} - 1)\alpha^{\Delta - |Y|} \\
&= \sum_{Y \subseteq T} a_Y \cdot (2^{|Y|} - 1)\alpha^{\Delta - |Y|} \qquad (11.13) \\
&\leq \sum_{Y \subseteq T} b_Y \cdot (2^{|Y|} - 1)\alpha^{\Delta - |Y|} \\
&\leq \sum_{Y \subseteq T} q^{|Y|} \cdot (2^{|Y|} - 1)\alpha^{\Delta - |Y|} \\
&\leq \alpha^\Delta \left[ \left( \sum_{Y \subseteq T} \left( \frac{2q}{\alpha} \right)^{|Y|} \right) - \left( \sum_{Y \subseteq T} \left( \frac{q}{\alpha} \right)^{|Y|} \right) \right] \\
&= \alpha^\Delta \left[ \left( 1 + \frac{2q}{\alpha} \right)^{|T|} - \left( 1 + \frac{q}{\alpha} \right)^{|T|} \right] \\
& \qquad (11.14)
\end{aligned}$$

The last equality in the above derivation is obtained using the identity, $\sum_{X \subseteq A} x^{|X|} = (1 + x)^{|A|}$.

If we set $c = 1/\log_e \phi \approx 2.078$, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the golden ratio, then we have $e^{2/c} - e^{1/c} = \phi^2 - \phi = 1$. Then, setting $\alpha = cqt < 5pt$, we get

$$\left( 1 + \frac{2q}{\alpha} \right)^t - \left( 1 + \frac{q}{\alpha} \right)^t = \left( 1 + \frac{2}{ct} \right)^t - \left( 1 + \frac{1}{ct} \right)^t < e^{2/c} - e^{1/c} = 1. \qquad (11.15)$$

since $(1 + \frac{\lambda}{t})^t < e^\lambda$ for all $t$ (to see this, compare (term by term) the binomial expansion of LHS and Taylor series of RHS).

We have shown that

$$\Pr[\ \mathrm{DC}(G|_\rho) \geq \Delta \mid F|_\rho \equiv 1, G_1|_\rho \not\equiv 1\ ] \ < \ \alpha^\Delta.$$

This completes the proof of

$$\Pr[\ \mathrm{DC}(G|_\rho) \geq \Delta \mid F|_\rho \equiv 1\ ] \ < \ (5pt)^\Delta.$$

♣

Now we prove the tighter bound for Lemma 11.14. Note that

$$b_Y \;=\; \sum_{Y \subseteq Z \subseteq T} a_Z,$$

and by the Möbius Inversion Formula,

$$a_Y \;=\; \sum_{Y \subseteq Z \subseteq T} (-1)^{|Z-Y|} b_Z$$

Substituting $\sum_{Y \subseteq Z \subseteq T} (-1)^{|Z-Y|} b_Z$ for $a_Y$ in (11.13), we have

$$\begin{aligned} &\Pr[\ \mathrm{DC}(G|_\rho) \geq \Delta \mid F|_\rho \equiv 1, G_1|_\rho \not\equiv 1\ ] \\ &\leq \sum_{Y \subseteq T} \sum_{Y \subseteq Z \subseteq T} (-1)^{|Z-Y|} b_Z \cdot (2^{|Y|} - 1)\alpha^{\Delta - |Y|} \\ &= \sum_{Z \subseteq T} b_Z (-1)^{|Z|} \alpha^{\Delta} \sum_{Y \subseteq Z} \left[ \left( \frac{-2}{\alpha} \right)^{|Y|} - \left( \frac{-1}{\alpha} \right)^{|Y|} \right] \end{aligned}$$

Therefore

$$\begin{aligned} &\Pr[\ \mathrm{DC}(G|_\rho) \geq \Delta \mid F|_\rho \equiv 1, G_1|_\rho \not\equiv 1\ ] \\ &\leq \alpha^{\Delta} \sum_{Z \subseteq T} b_Z (-1)^{|Z|} \left[ \left(1 - \frac{2}{\alpha}\right)^{|Z|} - \left(1 - \frac{1}{\alpha}\right)^{|Z|} \right] \\ &= \alpha^{\Delta} \sum_{Z \subseteq T} b_Z \left[ \left(\frac{2}{\alpha} - 1\right)^{|Z|} - \left(\frac{1}{\alpha} - 1\right)^{|Z|} \right] \\ &\leq \alpha^{\Delta} \sum_{Z \subseteq T} q^{|Z|} \left[ \left(\frac{2}{\alpha} - 1\right)^{|Z|} - \left(\frac{1}{\alpha} - 1\right)^{|Z|} \right] \\ &= \alpha^{\Delta} \left\{ \left[1 + q\left(\frac{2}{\alpha} - 1\right)\right]^{|T|} - \left[1 + q\left(\frac{1}{\alpha} - 1\right)\right]^{|T|} \right\} \\ &\leq \alpha^{\Delta} \left\{ \left[1 + q\left(\frac{2}{\alpha} - 1\right)\right]^{t} - \left[1 + q\left(\frac{1}{\alpha} - 1\right)\right]^{t} \right\} \end{aligned}$$

Now, we want to choose an appropriate $\alpha$ so that the quantity in curly braces above is $\leq 1$. Set $q = \beta/t$. Then,

$$\left[1 + \Big(\frac{2}{\alpha} - 1\Big)\frac{\beta}{t}\right]^t - \left[1 + \Big(\frac{1}{\alpha} - 1\Big)\frac{\beta}{t}\right]^t < e^{\left(\frac{2}{\alpha}-1\right)\beta} - e^{\left(\frac{1}{\alpha}-1\right)\beta}$$

So, we want to maintain $e^{(2/\alpha - 1)\beta} - e^{(1/\alpha - 1)\beta} < 1$. Solving this equation, we get

$$\alpha = \frac{\beta}{ln\left[\frac{1+\sqrt{1+4e^{\beta}}}{2}\right]}. \tag{11.16}$$

This completes proof of Lemma 11.14. ♣

Now we will show that for any $Z \subseteq T$, $b_Z = \Pr[\ \rho(Z) = * \mid F|_\rho \equiv 1 \wedge G_1|_\rho \not\equiv 1\ ] \leq q^{|Z|}$, where $q = p/(p + \frac{1-p}{2})$. Before we proceed, note that for $Z \subseteq T$, we have

$$\Pr[\ \rho(Z) = * \mid G_1|_\rho \not\equiv 1\ ] \ = \ q^{|Z|},$$

since $G_1|_\rho \not\equiv 1$ means that $\rho$ assigns only $*$ or 0 on $T$.

We now show that

$$b_Z \leq \Pr[\ \rho(Z) = * \mid G_1|_\rho \not\equiv 1\ ]$$

This is trivial if $Z = \emptyset$. Suppose $Z \neq \emptyset$.

Note that $b_Z \leq \max_{\rho'} \Pr[\ \rho(Z) = * \mid F|_\rho \equiv 1, G_1|_\rho \not\equiv 1, \rho|_{Z^c} = \rho'\ ]$.

Consider any fixed restriction $\rho'$ on the complement of $Z$, $\rho' : Z^c \rightarrow \{0, 1, *\}$. Then, there is a unique extension of $\rho'$ over $Z$, call it $\rho^*$, that satisfies $\rho^*(Z) = *$.

We claim that

$$\Pr[\ \rho(Z) = * \mid F|_\rho \equiv 1, G_1|_\rho \not\equiv 1, \rho|_{Z^c} = \rho'\ ] \ \leq \ q^{|Z|}.$$

The event $\rho(Z) = *$ refers to the unique $\rho^*$, under the condition $\rho|_{Z^c} = \rho'$. If $F|_{\rho^*} \equiv 1$ then $F|_\rho \equiv 1$ for all extensions $\rho$ of $\rho'$ to $Z$. Hence $F|_\rho \equiv 1, G_1|_\rho \not\equiv 1, \rho|_{Z^c} = \rho'$ refers to the $2^{|Z|}$ assignments $\rho$, such that $\rho(i) \in \{0, *\}$ for all $i \in Z$. Out of these $2^{|Z|}$ assignments, $\rho(Z) = *$ occurs with probability $q^{|Z|}$. On the other hand, if $F|_{\rho^*} \not\equiv 1$, then the above conditional probability is 0 and the claim trivially holds. Hence, the claim is true and $b_Z \leq q^{|Z|}$. ♣

### 11.3.5 Switching Lemma: Improved Lower Bounds

We now improve the bounds in switching lemma. Using the lemma, we establish lower bounds on size of circuits computing or approximating parity.

In the last lecture we proved the following lemma.

LEMMA 11.15 *Let $G$ be a $t$-And-Or formula $G_1 \wedge G_2 \wedge \ldots \wedge G_w$. For any $\beta$, $0 < \beta < t$, let $\rho$ be a random $p$-restriction, where $p = \frac{\beta}{t-\beta}$, and let $\alpha = \beta/ln\left[\frac{1+\sqrt{1+4e^\beta}}{2}\right]$. Then for all $\Delta \geq 0$, we have*

$$\Pr[DC(G|\rho) \geq \Delta] \leq \alpha^\Delta.$$

In the above lemma, $\alpha$ is minimized when $\beta = \beta_0 \approx 0.227537$. In which case, $\alpha = \alpha_0 \approx 0.4164447$. Let $\gamma_0 = \beta_0/2 \approx 0.1137685$.

Using Lemma 1, we will prove the Lemma 2, a stronger version of switching lemma. The key is the the following composite property of random restrictions. Observe that a $p_1$-restriction followed by a $p_2$-restriction has the same effect with a single $p_1p_2$-restriction. This property holds because the boolean variables are independently assigned at each step.

LEMMA 11.16 *Let $G$ be a $t$-And-Or formula $G_1 \wedge G_2 \wedge \ldots \wedge G_w$, and let $\rho$ be a random $\gamma_0/t$-restriction. Then for all $\Delta \geq 0$, we claim*

$$\Pr[DC(G|_\rho) \geq \Delta] \leq {\alpha_0}^{\Delta}$$

*Proof.* Let $q = \beta_0/t$ and $p = \frac{q}{2-q}$. Then $q = \frac{2p}{1+p} = \frac{p}{p+\frac{1-p}{2}}$ is the probability a variable is assigned a * in a random $p$-selection under the condition that it is assigned * or 0.

We have shown that

$$\Pr[DC(G|_{\rho\prime}) \geq \Delta] \leq {\alpha_0}^{\Delta},$$

where $\rho\prime$ is a random $p$-restriction.

Since $p = \frac{q}{2-q} > \frac{q}{2} = \frac{\gamma_0}{t}$, we know that $\gamma_0/(pt)$ is still a number less than 1. Because of the composite property of random restriction, a random $\gamma_0/t$-restriction $\rho$ can be realized by first applying a random $p$-restriction $\rho\prime$, followed by a $\frac{\gamma_0}{pt}$-restriction. Note that $DC(G|\rho\prime) < \Delta$ means there exists a decision tree with depth less than $\Delta$, which can compute $G$. Therefore we would also be able to compute $G$ within less than $\Delta$ depth with the more stringent restriction $\rho$. That is to say,

$$\Pr[DC(G|_\rho) \geq \Delta] \leq \Pr[DC(G|_{\rho\prime}) \geq \Delta] \leq {\alpha_0}^{\Delta}.$$

### 11.3.6 Circuit Lower Bounds

Consider general constant depth circuits. Denote by $C^d(s,t)$ the class of depth $d$ circuits with bfi (the abbreviation of bottom fanin) $\leq t$, and the number of gates above the first level $\leq s$. Denote by $C^d(s)$ the class of depth $d$ circuits without a bfi condition but with total size $\leq s$. It is clear that a circuit in $C^d(s)$ can be considered as a circuit in $C^{d+1}(s,1)$, by adding an extra layer of gates with fan-in 1.

The following lemma is proved using the switching lemma. In Lecture 16, we discussed on how to use the switching lemma to prove circuit lower bounds. We use the same ideas here and hence, we provide only a sketch of the proof.

LEMMA 11.17 *For all $C \in C^d(s, \gamma_0 n^{1/d})$, we have*

$$\Pr[DC(C|_\rho) \geq \gamma_0 n^{1/d}] \;\leq\; s \cdot \alpha_0^{\gamma_0 n^{1/d}} \;\approx\; s \cdot 2^{-0.143781 \cdot n^{1/d}},$$

*where $\rho$ is a random $1/n^{\frac{d-1}{d}}$-restriction.*

*Proof.* Let $t = \gamma_0 n^{1/d}$ and $p = 1/n^{\frac{1}{d}}$. Let $C \in C^d(s,t)$. Denote the number of gates on each level as $s_1, s_2, \ldots, s_d$ from the bottom to the top (output level). Clearly, $s_d = 1$ and $\sum_{i=1}^{d-1} s_i = s$. Apply Lemma 2 repeatedly $d-1$ times, each time with a random $p$-restriction.

After applying the first random restriction, suppose all the $s_1$ many $t$-AND-OR circuits at level 1 have DC $\leq t$. Then, we can switch these into $t$-OR-AND circuits and merge first and second levels eliminating a level. This process also eliminates $s_1$ gates from the circuit. Nonetheless, for each gate at the bottom level, it is possible that we cannot switch. By Lemma 2, for any one of the $s_1$ circuits, probability that we fail to to switch is $\leq \alpha_0^t$. So the probability of failure at this level is at most $s_1 \cdot \alpha_0^t$. Accumulating the probability of the failure on each level, and combining with the Lemma 2, we have

$$\Pr[DC(C|_\rho) \geq \gamma_0 t] \;\leq\; \sum_i s_i \cdot \alpha_0^t = s \cdot \alpha_0^t = s \cdot \alpha_0^{\gamma_0 n^{1/d}} \;\approx\; s \cdot 2^{-0.143781 \cdot n^{1/d}}$$

Finally, by the composite property of random restrictions, applying $d-1$ random $p$-restrictions has the same effect as one random $p^{d-1}$-restriction. The proof is complete.

We can apply Lemma 3 to $C \in C^d(s)$ by first transforming them into $C \in C^{d+1}(s,1)$. But we can actually do slightly better by a more delicate technique. Here we omit the proof of the following better bound.

Lemma 11.18 *For all $C \in C^d(s)$, we have*

$$\Pr[DC(C|_\rho) \geq \gamma_0 n^{1/d}] \;<\; s \cdot \alpha_0^{\gamma_0 n^{1/d}} \;\approx\; s \cdot 2^{-0.143781 \cdot n^{1/d}},$$

*where $\rho$ is a random $\alpha_0/(2n^{\frac{d-1}{d}})$-restriction.*

These results can be used to prove circuit lower bounds for the parity function. Consider any circuit $C$ in $C^d(s, \gamma_0 n^{1/(d-1)})$. Apply $d-2$ rounds of random $1/n^{1/(d-1)}$-restrictions. With probability $> 1 - s \cdot 2^{-0.143781 \cdot n^{1/(d-1)}}$, we get a circuit in $C^2(1, \gamma_0 n^{1/(d-1)})$ after switching and merging. The process is equivalent to applying a single random $n^{(d-2)/(d-1)}$-restriction. Let $N$ be the random variable for number of variables left (i.e., variables assigned *). Then, its expectation $E[N] = n^{1/(d-1)}$. By Chernoff bound we have,

$$\Pr[N \leq \gamma_0 n^{\frac{1}{d-1}}] < e^{-\frac{(1-\gamma_0)^2}{2} \cdot n^{\frac{1}{d-1}}} < e^{-0.3927 n^{\frac{1}{d-1}}}.$$

Hence, if $s < 2^{0.143781 \cdot n^{1/(d-1)}}$, the probability is approaching 1 that both $C$ is reduced to a circuit in $C^2(1, \gamma_0 n^{1/(d-1)})$ and $N > \gamma_0 n^{1/(d-1)}$. Suppose the circuit we started with computes parity on $n$ variables. Then, the circuit obtained after applying the random restriction computes parity on the remaining $N$ variables. Clearly, a $C^2(1,t)$ circuit cannot compute parity on $> t$ variables (see Lecture 16). We have proved the following lemma.

LEMMA 11.19 *For all* $C \in C^d(s, \gamma_0 n^{1/(d-1)})$*, if* $C$ *computes the parity function, then its size* $s$ *must satisfy*

$$s \;\geq\; 2^{0.143781 \cdot n^{1/(d-1)}}.$$

Lemma 11.19 can be used to obtain lower bounds for general circuits (without bfi). Again, we can simply transform a circuit in $C^d(s)$ into a circuit in $C^{d+1}(s,1)$, then apply Lemma 11.19. Using a more direct and finer analysis, one can prove the following lemma.

LEMMA 11.20 *For all* $C \in C^d(s)$*, if* $C$ *computes the parity function, then its size* $s$ *must satisfy*

$$s \geq 2^{0.143781 \cdot n^{\frac{1}{d-1}}}$$

### 11.3.7 Inapproximability Type Lower Bounds

Now we consider the inapproximability type lower bound. By *inapproximability*, we mean circuits with certain restrictions (like size, depth and bfi) cannot compute parity on significantly more than half of the possible cases. Specifically, for the parity function, we can simply guess 0 and 1 as the function value. So, it is easy to get 50% success. We want to show that, one cannot do significantly better. The decision tree depth lower bound is ideally suited for deriving the inapproximability type lower bound, and the decision tree perspective was introduced precisely for this reason. Our goal is to show that, when the sizes of the circuits are below some lower bound, the circuits will make asymptotically 50% error on all possible inputs.

Let $C$ be a depth $d$ circuit. Note that after some restriction $\rho$, if $C$ is reduced to a decision tree of depth smaller than the number of variables left, then for exactly half of the 0-1 extensions of $\rho$, $C$ agrees on the parity. This is because at every leaf of the decision tree, the circuit $C$ is completely determined.

Consider $\Pr[\; C(x_1, \ldots, x_n) = \oplus(x_1, \ldots, x_n) \;]$, where $\oplus(x_1, \ldots, x_n)$ denotes the parity function, and the probability is over all $2^n$ assignments. This random restriction technique can be realized by first assigning any random restriction, followed by an unbiased 0-1 assignments for all the remaining variables. Let $E_1$ denote the event that after the random restriction, we end up with a decision tree of depth not more than $t$, and let $E_2$ denote the event that the number of variables $N$ assigned to * is more than $t$. Then let $E = E_1 \wedge E_2$, and let $[\, C = \oplus \,]$ denote $[\, C(x_1, \ldots, x_n) = \oplus(x_1, \ldots, x_n) \,]$ for convenience. As we already pointed out, $\Pr[\, C = \oplus \,|\, E \,] = 1/2$ due to a property of the parity function.

Expending in terms of conditional probabilities, we have

$$\begin{aligned}
\Pr[C = \oplus] &= \Pr[E] \cdot \Pr[C = \oplus | E] + \Pr[\neg E] \cdot \Pr[C = \oplus | \neg E] \\
&= (1 - \Pr[\neg E]) \cdot \Pr[C = \oplus | E] + \Pr[\neg E] \cdot \Pr[C = \oplus | \neg E] \\
&= \Pr[C = \oplus | E] + \Pr[\neg E](Pr[C = \oplus | \neg E] - \Pr[C = \oplus | E]).
\end{aligned}$$

As we noted, $\Pr[\, C = \oplus \,|\, E \,] = 1/2$, and $\Pr[C = \oplus | \neg E] \leq 1$. Then substitute these 2 observations into the above equation,

$$\left| \Pr[C = \oplus] - \frac{1}{2} \right| \;\leq\; \frac{1}{2} \Pr[\neg E].$$

Since $\Pr[C = \oplus] + \Pr[C \neq \oplus] = 1$, we have

$$\Pr[\, C = \oplus \,] - \Pr[\, C \neq \oplus \,] = 2 \left( \Pr[\, C = \oplus \,] - \frac{1}{2} \right),$$

and hence

$$|\Pr[\, C = \oplus \,] - \Pr[\, C \neq \oplus \,]| \;\leq\; \Pr[\neg E].$$

Now we specify the parameters of the random restrictions. Let $m = \gamma_0 n^{1/d}$. First consider any $C \in C^d(s, \gamma_0 m)$. Let $t = \gamma_0 m$ and apply Lemma 11.17. With a random $1/n^{(d-1)/d}$-restriction, we have

$$\Pr[\neg E_1] \leq s\alpha_0^t \approx s\dot{2}^{0.143781 \cdot m}.$$

Again by using the Chernoff bound, we estimate $\Pr[\neg E_2] = \Pr[N \leq \gamma_0 m]$ as follows.

$$\Pr[\neg E_2] \leq e^{-\frac{(1-\gamma_0)^2}{2} m} < e^{0.3927m}.$$

Thus $\Pr[\neg E_2]$ is dominated by $\Pr[\neg E_1]$. This analysis gives the following bound.

Lemma 11.21 *For all $C \in C^d(2^{0.07189 n^{1/d}}, \gamma_0 n^{1/d})$, we have*

$$|\Pr[\, C = \oplus \,] - \Pr[\, C \neq \oplus \,]| \;\leq\; 2^{-0.07189 n^{1/d}}.$$

Again straightforward application of the above lemma gives inapproximability results for general circuits (without bfi). A more careful analysis leads to the following lemma.

Lemma 11.22 *For all circuits $C \in C^d(2^{0.07189 n^{1/d}})$, we have*

$$|\Pr[C = \oplus] - \Pr[C \neq \oplus]| \leq 2^{0.07189 n^{1/d}}.$$

# Chapter 12

# Miscellaneous Results

## 12.1 Relativized Separation of NP and P

We construct an oracle relative to which NP $\neq$ P. And also another oracle relative to which NP $=$ P.

Theorem 12.1 (Baker, Gill, Solovay) *There are computable languages $A$ and $B$ such that*

1. $\mathrm{NP}^A = \mathrm{P}^A$.
2. $\mathrm{NP}^B \neq \mathrm{P}^B$.

**Proof (Part I):** This is easy to show. Choose $A$ to be any PSPACE-Complete language. Then,

$$\mathrm{NP}^A \subseteq \mathrm{PSPACE}^A \subseteq \mathrm{PSPACE} \subseteq \mathrm{P}^A$$

The first inclusion is because, we can simulate an NP machine in PSPACE by running through all the possible guesses of the NP machine. The second inclusion is because $A$ is in PSPACE, and hence we can resolve queries to $A$ in PSPACE without asking the oracle. As $A$ is PSPACE-Hard, any PSPACE language can be reduced to $A$. So the last inclusion holds.

**Proof (Part II):** For any language $B$, let $T_B$ be the language

$$T_B = \{1^n | \exists x, |x| = n, x \in L\}$$

We first show that for any $B$, $T_B \in \mathrm{NP}^B$. Given $1^n$ as input, simply guess a string $x$ of length $n$, then, use the oracle to verify whether $x \in B$. If so accept, else reject.

We construct a language $B$ such that $T_B \notin P^B$. The basic idea is simple. Intuitively, any machine that decides $T_B$, given $1^n$ as input, has to find out whether there is some string of length $n$ in $B$. But there are $2^n$ such strings. No polynomial machine, can ask that many queries to oracle $B$.

Let $N_1, N_2, \ldots$ be an enumeration of all deterministic polynomial time Turing machines. Let the running time of $N_i$ be bounded by $n^i$. Such an enumeration can be achieved by, first enumerating all deterministic TMs and then adding a "clock" of size $n^i$ to the $i^{th}$ TM. We shall diagonalize over all these $N_i$. We construct $B$ in stages. We can think of $B$ being empty initially, and we add some strings to it in each stage. In the $k^{th}$ stage, we make sure that, $N_k$ does not decide $T_B$.

We describe the $k^{th}$ stage. We heave already simulated $N_1, N_2, \ldots N_{k-1}$ in previous $(k-1)$ stages and would have answered many oracle queries. Let $n_0$ be the maximum length of any query asked by any of these $k-1$ machines in our simulations. By the end of stage $k-1$, we would have constructed $B$ up to some length $n_0$. At the $k^{th}$ stage, we make sure that $N_k$ does not decide $T_B$. Now, choose $n$, such that $n > n_0$ and $2^n > n^k$ and simulate $N_k(1^n)$. The catch is in the second condition. $N_k$ is a machine with time bound $n^k$. It can ask at most $n^k$ queries. If it asks a query of length smaller than $n_0$, we already know the answer-(we have already constructed $B$ up to length $n_0$). If it asks any query of length more than $n_0$, we simply answer "no". (That is, keep those strings out of $B$). Now it's time to diagnolize. At the end of simulation, suppose $N_k$ accepts $1^n$. We keep all strings of length $n$ out of $B$. Suppose $N_k$ rejects $1^n$. It could have asked at most $n^k$ queries of length $n$ (all of which we answered "no"). As $2^n > n^k$, there exists at least one string $y$ of length $n$ that was not a query of $N_k$. We add this string to $B$. In either case, $N_k$ has failed to decide $1^n$ correctly.

A formal algorithm to enumerate $B$ is as follows. Think of $B$ being empty initially and we add strings to it in stages. At any stage $k$, $Q_k$ represents the set of queries asked by $N_k$ in that stage. And $n_0$ is set to 0 initially. Run the following procedure for $k = 1, 2 \ldots$.

1. Choose an $n$, with $n > n_0$ and $2^n > n^k$. Set $Q_k$ to be empty set.
2. Simulate $N_k(1^n)$ for $n^k$ steps. If $q$ is a query by $N_k$,
   - Add $q$ to $Q_k$.
   - If $|q| \leq n_0$ then answer $B(x)$ else answer "no".
3. At the end of simulation:-
   - If $n_k$ rejects $1^n$:– choose a string $y$ of length $n$ not in $Q_k$ and add it to $B$.
   - If $n_k$ accepts $1^n$:– do nothing.
4. Let $q$ be the longest string in $Q_k$. Set $n_0 \longleftarrow \max\{n_0, |x|\}$.

# Bibliography

[1] D. Du and K. Ko. *Theory of Computational Complexity.* John Wiley and Sons, 2000.

[2] M. Garey and D. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness.* W.H. Freeman, 1979.

[3] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[4] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.