Chapter 5

Polynomial Hierarchy

The class NP was defined to capture some natural computational problems, for instance

$$\text{CLIQUE} = \{< G, k >: \text{ graph } G \text{ has a clique of size} \geq k\}. \tag{5.1}$$

However, it seems to fail to capture many other natural problems. Consider the problem obtained by just modifying the above problem a little, namely, determinining the largest clique in a graph (phrased as a decision problem):

$$\text{EXACT-CLIQUE} = \{< G, k >: \text{ the largest clique in graph } G \text{ has size exactly } k\}. \tag{5.2}$$

There seems to be no short certificate for membership: $< G, k > \in$ EXACT-CLIQUE iff *there exists* a clique of size k in G and *every other* clique has size at most k.

Similary, consider the language MIN-CNF; it is the decision version of circuit minimization, a topic of interest in electrical engineering. We say that two boolean formulae are *equivalent* if they have the same set of satisfying assignments.

$$\text{MIN-CNF} = \{< \varphi >: \varphi \text{ is not equivalent to any smaller formula}\}. \tag{5.3}$$

$$= \{< \varphi >: \forall \psi, |\psi| < |\varphi|, \exists \text{ assignment } s \text{ such that } \varphi(s) \neq \psi(s)\}. \tag{5.4}$$

Again, there is no obvious notion of a certificate of membership. Note that the problem is in PSPACE, but does not appear to be PSPACE-complete either.

Clearly, one is faced with a new kind of complexity phenomenon and it is instructive to see how we define the right complexity class, namely, the polynomial hierarchy, to "capture" it. This class is a generalization of P, NP, and coNP. Recall the definition of NP, coNP.

DEFINITION 12 *Language L is in NP if there is a language $L_0 \in$ P and constants $c, d > 0$ such that*

$$\forall x \in \{0,1\}^* \quad x \in L \Leftrightarrow \quad \exists y \in \{0,1\}^*, |y| \leq d\,|x|^c \quad \textit{and} \quad (x, y) \in L_0.$$

Language L in in coNP iff $\overline{L} \in$ NP. In other words, there is a language $L_0 \in$ P and constants $c, d > 0$ such that

$$\forall x \in \{0,1\}^* \quad x \in L \Leftrightarrow \quad \forall y \in \{0,1\}^*, |y| \leq d\,|x|^c \quad \textit{and} \quad (x, y) \in L_0.$$

DEFINITION 13 (POLYNOMIAL HIERARCHY) *The polynomial hierarchy is defined as* $\cup_{i\geq 0}\mathbf{\Sigma}_i^p$ *or equivalently* $\cup_{i\geq 0}\mathbf{\Pi}_i^p$*, where*

1. $\mathbf{\Sigma}_0^p = \mathbf{\Pi}_0^p =$ *P.*
2. $\mathbf{\Sigma}_1^p =$ *NP,* $\mathbf{\Pi}_1^p =$ *coNP.*
3. $\mathbf{\Sigma}_i^p$ *consists of any language for which there is a language* $L_0 \in \mathbf{\Pi}_{i-1}^p$ *and* $c, d > 0$ *such that*

$$\forall x \in \{0,1\}^* \quad x \in L \Leftrightarrow \quad \exists y \in \{0,1\}^*, |y| \leq d\,|x|^c \quad \textit{and} \quad (x,y) \in L_0.$$

4. $\mathbf{\Pi}_i^p$ *consists of any language for which there is a language* $L_0 \in \mathbf{\Sigma}_{i-1}^p$ *such that*

$$\forall x \in \{0,1\}^* \quad x \in L \Leftrightarrow \quad \forall y \in \{0,1\}^*, |y| \leq d\,|x|^c \quad \textit{and} \quad (x,y) \in L_0.$$

EXAMPLE 7 To understand this definition, let us unwrap it for $i = 2$. Language L is in $\mathbf{\Sigma}_2^p$ if there is a language $L_0 \in$ P and constants $c, d > 0$ such that a string x is in L iff

$$\exists y_1 \ \leq d\,|x|^c \ \forall y_2 \leq d\,|x|^c \quad (x, y_1, y_2) \in L_0.$$

Similarly Language L is in $\mathbf{\Pi}_2^p$ if there is a language $L_0 \in$ P and constants $c, d > 0$ such that a string x is in L iff

$$\forall y_1 \ \leq d\,|x|^c \ \exists y_2 \leq d\,|x|^c \quad (x, y_1, y_2) \in L_0.$$

Clearly, $L \in \mathbf{\Pi}_2^p$ iff $\overline{L} \in \mathbf{\Sigma}_i^p$.

Similarly we can unwrap the definition for general i and directly define $\mathbf{\Sigma}_i^p$ using i quantifiers, the first being $\exists$ and the rest alternating between $\exists$ and $\forall$. The class $\mathbf{\Pi}_i^p$ involves involves i quantifiers, alternating between $\exists$ and $\forall$ and beginning with $\forall$.

What are some natural problems in these classes?

EXAMPLE 8 We show that EXACT-CLIQUE $\in \mathbf{\Sigma}_2^p \cap \mathbf{\Pi}_2^p$. (A finer placement of EXACT-CLIQUE is done in the exercises.)

To see that EXACT-CLIQUE $\in \mathbf{\Sigma}_2^p$, note that $< G, k >$ $\in$ EXACT-CLIQUE iff

$\exists$ S that is a clique of size k in G and $\forall S'$ set S' is not a clique of size $\geq k+1$.

Furthermore, EXACT-CLIQUE $\in \mathbf{\Pi}_2^p$ since one could also say

$\forall S'$ set S' is not a clique of size $\geq k+1$ and $\exists$ S that is a clique of size k in G.

The reader can similarly check that MIN-CNF in $\mathbf{\Pi}_2^p$.

The class Σ_i^p has a complete problem involving quantified boolean with limited number of alternations. Specifically, it is

$$\Sigma_i\text{-SAT} = \exists \vec{x}_1 \forall \vec{x}_2 \exists \cdots Q\vec{x}_i \ \varphi(\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_i) = 1, \tag{5.5}$$

where φ is a boolean formula, each $\vec{x}_i$ is a vector of boolean variables, and Q is $\exists$ or $\forall$ depending on whether i is odd or even.

The next simple theorem is the only theorem in this chapter.

THEOREM 13

1. *If* $P = NP$ *then* $PH = P$.

2. *If* $\mathbf{\Sigma}_i^p = \mathbf{\Pi}_i^p$ *then* $PH = \mathbf{\Sigma}_i^p$.

PROOF: We do the first part; the second part is similar and also easy.

Assuming $P = NP$ we prove by induction on i that $\mathbf{\Sigma}_i^p, \mathbf{\Pi}_i^p \subseteq P$. Clearly this is true for $i = 0$. Assume it is true for $i \leq m$ and $L \in \mathbf{\Sigma}_{m+1}^p$. Then by Definition 13 there is a language $L_0 \in \mathbf{\Pi}_m^p$ and $c, d > 0$ such that

$$\forall x \in \{0,1\}^* \quad x \in L \Leftrightarrow \quad \exists y \in \{0,1\}^*, |y| \leq d\,|x|^c \quad \text{and} \quad (x,y) \in L_0.$$

By the inductive hypothesis, $L_0 \in P$. By examining how L is defined in terms of L_0, we conclude that then $L \in NP$. Hence L must be in P, and the induction is completed. □

We strongly believe that not only is $P \neq NP$ but also that all levels of PH are distinct. This latter conjecture will be useful in the rest of the course; we will reduce other conjectures to it (in other words, show that the other conjectures follow from this conjecture).

5.1 Alternating Turing machines

There is an alternating characterization of the polynomial hierarchy using *alternating Turing machines*. (Aside: There is a third characterization of PH that uses oracle turing machines, and is explored in the exercises.)

Alternating TMs are generalizations of nondeterministic TMs; we think of the states as labelled with either $\exists$ or $\forall$. A nondeterministic TM is a special case in which states are labelled with only $\exists$. The acceptance criterion for such a machine is defined in the obvious way. Whenever a machine is an $\exists$ state, we say that it accepts the input if one of the branches leading out from it leads to accept. When it is in a $\forall$ state we say it accepts if every branche leading out from it accepts. Thus whether or not the computation tree accepts is determined by looking at the tree of all possible computation branches, and propagating the YES/NO decisions at the leaves to the root using the obvious semantics for $\exists$ and $\forall$. Then $\mathbf{\Sigma}_i^p$ is the class of languages accepted by polynomial time alternating TMs in which each computation branches features at most $(i-1)$ alternations between $\exists$ and $\forall$, and the machine starts in an $\exists$ state. The class $\mathbf{\Pi}_i^p$ is similarly defined except the machine starts in a $\forall$ state.

It is easily checked that this definition is equivalent to our earlier definitions.

Unlimited number of alternations? What if we consider polynomial-time alternating Turing machines with no *a priori* bound on the number of quantifiers? The class of languages accepted by them is called AP, and the exercises ask you to prove that $AP = PSPACE$.

Similarly, one can consider alternating Turing machines that run in polynomial space. The class of languages accepted by such machines is called APSPACE, and the exercises ask you to prove that $APSPACE = EXPTIME$. One can similarly consider alternating logspace machines; the set of languages accepted by them is exactly P.

Problems

§1 Show that the language in (5.5) is complete for $\mathbf{\Sigma}_i^p$ under polynomial time reductions. (Hint use the NP-completeness of SAT.)

§2 Show that PH has a complete language iff it collapses to some finite level $\mathbf{\Sigma}_i^p$.

§3 Prove that AP = PSPACE.

§4 Show that APSPACE = EXPTIME. (Hint: The nontrivial direction EXPTIME $\subseteq$ APSPACE uses ideas similar to those in the Cook-Levin theorem, namely, that computation is very local.)

§5 Show that $\mathbf{\Sigma}_2^p = \mathrm{NP}^{\mathrm{SAT}}$. Generalize your proof to give a characterization of PH in terms of oracle Turing machines.

§6 The class DP is defined as languages L for which there are two languages $L_1 \in \mathrm{NP}, L_2 \in$ coNP such that $L = L_1 \cap L_2$. (Remark: Do not confuse DP with NP $\cap$ coNP, which may seem superficially similar.) Show that

(a) EXACT-CLIQUE $\in$ DP.

(b) Every language in DP is polynomial-time reducible to EXACT-CLIQUE.

§7 Suppose A is some language such that $\mathrm{P}^A = \mathrm{NP}^A$. Then show that $\mathrm{PH}^A \subseteq \mathrm{P}^A$ (in other words, the proof of Theorem 13 *relativizes*).

Chapter 6

Circuits

The boolean circuit model mimics a silicon chip. A circuit with n inputs is a directed acyclic graph with n leaves (called the *input wires*) and each internal node is labelled with $\vee, \wedge, \neg$. The $\vee$ and $\wedge$ nodes have fanin (i.e., number of incoming edges) of 2 and the $\neg$ nodes have fanin 1. There is a designated *output node*. We think of n bits being fed into the input wires. Each internal node/gate applies the boolean operation it is labeled with on its incoming wires to produce a single bit. Thus bits filter through the circuit —remember that it is acylic, so feedback effects are absent— until a single output bit emerges at the output node. Thus the circuit implements a function from $\{0,1\}^n \to \{0,1\}$. The *size* of the circuit is the number of nodes in it.

A *circuit family* $(W_n)_{n\geq 1}$ is a sequence of circuits where W_n is a circuit with n inputs. We say that the family *decides* a language L if for every input x,

$$x \in L \Leftrightarrow W_{|x|}(x) = 1. \tag{6.1}$$

Note that every language is decidable by a circuit family of size $O(n2^n)$, since the circuit for input length n could contain 2^n "hardwired" bits indicating which inputs are in the language. Given an input, the circuit looks up the answer from this table. (The reader may wish to work out an implementation of this circuit.)

Definition 14 *The class P/poly consists of every language that is decidable by a circuit family of size $O(n^c)$ for some $c > 0$.*

The proof of the following theorem follows from an even stronger Theorem 15.

Theorem 14
$P \subseteq$ P/poly.

Can we give a good upperbound on the computational power represented by P/poly? The next example shows this is difficult since P/poly contains an undecidable language.

Example 9 Every unary language has linear size circuits since the circuit for an input size n only needs to have a single "hardwired" bit indicating whether or not 1^n is in the language. Hence the following unary language has linear size circuits, even though it is undecidable:

$$\left\{1^{<M,w>} : \text{ TM } M \text{ accepts } w\right\}, \tag{6.2}$$

where $< M, w >$ is an encoding that encodes strings with integers.

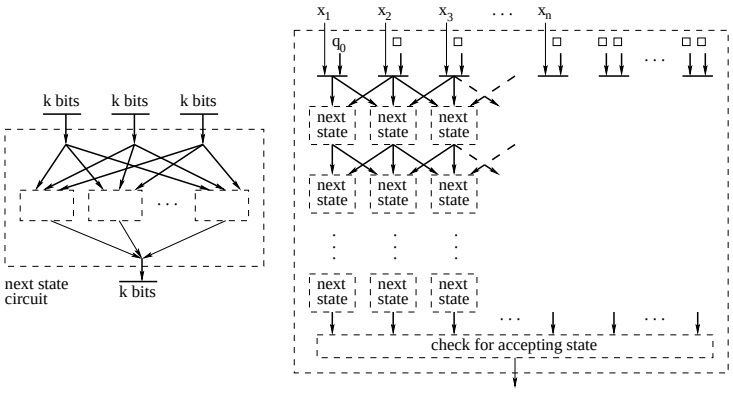

Figure 6.1: Circuit to simulate a Turing machine computation by constructing the tableau.

This example suggests that it may be fruitful to only consider circuits that can actually be built, say using a Turing machine. This motivates the following definition.

DEFINITION 15 (LOGSPACE-UNIFORM) *A circuit family* $\{C_n\}$ *is* logspace uniform *if there is a logspace transducer which, given the input* 1^n*, outputs a description of* C_n*.*

However, polynomial circuits that are logspace-uniform correspond to a familiar complexity class.

THEOREM 15
A language has logspace-uniform circuits of polynomial size iff it is in P.

PROOF: The only if part is trivial. We prove the if part.

We show that every language that is decidable in $t(n)$ time has logspace-uniform circuits of size $O(t(n)^2)$. (A more careful argument can actually yield circuits of size $O(t(n) \log t(n))$. The main idea is similar to that in the Cook-Levin theorem, where we noticed that computation is very local, since the machine affects its tape only in the cell currently occupied by its head.

On any input of size n, the tableau of the machine's computation is a matrix of size $t(n) \times t(n)$. Assume wlog that the machine enters the leftmost cell before accepting. Construct a circuit in which use a wire for each bit of the tableau. The bits of the machine's input enter at the input wires of the circuit. Construct a "next step" module, which is a circuit that, given the contents of three adjacent cells in a row of the tableau, computes the contents of the cell below the middle cell. Since the machine has a fixed transition diagram and alphabet, each tableau cell is represented by $k = O(1)$ bits, so the the next-step circuit has size $O(1)$ (could be exponential in k). Replicate this next-step module everywhere as needed so that all the bits in the tableau are computed correctly. (See Figure 6.1.) The output of the overall circuit is computed by examining the cell in the lower left corner of tableau; it is 1 iff the machine is in an accept state.

Clearly, the description of this circuit is very local, and a logspace machine could output the description in a gate-by-gate fashion. □

6.0.1 Turing machines that take advice

There is a way to define P/poly using TMs. We say that a Turing machine has *access to an advice family* $\{a_n\}_{n\geq 0}$ (where each a_n is a string) if while computing on an input of size n, the machine is allowed to examine a_n. The advice family is said to have *polynomial size* if there is a $c \geq 0$ such that $|a_n| \leq n^c$.

EXAMPLE 10 We showed that the language in (6.2) has linear size circuits. Now we show that can be decided by a polynomial time Turing machine with 1 bit of advice. The advice string for inputs of length n is the single bit indicating whether or not 1^n is in the language.

The previous example is an example of a more general phenomenon described in the next theorem.

THEOREM 16
$L \in$ P/poly iff L is decidable by a polynomial-time Turing machine with access to an advice family of polynomial size.

PROOF: If $L \in$ P/poly, we can provide the description of its circuit family as advice to a Turing machine. When faced with an input of size n, the machine just simulates the circuit for this circuit provided to it.

Conversely, if L is decidable by a polynomial-time Turing machine with access to an advice family of polynomial size, then we can use the construction of Theorem 14 to construct an equivalent circuit for each input size with the corresponding advice string hardwired into it. □

6.1 Karp-Lipton Theorem

Is SAT in P/poly? If so, one could imagine a government-financed project to construct a small circuit that solves SAT on, say, $10,000$ variables. Upon discovering such a circuit we could incorporate it in a small chip, to be used in all kinds of commercial products. In particular, such a chip would jeopardise all current cryptographic schemes.

The next theorem seems to dash such hopes, at least if you believe that the polynomial hierarchy does not collapse.

THEOREM 17 (KARP-LIPTON, WITH IMPROVEMENTS BY SIPSER)
If NP $\subseteq$ P/poly then PH $= \mathbf{\Sigma}_2^p$.

Note that if NP $\subseteq$ P/poly then SAT has a polynomial-size circuit family. Namely, for some $k > 0$ there is a sequence of circuits $W_1, W_2, W_3, \ldots$, where W_n is a circuit of size n^k that decides satisfiability of all boolean formulae whose encoding size is $\leq n$. Of course, NP $\subseteq$ P/poly merely implies the *existence* of such a family; there may be no easy way to construct the circuits.

LEMMA 18 (SELF-REDUCIBILITY OF SAT)
There is a polynomial-time computable function h such that if $\{W_n\}_{n\geq 1}$ is a circuit family that solves SAT, then for all boolean formulae φ:

$$\varphi \in SAT \quad \text{iff} \quad h(\varphi, W_{|\varphi|}) \text{ is a satisfying assignment for } \varphi. \tag{6.3}$$

PROOF: The main idea in the computation of h is to use the provided circuit to generate a satisfying assignment. Ask the circuit if φ is satisfiable. If so, ask it if $\varphi(x_1 = T)$ (i.e., φ with the first variable assigned True) is satisfiable. The circuit's answer allows us to reduce the size of the formula. If the circuit says no, we can conclude that $\varphi(x_1 = F)$ is true, and have thus reduced the number of variables by 1. If the circuit says yes, we can substitute $x_1 = T$ and again reduced the number of variables by 1. Continuing this way, we can generate a satisfying assignment. This proves the Lemma. (Aside: The formal name for the above property of SAT is *downward self-reducibility.* All NP-complete languages have this property.) □

Now we prove the Theorem.

PROOF:(Theorem 17) We show NP $\subseteq$ P/poly implies $\mathbf{\Pi}_2^p \subseteq \mathbf{\Sigma}_2^p$.

Let $L \in \mathbf{\Pi}_2^p$. Then there is a language $L_1 \in$ NP and $c > 0$ such that

$$L = \{x \in \{0,1\}^* : \forall\ y, |y| \leq |x|^c, (x,y) \in L_1\}. \tag{6.4}$$

Since $L_1 \in$ NP, there is a polynomial-time reduction, say g, from L_1 to SAT. Thus

$$\forall\ z \in \{0,1\}^* : z \in L_1 \qquad \text{iff} \qquad g(z) \in \text{SAT}.$$

Suppose further that $d > 0$ is such that $|g(z)| \leq |z|^d$.

Now we can rewrite (6.4) as

$$L = \{x \in \{0,1\}^* : \forall\ y, |y| \leq |x|_1^c, g(x,y) \in \text{SAT}\}.$$

Note that $|g(x,y)| \leq (|x|+|y|)^d$. Let us simplify this as $|x|^{cd}$. Thus if $W_{n\geq 1}$ is a n^k-sized circuit family for SAT, then by (6.3) we have

$$L = \left\{x : \forall\ y, |y| \leq |x|_1^c, h(g(x,y), W_{|x|^{cd}}) \text{ is a satisfying assignment for } g(x,y)\right\}.$$

Now we are almost done. Even though there may be no way for us to construct the circuit for SAT, we can just try to "guess" it. Namely, an input x is in L iff

$$\begin{gathered}\exists W, \text{ a circuit with } |x|^{cd} \text{ inputs and size } |x|^{cdk} \text{ such that} \\ \forall\ y, |y| \leq |x|_1^c, h(g(x,y), W) \text{ is a satisfying assignment for } g(x,y).\end{gathered}$$

Since h is computable in polynomial time , we have thus shown $L \in \mathbf{\Sigma}_2^p$. □

6.2 Circuit lowerbounds

Theorem 14 implies that if NP $\not\subseteq$ P/poly, then P $\neq$ NP. The Karp-Lipton theorem gives hope that NP $\not\subseteq$ P/poly. Can we resolve P versus NP by proving NP $\not\subseteq$ P/poly? There is reason to invest hope in this approach as opposed to proving direct lowerbounds on Turing machines. By representing computation using circuits (Theorem 14) we seem to actually peer into the guts of it rather than treating it as a blackbox. Thus we may be able to get around the relativization results of Lecture 3.

Sadly, such hopes have not come to pass. After two decades, the best circuit size lowerbound for an NP language is only $4n$. (However, see the Problems for a better lowerbound for a language in PH.) On the positive side, we have had notable success in proving lowerbounds for more restricted circuit models, as we will see later in the course.

By the way, it is easy to show that for large enough n, almost every boolean function on n variables requires circuits of size at least $2^n/10n$. The proof uses a simple counting argument. There are at most s^{3s} circuits of size s (just count the number of labelled directed graphs, where each node has indegree at most 2). Hence this is an upperbound on the number of functions on n variables with circuits of size s. For $s = 2^n/10n$, this number is at most $2^{2^n/10}$, which is miniscule compared 2^{2^n}, the number of boolean functions on n variables. Hence most boolean functions do not have such small circuits.

Of course, such simple counting arguments do not give an explicit boolean function that requires large circuits.

6.3 Finer gradations among circuit classes

There are two reasons why subclasses of P/poly are interesting. First, proving lowerbounds for these subclasses may give insight into how to separate NP from P/poly. Second, these subclasses correspond to interesting computational models.

The most interesting such connection is to the *massively parallel computers*. In such a computer —a popular commercial product in in the 1980s and 1990s—one uses simple off-the-shelf microprocessors and links them using an *interconnection network* that allows them to send messages to each other. Usual interconnection networks such as the *hypercube* allows linking n processors such that interprocessor communication is possible —with some upperbounds on the total load on the network—in $O(\log n)$ steps. The processors compute in lock-step (for instance, to the ticks of a global clock) are assumed to do a small amount of computation in each step, say an operation on $O(\log n)$ bits. Thus each processor computers has enough memory to remember its own address in the interconnection network and to write down the address of any other processor, and thus send messages to it. We are purposely omitting many details of the model since the validity of Theorem 19 below does not depend upon them.

DEFINITION 16 *A computational task is said to have* efficient parallel algorithms *if inputs of size n can be solved using a parallel computer with $n^{O(1)}$ processors and in time $\log^{O(1)} n$.*

EXAMPLE 11 Given two n bit numbers x, y we wish to compute $x + y$ fast in parallel. The gradeschool algorithm proceeds from the least significant bit and maintains a *carry bit*. The most significant bit is computed only after n steps. This algorithm does not take advantage of parallelism. A better algorithm called *carry lookahead* (see ??) assigns each bit position to a separate processor and then uses interprocessor communication to propagate carry bits. It takes $O(n)$ processors and $O(\log n)$ time.

There are also efficient parallel algorithms for integer multiplication and division (the latter is quite nonintuitive and quite unlike the gradeschool algorithm!).

Now we related parallel computation to circuits. The *depth* of a circuit is the length of the longest directed path from an input node to the output node.

DEFINITION 17 (NICK'S CLASS OR NC) *A language is in NC^i if there are constants $c, d > 0$ such that it can be decided by a logspace-uniform family of circuits $\{C_n\}$ where C_n has size $O(n^c)$ and depth $O(\log^d n)$. The class NC is $\cup_{i \geq 1} NC^i$.*

A related class is the following.

DEFINITION 18 (AC) *The class AC^i is defined similarly to NC^i except gates are allowed to have unbounded fanin. The class AC is $\cup_{i \geq 0} AC^i$.*

Since unbounded (but poly(n)) fanin can be simulated using a tree of ORs/ANDs of depth $O(\log n)$, we have $\mathrm{NC}^i \subseteq \mathrm{AC}^i \subseteq \mathrm{NC}^{i+1}$, and the inclusion is known to be strict for $i = 0$ as we will see in later Chapter. (Notice, NC^0 is a trivial class since the circuit's output depends upon a constant number of input bits, but AC^0 does not suffer from this limitation.) The classes AC, NC are important because of the following.

THEOREM 19
A language has efficient parallel algorithms iff it is in NC.

PROOF: Suppose a language $L \in$ NC and is decidable by a circuit family $\{C_n\}$ where C_n has size $N = O(n^c)$ and depth $D = O(\log^d n)$. Take a general purpose parallel computer with N nodes and configure it to decide L as follows. Compute a description of C_n and allocate the role of each circuit node to a distinct processor. (This is done once, and then the computer is ready to compute on any input of length n.) Each processor, after computing the output at its assigned node, sends the resulting bit to every other circuit node that needs it. Assuming the interconnection network delivers all messages in $O(\log N)$ time, the total running time is $O(\log^{d+1} N)$.

The reverse direction is similar, with the circuit having $N \cdot D$ nodes arranged in D layers, and the ith node in the tth layer performs the computation of processor i at time t. The role of the interconnection network is played by the circuit wires. □

A major open question in this area is whether P = NC. We believe that the answer is No (though we are currently even unable to separate PH from NC^1). This motivates the theory of *P-completeness*, a study of which problems are likely to be in NC and which are not.

6.3.1 P-completeness

DEFINITION 19 *A language* P-complete *if it is in P and every language in P is logspace-reducible to it.*

The following easy theorem is left for the reader.

THEOREM 20
If language L is P-complete then $L \in$ NC (resp., L) iff P = NC (resp., P = L).

Consider the following language.

$$\text{CIRCUIT-VALUE} = \{< C, x >: \ C \text{ is a circuit and } x \text{ is an input such that } C(x) = 1\}. \tag{6.5}$$

THEOREM 21
CIRCUIT-VALUE is P-complete.

PROOF: The language is clearly in P. A logspace-reduction from any other language in P to this language is implicit in the proof of Theorem 15. □

6.4 Circuits of exponential size

As noted, every language has circuits of size $O(n2^n)$. However, computing these circuits may difficult—and impossible if the language is undecidable.

However, if we require circuits to be efficiently computable in the sense of the next Definition, then it could be that languages require circuits of size more than $n2^n$. In fact this notion allows us to give new definitions of some familiar complexity classes analogous to the definition of P in Theorem 15.

DEFINITION 20 *Let $\{\mathcal{C}_n\}_{n\geq 1}$ be a circuit family of size $S(n) \geq n$ for input of length n. We say that this is a Direct Connect uniform (DC uniform) family if all the following functions can be computed in deterministic $O(poly(n)log(S(n)))$ time :*

- *$TYPE(n, i)$ = the type (AND, OR, NOT, INPUT, OUTPUT) of gate i in circuit $\mathcal{C}_n$.*
- *$IN(n, i, j) = k$ where k is j^{th} in the ordered list of indices to the gates that feed into gate i, $k = NONE$ if there is no j^{th} gate.*
- *$OUT(n, i, j) = k$ where k is j^{th} in the ordered list of indices to the gates that gate i feeds into, $k = NONE$ if there is no j^{th} gate.*

Note that the circuits may have exponential size, but they have a succinct representation in terms of a TM which can systematically generate any required node of the circuit in polynomial time.

Now we give a (yet another) characterization of the class PH, this time as languages computable by uniform circuit families of bounded depth. We leave it as an exercise.

THEOREM 22
$L \in PH$ iff L can be computed by a DC uniform circuit family $\{\mathcal{C}_n\}$ that

- *uses AND, OR, NOT gates.*
- *has size $2^{n^{O(1)}}$ and constant depth (i.e., depth $O(1)$).*
- *gates can have unbounded (exponential) fanin.*
- *the NOT gates appear only at the input level.*

If we drop the restriction that the circuits have constant depth, then we obtain exactly EXPTIME (see Exercises).

Exercises

§1 Show for every $k > 0$ that PH contains languages whose circuit complexity is $\Omega(n^k)$. (Hint: Keep in mind the proof of the *existence* of functions with high circuit complexity.)

§2 Solve the previous question with PH replaced by $\mathbf{\Sigma}_2^p$.

§3 Show that if some unary language is NP-complete, then P = NP.

§4 (*Open*) Suppose we make a stronger assumption than NP $\subseteq$ P/poly: every language in NP has linear size circuits. Can we show something stronger than PH $= \mathbf{\Sigma}_2^p$?

§5 (a) Describe an NC circuit for the problem of computing the product of two given $n \times n$ matrices A, B.

(b) Describe an NC circuit for computing, given an $n \times n$ matrix, the matrix A^n. (Hint: Use repeated squaring: $A^{2^k} = (A^{2^{k-1}})^2$.)

(c) Conclude that the PATH problem (and hence every NL language) is in NC. (Hint: What is the meaning of the (i, j)th entry of A^n?)

§6 A *formula* is a circuit in which every node (except the input nodes) has outdegree 1. Show that a language is computable by polynomial-size formulae iff it is in (nonuniform) NC^1. (Hint: a formula may be viewed —once we exclude the input nodes—as a directed binary tree, and in a binary tree of size m there is always a node whose removal leaves subtrees of size at most $2m/3$ each.)

§7 Show that $NC^1 =$ L. Conclude that PSPACE $\neq \mathrm{NC}^1$.

§8 Show that EXPTIME is exactly the set of languages with DC uniform circuits of size 2^{n^c} where c is some constant (c may depend upon the language).

§9 Show that if linear programming has a fast parallel algorithm then P = NC. (Hint: in your reduction, express the CIRCUIT-VALUE problem as a linear program and use the fact that $x \vee y = 1$ iff $x + y \geq 1$. Be careful; the variables in a linear program are real-valued and not boolean!)

Chapter 7

Randomized Computation

A randomized Turing Machine has a transition diagram similar to a nondeterministic TM: multiple transitions are possible out of each state. Whenever the machine can validly take more than one outgoing transition, we assume it chooses randomly among them. For now we assume that there is either one outgoing transition (a deterministic step) or two, in which case the machine chooses each with probability 1/2. (We see later that this simplifying assumption does not effect any of the theory we will develop.) Thus the machine is assumed to have a fair random coin.

The quantity of interest is the probability that the machine accepts an input. The probability is over the coin flips of the machine.

7.1 RP and BPP

DEFINITION 21 *RP is the class of all languages L, such that there is a polynomial time randomized TM M such that*

$$x \in L \Leftrightarrow \mathbf{Pr}[M \text{ accepts } x] \geq \frac{1}{2} \tag{7.1}$$

$$x \notin L \Leftrightarrow \mathbf{Pr}[M \text{ accepts } x] = 0 \tag{7.2}$$

EXAMPLE 12 Let COMPOSITES be the language $\{[n]_2 : n \text{ is a composite integer}\}$. COMPOSITES is in RP. (In a recent breakthrough, Agrawal, Kayal and Saxena proved that it is actually in P.)

There is a function called the *Jacobi symbol* that maps pairs of positive integers p, q to $\{-1, 1\}$. The value of the function on p, q is denoted $(\frac{p}{q})$. This function is computable in polynomial time.

EXAMPLE 13 POLYNOMIAL INEQUALITY $\in$ RP as well.

We can also define a class where we allow two-sided errors.

DEFINITION 22 *BPP is the class of all languages L, such that there is a polynomial time randomized TM M such that*

$$x \in L \Leftrightarrow \mathbf{Pr}[M \text{ accepts } x] \geq \frac{2}{3} \tag{7.3}$$

$$x \notin L \Leftrightarrow \mathbf{Pr}[M \text{ accepts } x] \leq \frac{1}{3} \tag{7.4}$$

As in the case of NP, we can given an alternative definiton for these classes. Instead of the TM flipping coins by itself, we think of a string of coin flips provided to the TM as an additional input.

DEFINITION 23 *BPP is the class of all languages L, such that there is a polynomial time deterministic TM M and $c > 0$ such that*

$$x \in L \Leftrightarrow \mathbf{Pr}_{r \in \{0,1\}^{|x|^c}}[M \text{ accepts } (x, r)] \geq \frac{2}{3} \tag{7.5}$$

$$x \notin L \Leftrightarrow \mathbf{Pr}_{r \in \{0,1\}^{|x|^c}}[M \text{ accepts } (x, r)] \leq \frac{1}{3} \tag{7.6}$$

We make a few observations about randomized classed. First, RP $\subseteq$ NP, since any random string that causes the RP machine to accept input x is a certificate that x is in the language. The same is not true for BPP machines, so we do not know if BPP is in NP. Note that BPP is closed under complementation (if $L \in$ BPP then $\overline{L} \in$ BPP). We do not know of any complete problems for these classes. One difficulty seems to be that the problem of determining whether or not a given randomized polynomial time TM is an RP machine is undecidable.

7.1.1 Probability Amplification

In the definition of BPP above the actual values in the place of $\frac{2}{3}$ and $\frac{1}{3}$ are not crucial, as we observe below. These two numbers can be replaced by $1 - 1/2^n$ and $1/2^n$ without changing the class.

By repeatedly running a machine M accepting a BPP language L with probabilities as above, and taking the majority vote of the different decisions it makes, we can get a much better probability of a correct answer. Using a polynomial (in input size) number of repeated trials, we get an exponentially small probability of deciding the input wrongly. This follows from bounding the tail of a binomial distribution using Chernoff bounds.

7.2 Theorems about BPP

Now we show that all BPP languages have polynomial sized circuits.

THEOREM 23
BPP $\subseteq \mathbf{P}/\text{poly}$

PROOF: For any language $L \in$ BPP consider a TM M (taking a random string as additional input) which decides any input x with an exponentially low probability of error; such an M exists by the probability amplification arguments mentioned earlier. In particular let the

probability $M(x,r)$ being wrong (taken over r) be $\leq 1/2^{(n+1)}$, where $n = |x|$. Say that an r is *bad* for x if $M(x,r)$ is an incorrect answer. Let t be the total number of choices for r. For each x, at most $t/2^{(n+1)}$ values of r are bad. Adding over all x, we conclude that at most $2^n \times t/2^{(n+1)} = t/2$ values of r are bad for some x. In other words, at least $t - t/2$ choices of r are not bad for *every* x. Let r_0 be such a choice for r. Then $M(x, r_0)$ is the correct answer for every x. We can hard-wire this r_0 into a polynomial size circuit. Given an input x, the circuit simulates M on (x, r_0). Hence we have shown $L \in \mathbf{P}/\text{poly}$. □

The next theorem relates BPP to the polynomial hierarchy.

THEOREM 24
BPP $\subseteq \mathbf{\Sigma}_2^p \cap \mathbf{\Pi}_2^p$

PROOF: It is enough to prove that BPP $\subseteq \mathbf{\Sigma}_2^p$ because BPP is closed under complementation.

Suppose $L \in$ BPP, and M is a randomized TM for L that uses m random bits and satisfies $x \in L \Rightarrow \mathbf{Pr}_r[M(x,r) \text{ accepts }] \geq 1 - 2^{-n}$ and $x \notin L \Rightarrow \mathbf{Pr}_r[M(x,r) \text{ accepts }] \leq 2^{-n}$.

Fix an input x, and let S_x denote the set of r for which M accepts (x,r). Then either $|S_x| \geq (1-2^{-n})2^m$ or $|S_x| \leq 2^{-n}2^m$, depending on whether or not $x \in L$. We will show how to check, using two alternations, which of the two cases is true.

Consider r as an element of $\mathbf{GF(2)}^m$. For a set $S \subset \mathbf{GF(2)}^m$ we define the *shift of* S *by* r_0 as $\{r + r_0 | r \in S\}$ (where the addition is as in $\mathbf{GF(2)}^m$, i.e., bit-wise XOR).

Suppose $x \in L$, so $|S_x| \geq (1-2^{-n})2^m$. We shall show that there are a small number of vectors such that the set of shifts of S_x by these vectors covers the whole space $\mathbf{GF(2)}^m$.

LEMMA 25
$\exists r_1, r_2, \ldots, r_k$, *where* $k = \lceil \frac{m}{n} \rceil + 1$ *such that* $\bigcup_{i=1}^k (S_x + r_i) = \mathbf{GF(2)}^m$.

PROOF: We shall show that $\mathbf{Pr}_{(r_1,r_2,\ldots,r_k)}[\bigcup_{i=1}^k (S_x + r_i) \neq \mathbf{GF(2)}^m] < 1/2$, whence the *existence* of the desired $r_1, r_2, \ldots, r_k$ follows.

Let $z \in \mathbf{GF(2)}^m$. If r_1 is a random vector, then so is $z + r_1$. So $\mathbf{Pr}_{r_1}[z \notin S_x + r_1] = \mathbf{Pr}_{r_1}[z + r_1 \notin S_x] \leq 2^{-n}$. So, for each z,

$$\mathbf{Pr}_{r_1,r_2,\ldots,r_k}[z \notin \bigcup_{i=1}^{k} S_x + r_i] \leq 2^{-nk}$$

So, $\mathbf{Pr}_{r_1,r_2,\ldots,r_k}[\text{some } z \notin \bigcup_{i=1}^k S_x + r_i] \leq 2^{m-nk} < 1/2 < 1$. □

Now suppose $x \notin L$. Now $|S_x| \leq 2^{-n}2^m$. This is a small set, and the union of any k shifts of S_x can be of size at most $k2^{m-n} < 2^m$, and hence cannot equal $\mathbf{GF(2)}^m$.

Thus we have established that

$$\begin{aligned} x \in L \Leftrightarrow \exists r_1, r_2, \ldots, r_k \in \mathbf{GF(2)}^m \text{ such that} \\ \forall\, z \in \mathbf{GF(2)}^m \; M \text{ accepts } x \text{ using at least one of } z + r_1, z + r_2, \ldots, z + r_k \end{aligned} \tag{7.7}$$

Thus L is in $\mathbf{\Sigma}_2^p$. □

7.3 Model Independence

Now we argue that our specific assumptions about randomized TMs do not affect the classes RP and BPP.

We made the assumption that the Turing machine uses a two-sided fair coins for randomization. Now we shall see that it can use these to simulate a machine that uses k-sided coins. To simulate a k-sided fair coin do as follows: flip our two-sided coin for $\lceil \log_2 k \rceil$ times. If the binary number generated is in the range $[0, k-1]$ output it as the result, else repeat the experiment. This terminates in expected $O(1)$ steps, and the probability that it does not halt in $O(k)$ steps is at most 2^{-k}. Since the machine needs to simulate only $\text{poly}(n)$ coin tosses, the probability that this simulation does not work can be made 2^{-n}, and so it does not substantially affect the probability of acceptance (which is something like $1/2$ or $2/3$).

Another issue that arises in real life is that one may not have an unbiased coin, and the probability it comes up heads is an unknown quantity p. But we can simulate a an unbiased coin as follows: flip the (biased) coin twice (we are assuming independence of different flips). Interpret 01 as 0 and 10 as 1; on a 00 or 11 repeat the experiment. The probability that we fail to produce a 0 or 1 is at most $1 - p^2 - (1-p)^2$. Since p is constant, this failure probability is a constant. The expected number of repetitions before we produce a bit is $O(1)$.

7.4 Recycling Entropy

Randomness may be considered a resource, because it is likely that a random bit generator is much slower than the rest of the computer. So one would like to minimize the number of random bits used. One place where we can recycle random bits is while repeating a BPP or RP algorithm.

Consider running an RP-machine on some input x. If it accepts x we know x is in the language L. But if it rejects x, we repeat the experiment, up to k times. But to repeat the experiment each time, it is not necessary to acquire all new random bits all over again. Intuitively, we can "recycle" most of the randomness in the previous random string, because when the experiment fails to detect x being in L, all we know about the random string used is that it is in the set of *bad* random strings. Thus it still has a lot of "randomness" left in it.

DEFINITION 24 *If $d > 2$ and $\beta > 0$, a family of (d, β)-*expanders *is a sequence of graphs $\{G_n\}$ where G_n is a d-regular graph on i nodes, and has the property that every subset S of at most $n/2$ nodes has edges to at least $\beta |S|$ nodes outside S.*

Deep mathematics (and more recently, simpler mathematics) has been used to construct expanders. These constructions yield algorithms that, given n and an index of a node in G_n, can produce the indices of the neighbors of this node in $\text{poly}(\log n)$ time.

Suppose the RP algorithm uses m random bits. The naïve approach to reduce its error probability to 2^{-k} uses $O(mk)$ random bits. A better approach involving recycling is as follows. Pick the first random string as a node in an *expander graph* on 2^m nodes, take a random walk of length k from there, and use the indices of the nodes encounted in this

walk (a total of mk bits) in your RP algorithm. Since the graph has constant degree, each step of the random walk needs $O(1)$ random bits. So a random walk of length k takes $m + O(k)$ random bits; m to produce the first node and $O(1)$ for each subsequent step. A surprising result shows that this sequence will be random enough to guarantee the probability amplification, but we will not give a proof here.

Of course, the same approach works for any randomized algorithm, such as a Monte Carlo simulation.

Chapter 8

Complexity classes having to do with counting

NP problems are defined around the notion of a *certificate* or *witness* for membership. In many contexts it is more natural to consider the *number* of certificates. This chapter studies #P, a complexity class that captures this notion. To study the properties of #P we will also study some related classes like $\oplus$ P.

First, we define the analogue of P for function computations.

DEFINITION 25 *FP is the set of all functions $f : \{0,1\}^* \to \mathbf{N}$ computable in polynomial time.*

Are the following problems in FP?

DEFINITION 26 #SAT *is the problem of computing, given a boolean function ϕ, the number of satisfying assignments for ϕ.*

#CYCLE *is the problem of computing, given a directed graph G, the number of simple cycles in G. (A simple cycle is one that does not visit any node twice.)*

Clearly, if #SAT $\in$ FP then SAT $\in$ P and so P = NP. Thus presumably #SAT $\notin$ FP. How about #CYCLE? The corresponding decision problem —given a directed graph decide if it has a cycle—can be solved in linear time by breadth-first-search. The next theorem suggests that the counting problem may be much harder.

THEOREM 26
If #CYCLE $\in$ FP, then P = NP.

PROOF: We show that if #CYCLE can be computed in polynomial time, then HAMCYCLE $\in$ P, where HAMCYCLE is the problem of deciding whether or not a digraph has a Hamiltonian cycle. Given a graph G with n nodes in the HAMCYCLE problem, we construct a graph G' for #CYCLE such that G has a HAMCYCLE iff G' has at least n^{n^2} cycles.

To obtain G', replace each edge (u, v) in G by a gadget as shown in Figure 8.1. The gadget has $N = n \log_2 n + 1$ levels. It is an acyclic digraph, so cycles in G' correspond to

Figure 8.1: Reducing HAMCYCLE to #CYCLE

cycles in G. Furthermore, there are 2^N directed paths from u to v in the gadget, so a simple cycle of length l in G yields $(2^N)^l$ simple cycles in G'.

Notice, if G has a Hamiltonian cycle, then G' has at least $(2^N)^n > n^{n^2}$ cycles. If G has no Hamiltonian cycle, then the longest cycle in G has length at most $n-1$. The number of cycles is bounded above by n^{n-1}. So G' can have at most $(2^N)^{n-1} \times n^{n-1} < n^{n^2}$ cycles. □

Counting problems are studied in a field called *enumerative combinatorics*, which tries to obtain closed-form mathematical expressions for them. Results such as Theorem 26 suggest that there is no such expression (at least an efficiently computable one) for the number of simple cycles in a digraph.

Counting problems are also important in practical situations having to do with estimations of probability. We give a simple example; many similar examples may be found in statistical physics and network theory.

EXAMPLE 14 In GRAPH-RELIABILITY we are given a directed graph on n nodes. Suppose we are told that each node can fail with probability 1/2 and told to compute the probability that node 1 has a path to n.

A moment's thought shows that under this simple edge failure model, the remaining graph is uniformly chosen at random from all subgraphs of the original graph.

Thus the correct answer is $\frac{1}{2^n}$(number of subgraphs in which 1 has a path to n.) We can view this as a *counting* version of the PATH problem.

8.1 #P

The following class characterizes many counting problems of interest.

DEFINITION 27 *#P is the set of all functions* $f : \{0,1\}^* \to \mathbf{N}$ *such that there is a polynomial time NDTM M such that for all inputs x,*

$$f(x) = |\{y : \text{ number of accepting branches in } M\text{'s computation tree on } x\}|$$

For example, #CYCLE ∈ #P, because one can construct a polynomial time NDTM whose each branch guesses a sequence of nodes and accepts iff they form a simple cycle. In fact, all the problems mentioned in this chapter thus far are in #P.

DEFINITION 28 *A function f is is #P-*complete *if it is in #P and* $\forall g \in \#P, g \in FP^f$

Counting versions of NP-complete languages naturally lead to #P-complete problems. This follows from the way reductions preserve the number of certificates (such a reduction is called *parsimonious*).

THEOREM 27
#SAT *is #P-complete*

PROOF:Recall the Cook-Levin reduction of an NP language to SAT. It gives a one-one correspondence between the satisfying assignments of the boolean function produced and the accepting tableaux of the NP machine. Thus the ability to compute #SAT gives the ability to determine the number of accepting paths for the NP machine. □

Now we study another problem. The *permanent* of an $n \times n$ matrix A is defined as

$$\text{perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i\sigma(i)} \tag{8.1}$$

where S_n denotes the set of all permutations of n elements. Recall that the expression for the determinant is similar

$$det(A) = \sum_{\sigma \in S_n} (-1)^{sgn(\sigma)} \prod_{i=1}^{n} a_{i\sigma(i)}$$

except for an additional "sign" term. This similarity does not translate into computational equivalence: the determinant can be computed in polynomial time, whereas computing the permanent seems much harder, as we see below.

Now we try to understand the combinatorial meaning of the permanent function. First, suppose the matrix A has each entry in $\{0, 1\}$. It may be viewed as the adjacency matrix of a bipartite graph $G(X, Y, E)$, with $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_n\}$ and $\{x_i, y_j\} \in E$ iff $A_{ij} = 1$. Then the term $\prod_{i=1}^{n} a_{i\sigma(i)}$ is 1 iff σ is a *perfect matching* (which is a set of n edges such that every node is in exactly one edge). Thus perm(A) is simply the number of perfect matchings in G. Thus perm$(\cdot) \in$ #P for $0, 1$ matrices. If A is a $\{-1, 0, 1\}$ matrix, then $\text{perm}(A) = \left|\left\{\sigma : \prod_{i=1}^{n} a_{i\sigma(i)} = 1\right\}\right| - \left|\left\{\sigma : \prod_{i=1}^{n} a_{i\sigma(i)} = -1\right\}\right|$, so one can make two calls to a #SAT oracle to compute perm(A). Thus perm$(\cdot) \in \text{FP}^{\#\text{SAT}}$ in this case. In fact one can show for general integer matrices that computing the permanent is in $\text{FP}^{\#\text{SAT}}$.

The next theorem came as a surprise to researchers in the 1970s, since it implies that if PERM $\in$ FP then P = NP. Thus the permanent may be computationally much more difficult then the determinant.

THEOREM 28 (VALIANT)
PERM *for* $0, 1$ *matrices is #P-complete.*

To prove this theorem we will use another useful way to look at the permanent. Consider matrix A as the the adjacency matrix of a weighted n-node digraph (with possible self loops). Then the expression $\prod_{i=1}^{n} a_{i\sigma(i)}$ is nonzero iff σ is a cycle-cover of A (A *cycle cover* is a subgraph in which each node has in-degree and out-degree 1; such a subgraph must be composed of cycles.) We define the *weight* of the cycle cover to be the product of the weights of the edges in it.

As a warm-up for the proof, we first use an example.

EXAMPLE 15 Consider the graph in Figure 8.2. (Unmarked edges have unit weight. We follow this convention through out this lecture). Even without knowing what the subgraph G' is, we show that the permanent of the whole graph is 0. For each cycle cover in G' there are exactly two cycle covers for the three nodes, one with weight 1 and one with weight -1. Any non-zero weight cycle cover of the whole graph is composed of a cycle cover for G' and one of these two cycle covers. Thus the sum of the weights of all cycle covers of G is 0.

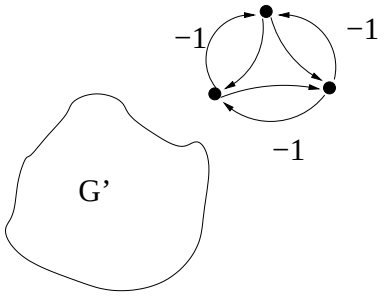

Figure 8.2: This graph has permanent 0

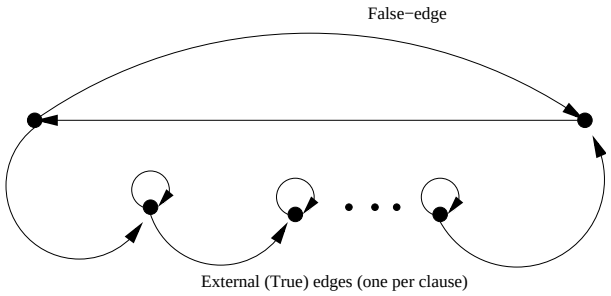

Figure 8.3: The Variable-gadget

Now we prove Valiant's Theorem.

PROOF: We shall reduce the #P-complete problem #3SAT to PERM. Given a boolean formula ϕ with n variables and m clauses, first we shall show how to construct an integer matrix A' with negative entries such that $\text{perm}(A') = 4^m \cdot (\#\phi)$. ($\#\phi$ stands for the number of satisfying assignments of ϕ). Later we shall show how to to get a 0-1 matrix A from A' such that knowing $\text{perm}(A)$ allows us to compute $\text{perm}(A')$.

The main idea is that there are two kinds of cycle covers in the digraph G' associated with A': those that correspond to satisfying assignments (we will make this precise) and those that don't. Recall that $\text{perm}(A')$ is the sum of weights of all cycle covers of the associated digraph, where the weight of a cycle cover is the product of all edge weights in it. Since A' has negative entries, some of these cycle covers may have negative weight. Cycle covers of negative weight are crucial in the reduction, since they help cancel out contributions from cycle covers that do not correspond to satisfying assignments. (The reasoning to prove this will be similar to that in Example 15.) On the other hand, each satisfying assignment contributes 4^m to $\text{perm}(A')$, so $\text{perm}(A') = 4^m \cdot (\#\phi)$.

To construct G' from ϕ, we use three kinds of gadgets as shown in Figures 8.3, 8.4 and 8.5. There is a variable gadget per variable and a clause gadget per clause. There are two possible cycle covers of a variable gadget, corresponding to an assignment of 0 or 1 to that variable. Assigning 1 corresponds to a single cycle taking all the external edges ("true-edges"), and assigning 0 correspond to taking all the self-loops and taking the "false-edge". Each external edge of a variable is associated with a clause in which the variable appears.

The clause gadget is such that the only possible cycle covers exclude at least one external edge. Also for a given (proper) subset of external edges used there is a unique cycle cover

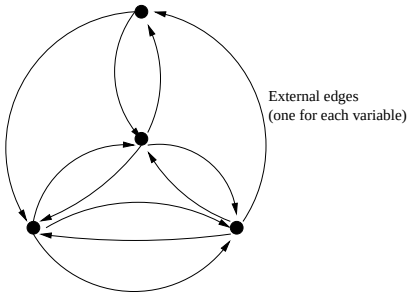

Figure 8.4: The Clause-gadget

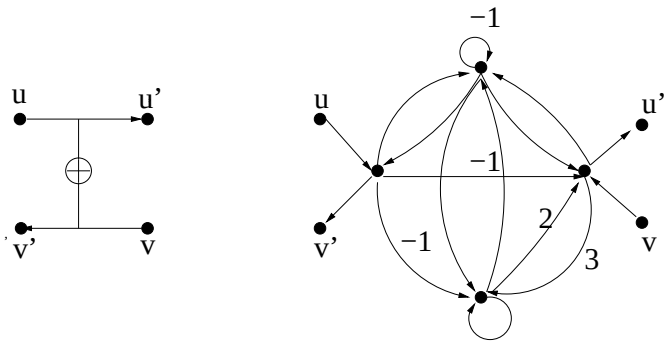

Figure 8.5: The XOR-gadget

(of weight 1). Each external edge is associated with a variable appearing in the clause.

We will also use a graph called the XOR gadget (Figure 8.5) which has the following purpose: we want to ensure that exactly one of the edges uu' and vv' (see the schematic representation in Figure 8.5) is used in a cycle cover that contributes to the total count. So after inserting the gadget, we want to count only those cycle covers which either enter the gadget at u and leave it at u' or enter it at v and leave it at v'. This is exactly what the gadget guarantees: one can check that the following cycle covers have total weight of 0: those that do not enter or leave the gadget; those that enter at u and leave at v', or those that enter at v and leave at u'. In other words, the only cycle covers that have a nonzero contribution are those that either (a) enter at u and leave at u' (which we refer to as using "edge" uu') or (b) enter at v and leave at v' (referred to as using edge vv'). These are cycle covers in the "schematic graph" (which has edges as shown in Figure 8.5) which *respect* the XOR gadget.

The XOR gadgets are used to connect the variable gadgets to the corresponding clause gadgets so that only cycle covers corresponding to a satisfying assignment need be counted towards the total number of cycle covers. Consider a clause, and a variable appearing in it. Each has an external edge corresponding to the other, connected by an XOR gadget (figure 8.6). If the external edge in the clause is not taken (and XOR is respected) the external edge in the variable must be taken (and the variable is true). Since at least one external edge of each clause gadget has to be omitted, each cycle cover respecting all the XOR gadgets corresponds to a satisfying assignment. (If the XOR is not respected, we need

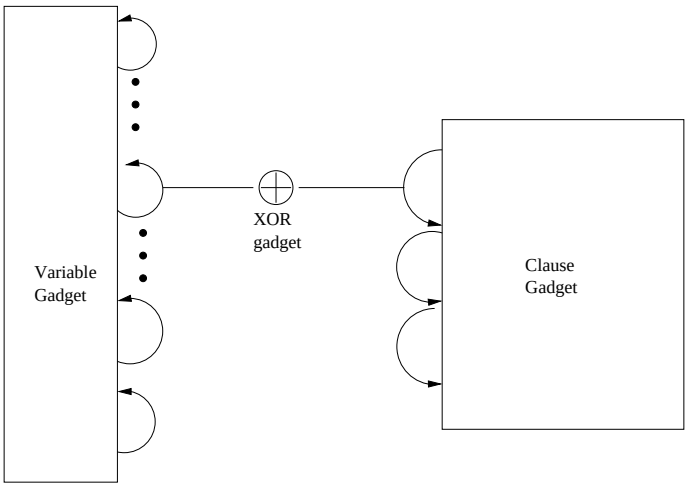

Figure 8.6: For each clause and variable appearing in it, an XOR-gadget connects the corresponding external edges. There are $3m$ such connections in total.

not count such a cycle cover as its weight will be cancelled by another cover, as we argued above). Conversely, for each satisfying assignment, there is a cycle cover (unique, in the schematic graph) which respects all the XOR gadgets.

Now, consider a satisfying assignment and the corresponding cycle cover in the schematic graph. Passing (exactly one of) the external edges through the XOR gadget multiplies the weight of each such cover by 4. Since there are $3m$ XOR gadgets, corresponding to each satisfying assignment there are cycle covers with a total weight of 4^{3m} (and all other cycle covers total to 0). So $\text{perm}(G') = 4^{3m}\#\phi$.

Finally we have to reduce finding $\text{perm}(G')$ to finding $\text{perm}(G)$, where G is an unweighted graph. First we reduce it to finding $\text{perm}(G)$ modulo $2^N + 1$ for a large enough N (but still polynomial in $|G'|$). For this, we can replace -1 edges with edges of weight 2^N, which can be converted to N edges of weight 2 in series. Changing edges of (small) positive integral weights (i.e., multiple or parallel edges) to unweighted edges is as follows: cut each (repeated) edge into two and insert a node to connect them; add a self loop to the node. This does not change the permanent, and the new graph has only unweighted edges. □

8.1.1 Approximate solutions to #P problems

Since computing exact solutions to #P-complete problems is presumably difficult, one should try to compute approximate solutions in the sense of the following definition.

DEFINITION 29 *Let $f, g : \{0,1\}^* \to \mathbf{N}$ be functions and $c > 1$. We say that f* approximates *g within a factor c if for every string x, $g(x) \leq f(x) \leq c \cdot g(x)$.*

Not all #P problems behave identically with respect to this notion. Approximating certain problems within any constant factor c is NP-hard (see Exercises). For other problems such as 0/1 permanent, there is a *Fully polynomial randomized approximation scheme* (FPRAS), which is an algorithm which, for any ϵ, δ, approximates the function within a

factor $1+\epsilon$ (its answer may be incorrect with probability δ) in time $\text{poly}(n, \log 1/\delta, \log 1/\epsilon)$. This algorithm —as well as other similar algorithms for a host of #P-complete problems— use the *Monte Carlo Markov Chain* technique, which we will not discuss here. We explore the complexity of approximation further in the exercises.

8.2 Toda's Theorem

Note that $\text{P} \subseteq \text{NP} \subseteq \text{P}^{\#\text{SAT}} \subseteq \text{PSPACE}$. An important question in the 1980s was the relative power of PH and $\text{P}^{\#\text{SAT}}$. Both are natural generalizations of NP, but it seemed that their features —alternation and the ability to count witnesses, respectively— are not directly comparable to each other. Thus it came as big surprise when in 1989 Toda showed $\text{PH} \subseteq \text{P}^{\#\text{SAT}}$. Now we prove this result; this (circuit-based) proof is due to Kannan, Venkateswaran, Vinay, and Yao.

The following complexity class will be used in the proof.

DEFINITION 30 *A language L in the class $\oplus$ P (pronounced "parity P") iff there exists a polynomial time NTM M such that $x \in L$ iff the number of accepting paths of M on input x is odd.*

DEFINITION 31 $\oplus SAT = \{\varphi : \text{ boolean formula } \varphi \text{ has an odd number of satisfying assignments}\}$.

A simple modification —essentially the same as the one in the proof of Theorem 27—of the Cook-Levin theorem shows that $\oplus$SAT is complete for $\oplus$ P. Now we give a characterization of $\oplus$ P using DC uniform circuits of exponential size; this is analogous to Theorem 22.

THEOREM 29
$L \in \oplus$ P iff L can be computed by a DC uniform circuit family $\{\mathcal{D}_n\}$ that

- *uses AND, OR, NOT, XOR gates.*
- *has size $2^{n^{O(1)}}$ and constant depth.*
- *XOR gates can have unbounded (exponential) fanin, but the AND, OR gates have fanin at most $n^{O(1)}$.*
- *NOT gates can appear anywhere in the circuit.*

PROOF: Needs proof. □

Now define a randomized reduction between two languages.

DEFINITION 32 *Language A reduces to language B under a randomized polynomial time reduction, denoted $A \leq_r B$, if there exists a deterministic, polynomial time computable function $f : \{0,1\}^* \to \{0,1\}^*$ and a polynomial $p()$ such that*

$$\forall x \in A \qquad \mathbf{Pr}_{y \in \{0,1\}^{p(|x|)}}[f(x,y) \in B] \geq 2/3 \tag{8.2}$$

$$\forall x \notin A \qquad \mathbf{Pr}_{y \in \{0,1\}^{p(|x|)}}[f(x,y) \in B] \leq 1/3 \tag{8.3}$$

We prove Toda's Theorem in two steps. First we show that any $L \in$ PH can be randomly reduced to $\oplus$SAT and then we show that this reduction can be transformed into a (deterministic) reduction to a #P problem.

8.2.1 Step 1: Random reduction from PH to $\oplus$SAT

This step consists of the following lemma.

LEMMA 30
For every language $L \in$ PH there is a probabilistic Turing machine M and $c > 0$ such that on input $< 1^n, r >$, where $|r| \leq n^c$, M can compute the connection functions of an $\oplus$ P circuit $\mathcal{C}_n$ of the type described in Theorem 29. Furthermore, for at least 2/3 of the r, this circuit $\mathcal{C}_n$ has the property that it decides L on all inputs of size n.

This Lemma will be used in the form of the following corollary, whose proof is straightforward since $\oplus$SAT is complete for $\oplus$ P.

COROLLARY 31
$\forall L \in PH, \;\; L \leq_r \oplus SAT$.

To prove Lemma 30 we will need the Valiant-Vazirani lemma.

DEFINITION 33 (NULLSPACE) *If $a_1, \ldots, a_k \in GF(2)^N$, then*

$$(a_1, \ldots, a_k)^{\perp} = \left\{ s \in GF(2)^N : \;\; s \cdot a_1 = s \cdot a_2 = \ldots = s \cdot a_k = 0 \right\}.$$

Let S be any nonempty subset of $GF(2)^N$. The Valiant-Vazirani lemma gives a probabilistic way to pick a *unique* representative from S, *and the method is the same for every subset S*: randomly pick $i \in [0, N]$ and then $a_1, a_2, \ldots, a_i \in GF(2)^N$; choose the vectors in $S \cap (a_1, \ldots, a_i)^{\perp}$ as the representatives.

The proof of the Lemma is left as an exercise.

LEMMA 32 (VALIANT-VAZIRANI)
If $S \subseteq GF(2)^N$ is nonempty then there is an $i \in \{0, 1, \ldots, N\}$ such that if $a_1, \ldots, a_i \in GF(2)^N$ are chosen uniformly at random then

$$\mathbf{Pr}[\left| S \cap (a_1, \ldots, a_i)^{\perp} \right| = 1] \geq \frac{1}{4}. \tag{8.4}$$

REMARK 2 The Valiant-Vazirani lemma was proved as part of investigation of the complexity of the USAT problem, which involves finding a satisfying assignment of boolean formulae where the satisfying assignment (if one exists) is guaranteed to be unique. This problem arises in study of cryptographic primitives.

Now we prove a simple corollary that introduces the ideas used in the proof of Lemma 30.

COROLLARY 33
$SAT \in RP^{\oplus SAT}$.

PROOF: Let φ be some formula with N variables and S be its set of satisfying assignments. (If φ is unsatisfiable, $S = \emptyset$.) The algorithm uses the deep fact that 1 is an odd number and 0 is even.

For any $a_1, a_2, \ldots, a_i \in GF(2)^N$, we can write in polynomial time a formula $\varphi_{a_1,\ldots,s_i}$ whose satisfying assignments are exactly the vectors in $S \cap (a_1, \ldots, a_i)^{\perp}$. First, using the

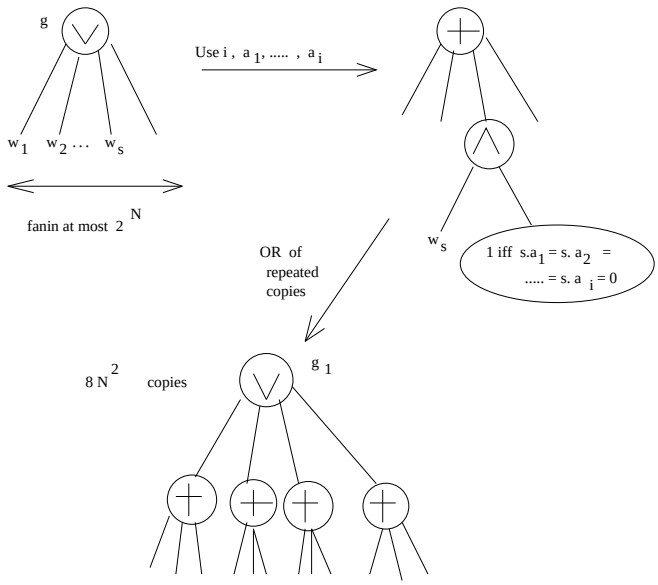

Figure 8.7: Probabilistic conversion of OR gate to $\oplus$ gate

P-completeness of CIRCUIT-VALUE, construct a circuit which, given an input s, accepts iff s satisfies φ and $s \cdot a_1 = s \cdot a_2 = \ldots = s \cdot a_i = 0$. (The formula φ and vectors $a_1, a_2, \ldots, a_n$ are "hardwired" into the circuit.) Then express the circuit as a boolean formula in the usual way.

The algorithm picks $a_1, a_2, \ldots, a_N \in GF(2)^N$ randomly and for $i = 1, 2, \ldots, N$ asks the $\oplus SAT$ oracle if $\varphi_{a_1,\ldots,s_i} \in \oplus SAT$. Every formula with an odd number of satisfying assignments is satisfiable, so if the oracle answers YES for any i then the algorithm accepts and otherwise it rejects. Clearly, the algorithm rejects unsatisfiable formulae with probability 1. By the VV Lemma it accepts satisfiable formulae with probability at least 1/4. □

Now we prove Lemma 30 itself.

PROOF:(Lemma 30) Let $\{\mathcal{W}_n\}$ be a DC uniform circuit family for L as described in Theorem 22 and M' be the associated polynomial-time Turing machine that computes its connection functions.

Let the size of circuit $\mathcal{W}_n$ be 2^N where $N = n^{O(1)}$. The main idea is to transform $\mathcal{W}_n$ by replacing each OR and AND gate of W_n (recall their fanin may be up to $2^{n^{O(1)}}$) by a depth 3 circuit using $\oplus$ gates in which all AND, OR gates have fanin $n^{O(1)}$. Doing this simultaneously for all gates gives the desired $\oplus$ P circuit. Our transformations preserve the DC uniform property.

This gate conversion use the Valiant-Vazirani lemma, and, just like the proof of Corollary 33 uses the deep fact that 1 is an odd number and 0 is even. We describe the conversion of an OR gate.

For simplicity assume the OR gate is applied to 2^N wires in the circuit and we index them using vectors in $GF(2)^N$: w_s is the wire for $s \in GF(2)^N$. Let t be a string consisting of i picked randomly from $\{0, 1, \ldots, N\}$ and strings $a_1, \ldots, a_i$ randomly from $GF(2)^N$. Let R be a string consisting of $8N^2$ independent samples of a type similar to t.

Assume that the constants 0 and 1 are available as inputs; this is without loss of generality since $1 = x_1 \vee \overline{x_1}$ and $0 = x_1 \wedge \overline{x_1}$. First we change the OR into a circuit g_t (see Figure 8.7) by replacing the the OR gate by $\oplus$ and the input w_s by logical AND of w_s and the input 1 if $s \cdot a_1 = \ldots = s \cdot a_i = 0$ and and AND of w_s and 0 otherwise. (The computation of whether to use 0 or 1 can be done in polynomial time using r, so the DC uniform property is maintained.) The following is just a restatement of the Valiant-Vazirani lemma.

CLAIM 1: *For every sequence of values* $\{w_s\}_{s \in GF(2)^N}$

$$\mathrm{OR}(\{w_s\}) = 0 \Longrightarrow \Pr_t[g_t(\{w_s\}) = 1] = 0$$

$$\mathrm{OR}(\{w_s\}) = 1 \Longrightarrow \Pr_t[g_t(\{w_s\}) = 1] \geq \frac{1}{4N}$$

Now the random string R has $8N^2$ independent samples of strings like t, so define g_t for each and take their OR; this is a depth 3 circuit called OR_R.

CLAIM 2: *For every sequence of values* $\{w_s\}_{s \in GF(2)^N}$*:*

$$\mathrm{OR}(\{w_s\}) = 0 \Longrightarrow \Pr_R[\mathrm{OR}_R(\{w_s\}) = 1] = 0$$

$$\mathrm{OR}(\{w_s\}) = 1 \Longrightarrow \Pr_R[\mathrm{OR}_R(\{w_s\}) = 1] \geq 1 - (1 - \frac{1}{4N})^{8N^2} \geq 1 - e^{-2N}$$

Apply this transformation to every gate in circuit $\mathcal{W}_n$ to obtain the $\oplus$ P circuit $\mathcal{C}_n$ (The AND gates are first transformed into OR's using De-Morgan's law.) We emphasize that we the same random string R for every gate, thus using only $n^{O(1)}$ random bits total. Clearly, the connection functions of $\mathcal{C}_n$ can be computed using R and the machine M' that described $\mathcal{W}_n$.

CLAIM 3: *For every input* $x \in \{0,1\}^n$*:*

$$\Pr_R[\mathcal{W}_n(x) \neq \mathcal{C}_n(x)] \leq 2^N e^{-2N}.$$

To prove Claim 3 observe that fixing the input x fixes, for each gate in $\mathcal{C}_n$, a sequence of values $\{w_s\}$ for the input wires to the gate. The gate transformation fails on this sequence of inputs with probability at most e^{-2N}. The Claim follows by the union bound applied to all 2^N gates.

Since there are only 2^n choices for x, we conclude that the probability (over the choice of R) is at least $1 - 2^n 2^N e^{-2N} > 2/3$ that $\mathcal{C}_n$ computes the same function as $\mathcal{W}_n$. □

8.2.2 Part 2 of Proof of Toda's Theorem

We use the following Lemma.

LEMMA 34
There is a (deterministic) polynomial-time transformation that, given $< \psi, 1^m >$*, outputs a formula* ψ^* *such that*

$$\psi \in \oplus SAT \Longrightarrow \#(\psi*) \equiv -1 (mod\ 2^{m+1})$$

$$\psi \notin \oplus SAT \Longrightarrow \#(\psi^*) \equiv 0 \ (mod\ 2^{m+1}).$$

Here $\#(\psi)$ is the number of satisfying assignments to formula ψ.

Assuming Lemma 34 we complete the proof of Toda's Theorem. Let $L \in PH$. We show that we can decide whether an input $x \in L$ by asking the #P oracle a single question about the number of accepting paths of a particular NP machine on input x. Recall that Corollary 31 gives a randomized reduction from L to $\oplus SAT$. Suppose this reduction uses a random string r of length m and produces a boolean formula ϕ_r such that

$$\begin{aligned} x \in L &\Longrightarrow \mathbf{Pr}_r[\phi_r \in \oplus\text{SAT}] \geq 2/3 \\ x \notin L &\Longrightarrow \mathbf{Pr}_r[\phi_r \in \oplus\text{SAT}] \leq 1/3 \end{aligned} \tag{8.5}$$

The NP machine, given input x, guesses a choice for r and then computes ϕ_r. Then it uses the reduction of Lemma 34 (applied on the input $< \phi_r, 1^m >$) to produce a boolean formula ϕ_r^*. Then it guesses an assignment for ϕ_r^* and accepts iff the assignment satisfies ϕ_r^*. Clearly,

$$\begin{aligned} \#\text{ accepting paths of } M \text{ on } x &= \sum_r \#(\phi_r^*) && (8.6)\\ &\equiv -|\{r|\phi_r \in \oplus\text{SAT}\}|\,(mod\ 2^{m+1}) && (8.7)\\ &= 2^{m+1} - |\{r|\phi_r \in \oplus\text{SAT}\}| && (8.8) \end{aligned}$$

where the last line follows from noticing that there are only 2^m random strings r, which is less than $\frac{1}{2} \cdot 2^{m+1}$. Thus knowing the number of accepting paths of the NP machine on input x allows us to decide —using (8.5)— whether on not x is in the PH language L. This proves Toda's Theorem.

To finish we prove Lemma 34.

PROOF:(Lemma 34) For formulas $\varphi(x_1, \ldots, x_k)$ and $\tau(y_1, \ldots, y_t)$, let $\varphi + \tau$ and $\varphi \cdot \tau$ be shorthands respectively for the following formulae:

$$(\varphi + \tau)(z, x_1, \ldots, x_k, y_1, \ldots, y_t) = (z \wedge \varphi(x_1, \ldots, x_k)) \vee (\overline{z} \wedge \tau(y_1, \ldots, y_t)) \tag{8.9}$$

$$(\varphi \cdot \tau)(x_1, \ldots, x_k, y_1, \ldots, y_t) = \varphi(x_1, \ldots, x_k) \wedge \tau(y_1, \ldots, y_t) \tag{8.10}$$

These formulae have size linear in the sizes of φ, τ and satisfy:

$$\#(\varphi + \tau) = \#(\varphi) + \#(\tau)$$

$$\#(\varphi \cdot \tau) = \#(\varphi) \cdot \#(\tau).$$

Now consider the formula $4\tau^3 + 3\tau^4$ (where τ^3 for example is shorthand for $\tau \cdot (\tau \cdot \tau)$). One can easily check that

$$\#(\tau) \equiv -1 (mod\ 2^{2^i}) \Longrightarrow \#(4\tau^3 + 3\tau^4) \equiv -1 (mod\ 2^{2^{i+1}}) \tag{8.11}$$

$$\#(\tau) \equiv 0\ (mod\ 2^{2^i}) \Longrightarrow \#(4\tau^3 + 3\tau^4) \equiv 0\ (mod\ 2^{2^{i+1}}). \tag{8.12}$$

Returning to our proof, let $\psi_0 = \psi$ and $\psi_{i+1} = 4\psi_i^3 + 3\psi_i^4$. Let

$$\psi^* = \psi_{\lceil \log(m+1) \rceil}$$

Repeated use of equations (8.11), (8.12) shows that if $\#(\psi)$ is odd, then $\#(\psi^*) \equiv -1 \ (mod\ 2^{m+1})$ and if $\#(\psi)$ is even, then $\#(\psi^*) \equiv 0 \ (mod\ 2^{m+1})$. Also, the size of ψ^* is only polynomially larger than size of ψ.

□

8.3 Open Problems

- What is the exact power of $\oplus$SAT and $\#SAT$?
- What is the average case complexity of $n \times n$ permanent modulo small prime, say 3 or 5 ? Note that for a prime $p > n$, random self reducibility of permanent implies that if permanent is hard to compute on at least one input then it is hard to compute on $1 - O(p/n)$ fraction of inputs, i.e. hard to compute on average [**?**].

Exercises

§1 Show that computing the permanent for matrices with integer entries is in $\mathrm{FP}^{\#\mathrm{SAT}}$.

§2 Prove Theorem 29.

§3 Prove the Valiant-Vazirani Lemma.

§4 Show that if there is a polynomial-time algorithm that approximates #CYCLE within a factor 2, then P = NP.

§5 Show that for every $g \in \#\mathrm{P}$ and every $\epsilon > 0$, there is a function in $FP^{\mathbf{\Sigma}_3^p}$ that approximates g within a factor $1 + \epsilon$. (Hint: Use hashing and ideas similar to those in the proof of BPP $\subseteq$ PH, where we also needed to estimate the size of a set of strings.) Thus assuming PH doesn't collapse to a finite level, approximation is easier than exact computation.

§6 Show that every for every language in AC^0 there is a depth 3 circuit of $n^{\mathrm{poly}(\log n)}$ size that decides it on $1 - 1/\mathrm{poly}(n)$ fraction of inputs and looks as follows: it has a single $\oplus$ gate at the top and the other gates are $\vee, \wedge$ of fanin at most $\mathrm{poly}(\log n)$.