Computational Complexity: A Modern Approach

Draft of a book: Dated January 2007
Comments welcome!

Sanjeev Arora and Boaz Barak
Princeton University
complexitybook@gmail.com

This is an Internet draft. Some chapters are more finished than others. References and attributions are very preliminary and we apologize in advance for any omissions (but hope you will nevertheless point them out to us).

Please send us bugs, typos, missing references or general comments to complexitybook@gmail.com — **Thank You!!**

Chapter 6

Boolean Circuits

> *"One might imagine that* $\mathbf{P} \neq \mathbf{NP}$*, but* SAT *is tractable in the following sense: for every* ℓ *there is a very short program that runs in time* ℓ^2 *and correctly treats all instances of size* ℓ*. "*
> Karp and Lipton, 1982

This chapter investigates a model of computation called the *Boolean circuit*, that is a generalization of Boolean formulae and a formalization of the silicon chips used by electronic computers. It is a natural model for *non-uniform* computation, which means a computational model that allows a different algorithm to be used for each input size, in contrast to the standard (or *uniform*) TM model where the same TM is used on all the infinitely many input sizes. Non-uniform computation crops up often in complexity theory (e.g., see chapters 18 and 19).

Another motivation for studying Boolean circuits is that they seem mathematically simpler than Turing machines, and hence it might be easier to prove *upper bounds* on their computational power. Furthermore, as we shall see in Section 6.1, sufficiently strong results of this type for circuits will resolve some important questions about Turing machines such as **P** versus **NP**. Since the late 1970's, researchers have tried to prove such results. Chapter 13 will describe the partial successes of this effort and Chapter 23 describes where and why it is stuck.

In Section 6.1 we define Boolean circuits and the class **P**/poly of languages computed by polynomial-sized circuits. We also show that **P**/poly contains the class **P** of languages computed by polynomial-time Turing machines and use them to give an alternative proof to the Cook-Levin Theorem (Theorem 2.10). In Section 6.2 we study *uniformly generated* circuit families and show such families give rise to an alternative characterization of **P**. Going on the opposite track, in Section 6.3 we show a characterization of **P**/poly using Turing machines that take advice. In Section 6.4 we study the question alluded to in the above quote by Karp and Lipton, namely, whether $\mathbf{NP} \subseteq \mathbf{P}$/poly. We show that the answer is probably "No" since otherwise there would be surprising consequences. In Section 6.5 we show that *almost all* Boolean functions require exponential-sized circuits, although finding even a single function in **NP** (or even in **NEXP**!) with this property is a major open problem in complexity. In Section 6.7.1 we study some interesting subclasses of **P**/poly such as **NC** and **AC** and show their relation to parallel computing. Finally, in Section 6.6 we give yet another characterization of the polynomial hierarchy (defined in Chapter 5) this time using exponential-sized uniformly generated circuits of constant depth.

6.1 Boolean circuits and P/poly

A Boolean circuit is a diagram showing how to derive an output from an input by a combination of the basic Boolean operations of OR ($\vee$), AND ($\wedge$) and NOT ($\neg$). For example, Figure 6.1 shows a circuit computing the XOR function. Now we give the formal definition. For convenience we

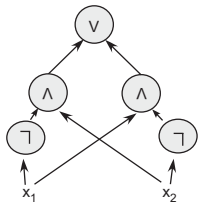

Figure 6.1: A circuit C computing the XOR function (i.e., $C(x_1, x_2) = 1$ iff $x_1 \neq x_2$).

assume the circuit produces 1 bit of output; it is trivial to generalize to circuits with more than one bit of output, though we typically will not need this generalization.

Definition 6.1 (Boolean circuits)
For every $n \in \mathbb{N}$, an *n-input single output Boolean circuit* is a directed acyclic graph with n *sources* (vertices with no incoming edges) and one *sink* (vertex with no outgoing edges). All non-source vertices are called *gates* and are labeled with one of $\vee$, $\wedge$ or $\neg$ (i.e., the logical operations OR, AND, and NOT). The $\vee$ and $\wedge$ labeled vertices have fan-in (i.e., number of incoming edges) equal to 2 and the $\neg$ labeled vertices have fan-in 1. The *size* of C, denoted by $|C|$, is the number of vertices in it.

If C is an Boolean circuit, and $x \in \{0,1\}^n$ is some input, then the *output* of C on x, denoted by $C(x)$, is defined in the natural way. More formally, for every vertex v of C we give a value $\mathsf{val}(v)$ as follows: if v is the i^{th} input vertex then $\mathsf{val}(v) = x_i$ and otherwise $\mathsf{val}(v)$ is defined recursively by applying v's logical operation on the values of the vertices connected to v. The output $C(x)$ is the value of the output vertex.

One motivation for this definition is that it models the silicon chips used in modern computers.[1] Thus if we show that a certain task can be solved by a small Boolean circuit then it can be implemented efficiently on a silicon chip.

As usual, we use asymptotic analysis to study the complexity of deciding a language by circuits.

Definition 6.2 (Circuit families and language recognition)
Let $T : \mathbb{N} \to \mathbb{N}$ be a function. A *$T(n)$-size circuit family* is a sequence $\{C_n\}_{n\in\mathbb{N}}$ of Boolean circuits, where C_n has n inputs and a single output, and its size $|C_n| \leq T(n)$ for every n.

We say that a language L is in **SIZE**$(T(n))$ if there exists a $T(n)$-size circuit family $\{C_n\}_{n\in\mathbb{N}}$ such that for every $x \in \{0,1\}^n$, $x \in L \Leftrightarrow C_n(x) = 1$.

Example 6.3
The language $\{1^n : n \in \mathbb{Z}\}$ can be decided by a linear-size circuit family. The circuit is simply a tree of AND gates that computes the AND of all input bits.

The language $\{< m, n, m + n >: m, n \in \mathbb{Z}\}$ also has linear sized circuits which implement the grade-school algorithm for addition. Recall that this algorithm adds two numbers bit by bit, and addition of two bits can be done by a circuit of $O(1)$ size. The carry bit is used as input for the addition of the bits in the next position.

[1] Actually, the circuits in silicon chips are not acyclic and use cycles to implement memory. However any computation that runs on a silicon chip with C gates and finishes in time T can be performed by a Boolean circuit of size $O(C \cdot T)$.

NOTE 6.4 (STRAIGHT-LINE PROGRAMS)

Instead of modeling Boolean circuits as labeled graphs, we can also model them a *straight-line programs*. A program P is straight-line if it contains no loop operations (such as `if` or `goto`), and hence P's running time is bounded by the number of instructions it contains.
The equivalence between Boolean circuits and straightline programs is fairly general, and holds (up to polynomial factors) for essentially any reasonable programming language. However it is most obviously demonstrated using straight line programs with *boolean operations*.
A *boolean straight line program* of length T with input variables $x_1, x_2, \ldots, x_n \in \{0,1\}$ is a sequence of T statements of the form $y_i = z_{i_1}\ OP\ z_{i_2}$ for $i = 1, 2, \ldots, T$ where OP is either $\vee$ or $\wedge$ and each z_{i_1}, z_{i_2} is either an input variable, or negation of an input variable, or y_j for $j < i$. For every setting of values to the input variables, the straight-line computation consists of executing these simple statements in order, finding values for $y_1, y_2, \ldots, y_T$. The *output* of the computation is the value of y_T.
It is straightforward to show that a function f can be computed by an S-line straightline of program of this form if and only if it can be computed by an S-sized Boolean circuits (see Exercise 6.2).

Every function from $\{0,1\}^n$ to $\{0,1\}$ can be computed by a Boolean circuits of sufficient size. Indeed, Claim 2.13 shows that a circuit of size $n2^n$ suffices (since a CNF formula is a special type of a circuit). In fact, one can even do so using a circuit of size $O(2^n/n)$ (see Exercise 6.1). Therefore, interesting complexity classes arise when we consider "small" circuits such as the following case:

DEFINITION 6.5 (THE CLASS **P**/poly)
P/poly is the class of languages that are decidable by polynomial-sized circuit families. That is, $\mathbf{P}/\text{poly} = \cup_c \mathbf{SIZE}(n^c)$.

Of course, one can make the same kind of objections to the practicality of **P**/poly as for **P**: viz., in what sense is a circuit family of size n^{100} practical, even though it has polynomial size. This was answered to some extent in Section 1.5.1. Another answer is that as complexity theorists we hope (eventually) to show that languages such as SAT are not in **P**/poly. Thus the result will only be stronger if we allow even such large circuits in the definition of **P**/poly.

6.1.1 Relation between **P**/poly and **P**

What is the relation between **P**/poly and **P**? First we show $\mathbf{P} \subseteq \mathbf{P}/\text{poly}$.

THEOREM 6.6 (**P** IS IN **P**/poly)
$\mathbf{P} \subseteq \mathbf{P}/\text{poly}$.

PROOF: The proof is very similar to the proof of the Cook-Levin Theorem (Theorem 2.10). In fact Theorem 6.6 can be used to give an alternative proof to the Cook-Levin Theorem.

Recall that by Remark 1.7 we can simulate every time $O(T(n))$ TM M by an *oblivious* TM $\tilde{M}$ (whose head movement is independent of its input) running in time $O(T(n)^2)$ (or even $O(T(n)\log T(n))$ time if we are more careful). Thus it suffices to show that for every oblivious $T(n)$-time TM M, there exists an $O(T(n))$-sized circuit family $\{C_n\}_{n\in\mathbb{N}}$ such that $C_n(x) = M(x)$ for every $x \in \{0,1\}^n$.

Let M be such an oblivious TM, let $x \in \{0,1\}^*$ be some input for M and define the *transcript* of M's execution on x to be the sequence $z_1, \ldots, z_{T(n)}$ of *snapshots* (machine's state and symbols

read by all heads) of the execution at each step in time. We can encode each such snapshot z_i by a constant size a binary string and furthermore, we can compute the string z_i based on the input x, the previous snapshot z_{i-1} and the snapshots $z_{i_1}, \ldots, z_{i_k}$ where z_{i_j} denote the last step that M's j^{th} head was in the same position as it is in the i^{th} step.[2] Because these are only a constant number of strings of constant length, this means that we can compute z_i from these previous snapshots using a constant-sized circuit.

The composition of all these constant-sized circuits gives rise to a circuit that computes from the input x the snapshot $z_{T(n)}$ of the last step of $\tilde{M}$'s execution on x. There is a simple constant-sized circuit that, given $z_{T(n)}$ outputs 1 if and only if $z_{T(n)}$ is an accepting snapshot (in which M outputs 1 and halts). Thus, there is an $O(T(n))$-sized circuit C_n such that $C_n(x) = M(x)$ for every $x \in \{0,1\}^n$. ■

REMARK 6.7
The proof of Theorem 6.6 actually gives a stronger result than its statement: the circuit is not only of polynomial size but can also be computed in polynomial time, and even in logarithmic space. One only needs to observe that it's possible to simulate every TM M by an oblivious TM $\tilde{M}$ such that the function that maps n, i to the $\tilde{M}$'s position on n-length inputs in the i^{th} tape can be computed in logarithmic space.

The inclusion $\mathbf{P} \subseteq \mathbf{P}/\text{poly}$ is proper. For instance, there are unary languages that are undecidable and hence are not in $\mathbf{P}$ (or for that matter in **EXP**), whereas every unary language is in $\mathbf{P}/\text{poly}$.

CLAIM 6.8
Let $L \subseteq \{0,1\}^*$ be a unary language (i.e., $L \subseteq \{1^n : n \in \mathbb{N}\}$). Then, $L \in \mathbf{P}/\text{poly}$.

PROOF: The following linear-size circuit family works. If $1^n \in L$ then the circuit for inputs of size n is the circuit from Example 6.3, and otherwise it is the circuit that always outputs 0. ■

And here is a unary language that is undecidable (see Section 1.4.1).

$$\mathsf{UHALT} = \{1^n : n\text{'s binary expansion encodes a pair } \langle M, x\rangle \text{ such that } M \text{ halts on input } x.\} .$$

6.1.2 Circuit Satisfiability and an alternative proof of the Cook-Levin Theorem

Boolean circuits can be used to provide an alternative proof for the Cook-Levin Theorem (Theorem 2.10) using the following language

DEFINITION 6.9 (CIRCUIT SATISFIABILITY OR CKT-SAT)
The language CKT-SAT consists of all (strings representing) circuits with a single output that have a satisfying assignment. That is, a string representing an n-input circuit C is in CKT-SAT iff there exists $u \in \{0,1\}^n$ such that $C(u) = 1$.

CKT-SAT is clearly in **NP** because the satisfying assignment can serve as the certificate. The Cook-Levin Theorem follows immediately from next two lemmas:

LEMMA 6.10
CKT-SAT *is* **NP**-*hard.*

PROOF: If $L \in \mathbf{NP}$ then there is a polynomial-time TM M and a polynomial p such that $x \in L$ iff $M(x,u) = 1$ for some $u \in \{0,1\}^{p(|x|)}$. But the proof of Theorem 6.6 yields a polynomial-time transformation from M, x to a circuit C such that $M(x,u) = C(u)$ for every $u \in \{0,1\}^{\text{poly}(|x|)}$. Thus, x is in L iff $C \in$ CKT-SAT. ■

[2] Because M is oblivious, the indices $i_1, \ldots, i_k$ depend only on i and not on the actual input x.

LEMMA 6.11
CKT-SAT $\leq_p$ 3SAT

PROOF: If C is a circuit, we map it into a 3CNF formula φ as follows: For every node v_i of C we will have a corresponding variable z_i in φ. If the node v_i is an AND of the nodes v_j and v_k then we add to φ clauses that are equivalent to the condition "$z_i = (z_j \wedge z_k)$". That is, we add the clauses

$$(\overline{z}_i \vee \overline{z}_j \vee z_k) \wedge (\overline{z}_i \vee z_j \vee \overline{z}_k) \wedge (\overline{z}_i \vee z_j \vee z_k) \wedge (z_i \vee \overline{z}_j \vee \overline{z}_k)\,.$$

Similarly, if v_i is an OR of v_j and v_k then we add clauses equivalent to "$z_i = (z_j \vee z_k)$", and if v_i is the NOT of v_j then we add the clauses $(z_i \vee z_j) \wedge (\overline{z}_i \vee \overline{z}_j)$. Finally, if v_i is the output node of C then we add the clause (z_i) to φ (i.e., we add the clause that is true iff z_i is true). It is not hard to see that the formula φ is satisfiable if and only if the circuit C is. ■

6.2 Uniformly generated circuits

The class **P**/polydoes not fit too well in the complexity world, since it contains undecidable languages such as the language UHALT defined in Section 6.1.1. The root of the problem is that for a language L to be in **P**/poly it suffices that a circuit family for L *exists* even if we have no way of actually constructing the circuits. Thus it may be fruitful to try to restrict attention to circuits that can actually be built, say using a fairly efficient Turing machine:

DEFINITION 6.12 (**P**-UNIFORM CIRCUIT FAMILIES)
A circuit family $\{C_n\}$ is **P**-*uniform* if there is a polynomial-time TM that on input 1^n outputs the description of the circuit C_n.

However, restricting circuits to be **P**-uniform "collapses" **P**/poly to the class **P**:

THEOREM 6.13
A language L is computable by a **P**-*uniform circuit family iff $L \in$* **P**.

PROOF SKETCH: If L is computable by a circuit family $\{C_n\}$ that is generated by a polynomial-time TM M, then we can come up with a polynomial-time TM $\tilde{M}$ for L as follows: on input x, the TM $\tilde{M}$ will run $M(1^{|x|})$ to obtain the circuit $C_{|x|}$ which it will then evaluate on the input x.

The other direction is obtained by following closely the proof of Theorem 6.6, and noting that it actually yields a **P**-uniform circuit family for any $L \in$ **P**. ■

6.2.1 Logspace-uniform families

We can impose an even stricter notion of uniformity: generation by logspace machines. Recall that a function $f : \{0,1\}^* \to \{0,1\}^*$ is *implicitly logspace computable* if the mapping $x, i \mapsto f(x)_i$ can be computed in logarithmic space; see Definition 4.14.

DEFINITION 6.14 (LOGSPACE-UNIFORM CIRCUIT FAMILIES)
A circuit family $\{C_n\}$ is *logspace uniform* if there is an implicitly logspace computable function mapping 1^n to the description of the circuit C_n.

Since logspace computations run in polynomial time, logspace-uniform circuits are also **P**-uniform. We note that Definition 6.14 is robust to variations in how we represent circuits by strings. A concrete way is to represent a circuit of size S is by the $S \times S$ adjacency matrix of its underlying directed graph and an array of size S that provides the labels (gate type) of each vertex. Identifying the vertices with numbers in $[S]$, we let the first n vertices be the input vertices and the last vertex be the output vertex. In other words, the family $\{C_n\}$ is logspace uniform if and only if the following functions are computable in $O(\log n)$ space:

- `SIZE`(n) returns the size S (in binary representation) of the circuit C_n.
- `TYPE`(n, i), where $i \in [m]$, returns the label of the i^{th} vertex of C_n. That is it returns one of $\{\vee, \wedge, \neg, NONE\}$.
- `EDGE`(n, i, j) returns 1 if there is a directed edge in C_n between the i^{th} vertex and the j^{th} vertex.

Note that both the inputs and the outputs of these functions can be encoded using a logarithmic (in $|C_n|$) number of bits. Exercise 6.10 asks you to prove that the class of languages decided by such circuits does not change if we use the adjacency list (as opposed to matrix) representation. A closer scrutiny of the proof of Theorem 6.6 shows that it implies the following theorem (see Exercise 6.4):

THEOREM 6.15
A language has logspace-uniform circuits of polynomial size iff it is in **P**.

6.3 Turing machines that take advice

We can define **P**/poly in an equivalent way using Turing machines that "take advice":

DEFINITION 6.16
Let $T, a : \mathbb{N} \to \mathbb{N}$ be functions. The class of *languages decidable by time-$T(n)$ TM's with $a(n)$ bits of advice*, denoted $\mathbf{DTIME}(T(n))_{/a(n)}$, contains every L such that there exists a sequence $\{\alpha_n\}_{n\in\mathbb{N}}$ of strings with $\alpha_n \in \{0,1\}^{a(n)}$ and a TM M satisfying

$$M(x, \alpha_n) = 1 \Leftrightarrow x \in L$$

for every $x \in \{0,1\}^n$, where on input (x, α_n) the machine M runs for at most $O(T(n))$ steps.

EXAMPLE 6.17
Every unary language can be 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. In particular this is true of the language UHALT defined in Section 6.1.1.

Turing machines with advice yield the following characterization of **P**/poly:

THEOREM 6.18 (POLYNOMIAL-TIME TM'S WITH ADVICE DECIDE **P**/poly)
P/poly $= \cup_{c,d}\mathbf{DTIME}(n^c)/n^d$

PROOF: If $L \in$ **P**/poly then it's computable by a polynomial-sized circuit family $\{C_n\}$. We can just use the description of C_n as an advice string on inputs of size n, where the TM is simply the polynomial-time TM M that on input a string x and a string representing an n-input circuit C outputs $C(x)$.

Conversely, if L is decidable by a polynomial-time Turing machine M with access to an advice family $\{\alpha_n\}_{n\in\mathbb{N}}$ of size $a(n)$ for some polynomial a, then we can use the construction of Theorem 6.6 to construct for every n, a polynomial-sized circuit D_n such that on every $x \in \{0,1\}^n$, $\alpha \in \{0,1\}^{a(n)}$, $D_n(x, \alpha) = M(x, \alpha)$. We let the circuit C_n be the polynomial circuit that maps x to $D_n(x, \alpha_n)$. That is, C_n is equal to the circuit D_n with the string α_n "hardwired" as its second input. (By "hardwiring" an input into a circuit we mean taking a circuit C with two inputs $x \in \{0,1\}^n, y \in \{0,1\}^m$ and transforming it into the circuit C_y that for every x returns $C(x, y)$. It is easy to do so while ensuring that the size of C_y is not greater than the size of C and this simple idea is often used in complexity theory.) ■

6.4 **P**/poly and **NP**

Karp and Lipton formalized the question of whether or not SAT has small circuits as: Is SAT in **P**/poly? They showed that the answer is "NO" if the polynomial hierarchy does not collapse.

THEOREM 6.19 (KARP-LIPTON THEOREM [KL80])
If $\mathbf{NP} \subseteq \mathbf{P}/\text{poly}$ *then* $\mathbf{PH} = \mathbf{\Sigma}_2^p$.

PROOF: By Theorem 5.4, to show $\mathbf{PH} = \mathbf{\Sigma}_2^p$ it suffices to show that $\mathbf{\Pi}_2^p \subseteq \mathbf{\Sigma}_2^p$ and in particular it suffices to show that $\mathbf{\Sigma}_2^p$ contains the $\mathbf{\Pi}_2^p$-complete language Π_2SAT consisting of all true formulae of the form

$$\forall u \in \{0,1\}^n \, \exists v \in \{0,1\}^n \, \varphi(u,v) = 1 \,. \tag{1}$$

where φ is an unquantified Boolean formula.

If $\mathbf{NP} \subseteq \mathbf{P}/\text{poly}$ then there *exists* a polynomial p and a $p(n)$-sized circuit family $\{C_n\}_{n\in\mathbb{N}}$ such that for every Boolean formula φ and $u \in \{0,1\}^n$, $C_n(\varphi, u) = 1$ if and only if there exists $v \in \{0,1\}^n$ such that $\varphi(u,v) = 1$. Thus the circuit solves the *decision problem*. However, our algorithm of Theorem 2.18 converts any decision algorithm for SAT into an algorithm that actually outputs a satisfying assignment whenever one exists. Thinking of this algorithm as a circuit, we obtain from the family $\{C_n\}$ a $q(n)$-size circuit family $\{C'_n\}_{n\in\mathbb{N}}$, where $q(\cdot)$ is a polynomial, such that for every such formula φ and $u \in \{0,1\}^n$, if there is a string $v \in \{0,1\}^n$ such that $\varphi(u,v) = 1$ then $C'_n(\varphi, u)$ outputs such a string v. (Note: We did not formally define circuits with more than one bit of output, but it is an obvious generalization of Definition 6.1.)

Of course, the assumption $\mathbf{NP} \subseteq \mathbf{P}/\text{poly}$ only implies the *existence* of such circuits. The main idea of Karp-Lipton is that this circuit can be "guessed" using $\exists$ quantification. Since the circuit outputs a satisfying assignment if one exists, this answer can be checked directly. Formally, since C'_n can be described using $10q(n)^2$ bits, if (1) holds then the following quantified formula is true:

$$\exists w \in \{0,1\}^{10q(n)^2} \, \forall u \in \{0,1\}^n \text{ s.t. } w \text{ describes a circuit } C' \text{ and } \varphi(u, C'(\varphi, u)) = 1 \,. \tag{2}$$

Furthermore, if (1) is false then for some u, *no* v exists such that $\varphi(u,v) = 1$, and hence (2) is false as well. Thus (2) holds if and only if (1) does! Finally, since evaluating a circuit on an input can be done deterministically in polynomial time, the truth of (2) can be verified in $\mathbf{\Sigma}_2^p$. ■

Similarly, the following theorem shows that **P**/poly is unlikely to contain **EXP**:

THEOREM 6.20 (MEYER'S THEOREM [KL80])
If $\mathbf{EXP} \subseteq \mathbf{P}/\text{poly}$ *then* $\mathbf{EXP} = \mathbf{\Sigma}_2^p$.

PROOF SKETCH: Let $L \in \mathbf{EXP}$. Then L is computable by an $2^{p(n)}$-time oblivious TM M, where p is some polynomial. Let $x \in \{0,1\}^n$ be some input string. For every $i \in [2^{p(n)}]$ we denote by z_i the encoding of the i^{th} snapshot of M's execution on input x (see the proof of Theorem 6.6). If M has k tapes then $x \in L$ if and only if for every $k+1$ indices $i, i_1, \ldots, i_k$, the snapshots $z_i, z_{i_1}, \ldots, z_{i_k}$ satisfy some easily checkable criteria: if z_i is the last snapshot than it should encode M outputting 1, and if $i_1, \ldots, i_k$ are the last indices where M's heads where in the same locations as in i then the values read in z_i should be consistent with input and the values written in $z_{i_1}, \ldots, z_{i_k}$. (Note that these indices can be represented in polynomial-time.) But if $\mathbf{EXP} \subseteq \mathbf{P}/\text{poly}$ then there is a $q(n)$-sized circuit C (for a polynomial q) that computes z_i from i. Hence, $x \in L$ iff the following condition is true

$$\exists C \in \{0,1\}^{q(n)} \, \forall i, i_1, \ldots, i_k \in \{0,1\}^{p(n)} \, T(x, C(i), C(i_1), \ldots, C(i_k)) = 1 \,,$$

where T is some polynomial-time TM checking the above conditions. This implies that $L \in \mathbf{\Sigma}_2^p$. ■

Theorem 6.20 implies that if $\mathbf{P} = \mathbf{NP}$ then $\mathbf{EXP} \not\subseteq \mathbf{P}/\text{poly}$. Indeed, by Theorem 5.4 if $\mathbf{P} = \mathbf{NP}$ then $\mathbf{P} = \mathbf{\Sigma}_2^p$, and so if $\mathbf{EXP} \subseteq \mathbf{P}/\text{poly}$ we'd get $\mathbf{P} = \mathbf{EXP}$, contradicting the Time Hierarchy theorem (Theorem 3.1). Thus upper bounds (in this case, $\mathbf{NP} \subseteq \mathbf{P}$) can potentially be used to prove circuit lower bounds.

6.5 Circuit lower bounds

Since $\mathbf{P} \subseteq \mathbf{P}/\text{poly}$, if we ever prove $\mathbf{NP} \not\subseteq \mathbf{P}/\text{poly}$ then we have shown $\mathbf{P} \neq \mathbf{NP}$. The Karp-Lipton theorem gives hope that $\mathbf{NP} \not\subseteq \mathbf{P}/\text{poly}$. Can we resolve $\mathbf{P}$ versus $\mathbf{NP}$ by proving $\mathbf{NP} \not\subseteq \mathbf{P}/\text{poly}$? There is reason to invest hope in this approach as opposed to proving direct lower bounds on Turing machines. By representing computation using circuits we seem to actually peer into the guts of it rather than treating it as a black box. Thus we may be able to get around the limitations of relativizing methods shown in Chapter 3. Indeed, it is easy to show that *some* functions do require very large circuits to compute:

THEOREM 6.21 (EXISTENCE OF HARD FUNCTIONS)
For every $n > 1$, there exists a function $f : \{0,1\}^n \to \{0,1\}$ that cannot be computed by a circuit C of size at most $2^n/(10n)$.

PROOF: The proof uses a simple counting argument:

- The number of functions from $\{0,1\}^n$ to $\{0,1\}$ is 2^{2^n}.
- Since every circuit of size at most S can be represented as a string of $9 \cdot S \log S$ bits (e.g., using the adjacency list representation) the number of such circuits is at most $2^{9S \log S}$.

Setting $S = 2^n/(10n)$, we see that the number of such circuits is at most $2^{9S \log S} \leq 2^{2^n 9n/10n} < 2^{2^n}$. Hence the number of functions computed by such circuits is smaller than 2^{2^n} implying that there exists a function that is not computed by circuits of that size. ■

There is another way to phrase this proof. Suppose that we pick a function $f : \{0,1\}^n \to \{0,1\}$ at random, by picking for every one of the 2^n possible inputs $x \in \{0,1\}^n$ the value $f(x)$ in $\{0,1\}$ uniformly and independently. Then, for every fixed circuit C and input x the probability that $C(x) = f(x)$ is $1/2$ and since these choices are independent, the probability that C computes f (i.e., $C(x) = f(x)$ for *every* $x \in \{0,1\}^n$) is 2^{-2^n}. Since there are at most $2^{0.92^n}$ circuits of size at most $2^n/(10n)$, we can apply the union bound (see Section A.3) to conclude that the probability that *there exists* such a circuit C computing f is at most

$$\frac{2^{0.92^n}}{2^{2^n}} = 2^{-0.12^n},$$

a number that is tends very fast to zero as n grows. In particular, since this number is smaller than one, it implies that there exists a function f that is not computed by any circuit of size at most $2^n/(10n)$. This proof technique (showing an object with a particular property exists by showing a random object satisfies this property with nonzero probability) is called the *probabilistic method* and it is widely used in theoretical computer science and combinatorics (e.g., see chapters 12, 18 and 19 of this book). Note that it yields a stronger result than Theorem 6.21: not only there exists a hard function (not computable by $2^n/(10n)$-sized circuits) but in fact the *vast majority* of the functions from $\{0,1\}^n$ to $\{0,1\}$ are hard. This gives hope that we should be able to find one such function that also happens to lie in $\mathbf{NP}$, thus proving $\mathbf{NP} \not\subseteq \mathbf{P}/\text{poly}$. Sadly, such hopes have not

yet come to pass. After two decades, the best circuit size lower bound for an **NP** language is only $(5-o(1))n$ [**??**]. (However, see Exercise 6.5 for a better lower bound for a language in **PH**.) On the positive side, we have notable success in proving lower bounds for more restricted circuit models, as we will see in Chapter 13.

6.6 Non-uniform hierarchy theorem

As in the case of deterministic time, non-deterministic time and space bounded machines, Boolean circuits also have a hierarchy theorem. That is, larger circuits can compute strictly more functions than smaller ones:

THEOREM 6.22 (NON-UNIFORM HIERARCHY THEOREM)
For every functions $T, T' : \mathbb{N} \to \mathbb{N}$ *with* $2^n/(100n) > T'(n) > T(n) > n$ *and* $T(n)\log T(n) = o(T'(n))$,

$$\mathbf{SIZE}(T(n)) \subsetneq \mathbf{SIZE}(T'(n))$$

PROOF: Interestingly, the diagonalization methods of Chapter 3 do not seem to apply in this setting, but nevertheless, we are able to prove 6.22 using the counting argument of Theorem 6.21. To show the idea, we prove that $\mathbf{SIZE}(n) \subsetneq \mathbf{SIZE}(n^2)$.

By Theorem 6.21, for every ℓ there is a function $f : \{0,1\}^\ell \to \{0,1\}$ that is not computable by $2^\ell/(10\ell)$-sized circuits. On the other hand, by Claim 2.13, every function from $\{0,1\}^\ell$ to $\{0,1\}$ is computable by a $2^\ell 10\ell$-sized circuit.

Therefore, if we set $\ell = 1.1\log n$ and let $g : \{0,1\}^n \to \{0,1\}$ be the function that applies f on the first ℓ bits of its input, then

$$\begin{array}{lrcl} g \in & \mathbf{SIZE}(2^\ell 10\ell) = & \mathbf{SIZE}(11n^{1.1}\log n) \subseteq & \mathbf{SIZE}(n^2) \\ g \notin & \mathbf{SIZE}(2^\ell/(10\ell)) = & \mathbf{SIZE}(n^{1.1}/(11\log n)) \supseteq & \mathbf{SIZE}(n) \end{array}$$

■

6.7 Finer gradations among circuit classes

This section introduces some subclasses of $\mathbf{P}$/poly, which are interesting for two reasons. First, separating **NP** from these subclasses may give insight into how to separate **NP** from $\mathbf{P}$/poly. Second, these subclasses correspond to interesting computational models in their own right.

Perhaps the most interesting connection is to *massively parallel computers*, which we will now briefly describe. (A detailed understanding is not necessary as the validity of Theorem 6.27 below does not depend upon it.) In a parallel computer 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 —assuming some upper bounds 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) and 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 will say that a computational has an *efficient parallel algorithm* if it can be solved for inputs of size n using a parallel computer with $n^{O(1)}$ processors in time $\log^{O(1)} n$.

EXAMPLE 6.23
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* 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 unlike the gradeschool algorithm!).

Many matrix computations can be done efficiently in parallel: these include computing the product, rank, determinant, inverse, etc. (See exercises.)

Some graph theoretic algorithms such as shortest paths and minimum spanning tree also have fast parallel implementations.

However well-known polynomial-time problems such as maximum flows and linear programming are not known to have any good parallel implementations and are conjectured not to have any; see our discussion of **P**-completeness in Section 6.7.2.

6.7.1 The classes NC and AC

Now we relate 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 6.24 (THE CLASS **NC**)
For every d, a language L is in $\mathbf{NC}^d$ there L is decided by a logspace-uniform family of circuits $\{C_n\}$ where C_n has depth $O(\log^d n)$. The class **NC** is $\cup_{i \geq 1} \mathbf{NC}^i$.

A related class is the following.

DEFINITION 6.25 (**AC**)
The class $\mathbf{AC}^i$ is defined similarly to $\mathbf{NC}^i$ except gates are allowed to have unbounded fan-in (i.e., the OR and AND gates can be applied to more than two bits). The class **AC** is $\cup_{i \geq 0} \mathbf{AC}^i$.

Since unbounded (but poly(n)) fan-in can be simulated using a tree of ORs/ANDs of depth $O(\log n)$, $\mathbf{NC}^i \subseteq \mathbf{AC}^i \subseteq \mathbf{NC}^{i+1}$, and the inclusion is known to be strict for $i = 0$ as we will see in Chapter 13. Note that $\mathbf{NC}^0$ is extremely limited since the circuit's output depends upon a constant number of input bits, but $\mathbf{AC}^0$ does not suffer from this limitation.

EXAMPLE 6.26
The language PARITY $=\{x : x$ has an odd number of 1s$\}$ is in $\mathbf{NC}^1$. The circuit computing it has the form of a binary tree. The answer appears at the root; the left subtree computes the parity of the first $|x|/2$ bits and the right subtree computes the parity of the remaining bits. The gate at the top computes the parity of these two bits. Clearly, unwrapping the recursion implicit in our description gives a circuit of dept $O(\log n)$.

In Chapter 13 we will show that PARITY is not in $\mathbf{AC}^0$

It turns out that **NC** characterizes the languages with efficient parallel algorithms:

THEOREM 6.27 (**NC** AND PARALLEL ALGORITHMS)
A language has efficient parallel algorithms iff it is in **NC**.

PROOF SKETCH: Suppose a language $L \in \mathbf{NC}$ 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. ■

6.7.2 P-completeness

A major open question is whether every polynomial-time algorithm has an efficient parallel implementation, or in other words whether $\mathbf{P} = \mathbf{NC}$. We believe that the answer is NO (though we are currently even unable to separate $\mathbf{PH}$ from $\mathbf{NC}^1$). This motivates the theory of $\mathbf{P}$-*completeness*, a study of which problems are likely to be in $\mathbf{NC}$ and which are not.

DEFINITION 6.28
A language is $\mathbf{P}$-*complete* if it is in $\mathbf{P}$ and every language in $\mathbf{P}$ is logspace-reducible to it (as per Definition 4.14).

The following theorem is left for the reader as Exercise 6.15.

THEOREM 6.29
If language L is $\mathbf{P}$-complete then

1. $L \in \mathbf{NC}$ *iff* $\mathbf{P} = \mathbf{NC}$.
2. $L \in \mathbf{L}$ *iff* $\mathbf{P} = \mathbf{L}$. *(Where $\mathbf{L}$ is the set languages decidable in logarithmic space, see Definition 4.4.)*

The following is a fairly natural $\mathbf{P}$-complete language:

THEOREM 6.30
Let CIRCUIT-EVAL *denote the language consisting of all pairs $\langle C, x\rangle$ such that C is an n-inputs single-output circuit and $x \in \{0,1\}^n$ satisfies $C(x) = 1$. Then* CIRCUIT-EVAL *is $\mathbf{P}$-complete.*

PROOF SKETCH: The language is clearly in $\mathbf{P}$. A logspace-reduction from any other language in $\mathbf{P}$ to this language is implicit in the proof of Theorem 6.15. ■

6.8 Circuits of exponential size

As noted, every language has circuits of size $O(2^n/n)$. But actually finding these circuits may be difficult— sometimes even undecidable. If we place a uniformity condition on the circuits, that is, require them to be efficiently computable then the circuit complexity of some languages could exceed 2^n. In fact it is possible to give alternative definitions of some familiar complexity classes, analogous to the definition of $\mathbf{P}$ in Theorem 6.15.

DEFINITION 6.31 (DC-UNIFORM)
Let $\{\mathcal{C}_n\}_{n\geq 1}$ be a circuit family. We say that it is a *Direct Connect uniform* (DC uniform) family if, given $\langle n, i\rangle$, we can compute in polynomial time the i^{th} bit of (the adjacency matrix representation of) the circuit C_n. That is, a family $\{C_n\}_{n\in\mathbb{N}}$ is DC uniform iff the functions `SIZE`, `TYPE` and `EDGE` defined in Section 6.2.1 are computable in polynomial time.

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 vertex 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.

THEOREM 6.32
$L \in PH$ 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.*
- *gates can have unbounded (exponential) fan-in.*
- *the NOT gates appear only at the input level (i.e., they are only applied directly to the input and not to the result of any other gate).*

We leave proving Theorem 6.32 as Exercise 6.16. If we drop the restriction that the circuits have constant depth, then we obtain exactly **EXP** (see Exercise 6.17).

WHAT HAVE WE LEARNED?

- Boolean circuits can be used as an alternative computational model to TMs. The class $\mathbf{P}/\text{poly}$ of languages decidable by polynomial-sized circuits is a strict superset of **P** but does not contain **NP** unless the hierarchy collapses.
- Almost every function from $\{0,1\}^n$ to $\{0,1\}$ requires exponential-sized circuits. Finding even one function in **NP** with this property would show that $\mathbf{P} \neq \mathbf{NP}$.
- The class **NC** of languages decidable by (uniformly constructible) circuits with polylogarithmic depth and polynomial size corresponds to computational tasks that can be efficiently parallelized.

Chapter notes and history

Circuits have been studied in electrical engineering since the 1940s, at a time when gates were implemented using vacuum tube devices. Shannon's seminal paper [**shannon49**] mentions the problem of finding the smallest circuit implementing a boolean function. Such topics are studied in a field called "switching theory" (see for instance Harrison's 1965 book). Savage [Sav72] makes some of the first connections between Turing machine computations and circuits, and describes the tight relationship between circuits and straight line programs.

The class $\mathbf{P}/\text{poly}$ and its characterization as the set of languages computed by polynomial-time TM's with polynomial advice (Theorem 6.18) is due to Karp and Lipton [**KL82**]. They also give a more general definition that can be used to define the class $\mathcal{C}/a(n)$ for every complexity class $\mathcal{C}$ and function $a : N \to \mathbb{N}$. However, we do not use this definition in this book since it does not seem to capture the intuitive notion of advice for classes such as $\mathbf{NP} \cap \mathbf{coNP}$, **BPP** and others.

Karp and Lipton [KL80] originally proved Theorem 6.19 with the weaker conclusion $\mathbf{PH} = \mathbf{\Sigma}_3^p$; they attribute the stronger version given here to Sipser. They also state Theorem 6.20 and attribute it to A. Meyer.

NC stands for "Nick's Class," defined first by Nick Pippenger and named by Steve Cook in his honor. However, the "A" in **AC** stands not for a person but for "alternations." The class of

NC algorithms as well as many related issues in parallel computation are discussed in the text by Leighton [**leighton91**].

Boppana and Sipser give an excellent survey of the knowledge on circuits lower bounds circa 1989 [BS90]. Unfortunately, unlike other areas of complexity theory, there has not been that much progress in this area since then.

Exercises

6.1. Prove that every function from $\{0,1\}^n$ to $\{0,1\}$ can be computed by a circuit of size $1000 \cdot 2^n/n$.

6.2. Prove that for every $f : \{0,1\}^n \to \{0,1\}$ and $S \in \mathbb{N}$, f can be computed by a Boolean circuit of size S if and only if f can be computed by an S-line program of the type described in Example 6.4.

6.3. Describe a *decidable* language in $\mathbf{P}/\text{poly}$ that is not in $\mathbf{P}$.

6.4. Prove Theorem 6.15.

6.5. [Kannan [Kan81]] 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.

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

6.7. Show that if $\mathbf{P} = \mathbf{NP}$ then there is a language in **EXP** that requires circuits of size $2^n/n$.

6.8. A language $L \subseteq \{0,1\}^*$ is sparse if there is a polynomial p such that $|L \cap \{0,1\}^n| \leq p(n)$ for every $n \in \mathbb{N}$. Show that every sparse language is in $\mathbf{P}/\text{poly}$.

6.9. (Mahaney's Theorem [**??**]) Show that if a sparse language is **NP**-complete then $\mathbf{P} = \mathbf{NP}$. (This is a strengthening of Exercise 2.29 of Chapter 2.)

> **Hint:** Show a recursive exponential-time algorithm S that on input a n-variable formula φ and a string $v \in \{0,1\}^n$ outputs 1 iff φ has a satisfying assignment v such that $v > u$ when both are interpreted as the binary representation of a number in $[2^n]$. Use the reduction from SAT to L to prune possibilities in the recursion tree of S.

6.10. Show a logspace implicitly computable function f that maps any n-vertex graph in adjacency matrix representation into the same graph in adjacency list representation. You can think of the adjacency list representation of an n-vertex graph as a sequence of n strings of size $O(n \log n)$ each, where the i^{th} string contains the list of neighbors of the i^{th} vertex in the graph (and is padded with zeros if necessary).

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

6.12. (a) Describe an **NC** circuit for the problem of computing the product of two given $n \times n$ matrices A, B over a finite field $\mathbb{F}$ of size at most polynomial in n.

> **Hint:** You can use a different processor to compute each entry of AB.

(b) Describe an **NC** circuit for computing, given an $n \times n$ matrix, the matrix A^n over a finite field $\mathbb{F}$ of size at most polynomial in 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: Let A be the adjacency matrix of a graph. What is the meaning of the (i, j)th entry of A^n?

6.13. 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 non-uniform $\mathbf{NC}^1$, where this denotes the variant of $\mathbf{NC}^1$ dropping requirements that the circuits are generated by logspace algorithms.

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.

6.14. Show that $\mathbf{NC}^1 \subseteq \mathbf{L}$. Conclude that $\mathbf{PSPACE} \neq \mathbf{NC}^1$.

6.15. Prove Theorem 6.29. That is, prove that if L is **P**-complete then $L \in \mathbf{NC}$ (resp. **L**) iff $\mathbf{P} = \mathbf{NC}$ (resp. **L**).

6.16. Prove Theorem 6.32 (that **PH** is the set of languages with constant-depth DC uniform circuits).

6.17. Show that **EXP** 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).

6.18. Show that if linear programming has a fast parallel algorithm then $\mathbf{P} = \mathbf{NC}$.

Hint: in your reduction, express the CIRCUIT-EVAL 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!