Lecture Notes: Boolean Circuits

Neeldhara Misra

STATUTORY WARNING
This document is a draft version.

1 Introduction

Most of us have encountered circuits before. Visually, Figure 1 shows a typical circuit, a network of nodes engineered to perform a specific computation. There are designated *input nodes*, that take binary values, and we can work our way from these gates to a special gate called the *output nodes*, by performing all the intermediate computations that we encounter. These computational *gates* can perform one of the three basic boolean operations, namely **and**, **or**, and **not**.

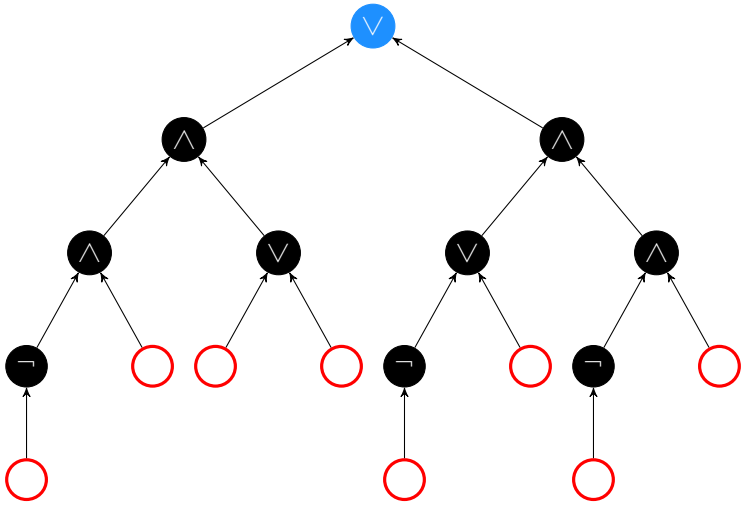

Figure 1: A prototypical Boolean Circuit

In this lecture, we will study boolean circuits as a candidate model of computation. These lecture notes are based on the chapter on Boolean Circuits in [1].

Definition 1 (Boolean circuits)**.**

- *For every* $n \in \mathbb{N}$*, an* n*-input, single-output Boolean circuit* C *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 vertices labeled with* $\vee$ *and* $\wedge$ *have fan-in (i.e.,number of incoming edges) equal to two, and the vertices labeled with* $\neg$ *have fan-in one.*
- *The* size *of* C*,denoted by* $|C|$*, is the number of vertices in it.*
- *If* C *is a 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 it a value* $val(v)$ *as follows:*

$$val(x) = \begin{cases} x & \text{if } x \text{ is an input node,} \\ val(x_1) + val(x_2) & \text{if } x \text{ is a } \vee \text{gate, and has children } x_1 \text{ and } x_2, \\ val(x_1) \times val(x_2) & \text{if } x \text{ is a } \wedge \text{gate, and has children } x_1 \text{ and } x_2, \\ 1 - val(y) & \text{if } x \text{ is a } \neg \text{gate, and } y \text{ is its child node.} \end{cases}$$

Exercise 1. *Can you construct a circuit that computes* Parity*? That is, this circuit outputs* 1 *if, and only if, an odd number of inputs are* 1*. (Hint: Construct a circuit for two inputs first. Can you build it up from here?)*

Evidently, every circuit with n input nodes computes some boolean function from $f : \{0,1\}^n \to \{0,1\}$. We would now like to propose the notion of a *language* associated with a circuit. It's quite natural to suggest that a circuit *accepts* those inputs on which evaluates to one. However, this limits the scope of our language only to strings of length n, which would be unfortunate. But this is easily fixed, by extending our definition to permit *a different circuit for every length of input*.

Remark 1. *We are experiencing our first significant departure from the spirit of the Turing Machine model — circuits are inherently* non-uniform*, that is, we are permitted to engineer different internal mechanisms of computation for differently-sized inputs!*

Let us also consider (asymptotic) measures of the complexity of a circuit. The most obvious measure appears to be the *size* of a circuit, and this will be our focus for the most part. One might also consider the *depth* of a circuit, which is the length of a longest path from an input node to the output node. The depth of a circuit appears to reflect the degree to which a circuit can be parallelized.

Also, we will begin by assuming that the circuits we are dealing with have fan-in at most two. Note that restricting the fan-in gives us a "logarithmic disadvantage" in terms of depth — indeed, any node with fan-in t can be replaced by an equivalent "sub-circuit" of fan-in two (how?), but will cause the depth to increase by $\log t$ (why?). However, when size is the only consideration, then this assumption is reasonable (up to poly-logarithmic factors).

We are now ready to formally define the notion of the class of languages that can be recognized by a *family* of circuits, measured by size.

Definition 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* $|C_n| \leqslant T(n)$ *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* $x \in \{0,1\}^n$*,*

$$x \in L \Leftrightarrow C_n(x) = 1.$$

How do circuits compare to Turing machines in expressiveness - does the non-uniformity indeed wield additional powers? The following simple illustration sets the two models fundamentally apart. We know that there are (many) languages that cannot be recognized by *any* Turing machine with finite resources. In sharp contrast, given enough resources, any boolean function admits a circuit family that decides it. To see this, consider a boolean function f, and let $L_f := \{x_1, \ldots, x_t\}$ be the set of strings of length n on which f evaluates to **True**. Let S_i be the circuit that outputs 1 precisely on input $x_i = b_1 b_2 \cdots b_n$. Note that S_i is easy to engineer with the use of an AND

gate on the inputs i for which b_i is one, and the negation of the inputs i for which $b_i = 0$. A circuit for accepting strings in L_f can now be designed as follows: we redirect the inputs to each S_i, and then merge the outputs of the S_i's by an OR gate. Note that this circuit has size proportional to $|L_f|$, and therefore potentially exponential in the input.

Since circuits can evidently decide anything given enough resources, our natural next step is to limit these resources to identify interesting classes of languages. The first consideration along these lines is to look at circuits of polynomial size:

Definition 3 (The class $\mathsf{P}_{/\mathbf{poly}}$)**.** $\mathsf{P}_{/\mathbf{poly}}$ *is the class of languages that are decidable by polynomial-sized circuit families. That is,*

$$\mathsf{P}_{/poly} = \cup_c \mathbf{SIZE}(n^c).$$

Example 1. *The language of "all ones", that is,* $\{1^n \mid n \in \mathbb{Z}\}$ *can be decided by a linear-sized circuit family. The circuit is simply a tree of AND gates that computes the AND of all input bits.*

We will soon show that the class P is contained in $\mathsf{P}_{/\mathbf{poly}}$. However, this is also a good time to note that despite the restriction of resources, circuits continue to be quite powerful: indeed, even linear-sized circuit families can recognize undecidable languages! To see this:

- Note that any unary language L can be decided by a linear-sized circuit family. For all lengths n for which the string 1^n belongs to L, use the circuit from Example 1. Otherwise, use a trivial circuit that outputs zero.
- Consider the language of Turing Machines that halt on the empty tape. It is easy to see that this is as hard as the halting problem (why?), and therefore undecidable. On the other hand, recall that every Turing machine may be encoded as a distinct integer, and therefore we have the following undecidable unary language:

$$\mathbf{UHALT} := \{1^n \mid \text{The Turing machine corresponding to } n \text{ halts on the empty string.}\}$$

Therefore, even $\mathsf{P}_{/\mathbf{poly}}$ continues to be a somewhat awkward class, admitting even undecidable languages. However, notice that some of the awkwardness is possible because the definition of the class only demands the *existence* of a polynomial circuit family, and does not insist on these circuits being efficiently computable. It turns out that the subclass of $\mathsf{P}_{/\mathbf{poly}}$ that is constructible tallies with the class P. But before we state this formally, let us show the containment of P in $\mathsf{P}_{/\mathbf{poly}}$ first.

Theorem 1. $\mathsf{P} \subseteq \mathsf{P}_{/\mathbf{poly}}$.

Proof. (Sketch) Let L be a language in P and let M be a deterministic polynomial time TM that decides L in at most $t(n)$ steps. Define a tableau for M on w to be a $t(n) \times t(n)$ table whose rows are configurations of M. The top row of the tableau contains the start configuration of M on w. The i^{th} row contains the configuration at the i^{th} step of the computation.

For every cell of the computation tableau, we introduce several gates. For example, for all $1 \leqslant i, j \leqslant t(n)$, we introduce the following gates:

- $h_{i,j}[q, a]$, for every state q of M, and for every symbol a in the tape alphabet
- $g_{i,j}[q, a]$, for every state q of M, and for every symbol a in the tape alphabet

The semantics of the gates are as follows: we would like the value of the gate $h_{i,j}[q, a]$ to be one if, and only if, state of the computation in the i^{th} step is q, the j^{th} tape symbol is a, and the head is at location j. Similarly, we would like the value of the gate $g_{i,j}[q, a]$ to be one if, and only if, state of the computation in the i^{th} step is q, the j^{th} tape symbol is a, and the head is *not* at location j.

Note that we introduce polynomially many gates for each cell of the tableau. Further, wiring these gates to respect the semantics is also easily done; because the value of $h_{i,j}[q, a]$, for instance, only depends on the values of

$$h_{i-1,j-1}[], h_{i-1,j}[] \text{ and } h_{i-1,j+1}[]$$

and

$$g_{i-1,j-1}[], g_{i-1,j}[] \text{ and } g_{i-1,j+1}[].$$

We refer the reader to [2] (Theorem 9.30) for a more detailed exposition of this construction. □

Let us now return to the theme of small-sized circuits that can be efficiently constructed. We have the following class:

Definition 4 (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*.*

We also see that this restriction brings us to the class P.

Theorem 2. *A language* L *is computable by a* P*-uniform circuit family if, and only if,* $L \in P$*.*

Proof. (Sketch) In the forward direction, we can construct the circuit for L and simulate it in polynomial time. The reverse direction is obtained by observing that a careful analysis of the proof of Theorem 1 gives us not only a polynomial-sized circuit, but also a polynomial-time procedure for constructing one. □

2 Turing Machines and Circuits

With the examples we have seen so far, clearly Turing machines appear to be less equipped than their circuit counterparts. Is there a natural way of "empowering" the standard Turing machine model so as to bring them "at par"? Notice that the additional computational advantage in circuits seems to come from the freedom of non-uniformity — the ability to have a different internal mechanism for every length of input. Let us, therefore, equip Turing machines with additional abilities that are proportional to the length of the input.

Imagine, for example, if a Turing machine was provided with one magical bit of information to help its decision-making on strings of length n. This "advice" is enough, for instance, to assist a Turing machine in deciding any unary language — the bit would act as an indicator for whether 1^n is in the language or not. Already, we have that there exists advice that can help Turing machines recognize undecidable languages!

On the other extreme, suppose we were allowed to use advice strings of exponential length — that is, suppose we were allowed to equip our Turing machines with "hints" of length 2^n for inputs of length n. Again, this is powerful enough for us to recognize any binary language: the i^{th} bit of the advice string could act as an indicator for whether the string that is the binary representation of i is in the language or not. Intuitively, this is analogous to circuits of exponential size.

This leads us to the following definition of Turing machines *with advice*.

Definition 5. *Let* $T, a : \mathbb{N} \to \mathbb{N}$ *be functions. The class of languages decidable by time-*$T(n)$ *TMs 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 steps.*

We now formally establish the connection between circuits and Turing machines that take advice. In particular, we claim that the latter class corresponds to $\mathsf{P}_{/\mathbf{poly}}$ when the advice is restricted to be polynomially long in the input.

Theorem 3 (Polynomial-time TM's with advice decide $\mathsf{P}_{/\mathbf{poly}}$). $\mathsf{P}_{/\mathbf{poly}} = \cup_{c,d} \mathbf{DTIME}(n^c)/n^d$.

Proof. (Sketch) For a language in $\mathsf{P}_{/\mathbf{poly}}$, one could use an encoding of the polynomial-sized circuit as advice. The TM can then work by simulating the circuit in the advice string on its input to decide the outcome. In the other direction, we can mimic the proof of Theorem 1 to construct a circuit D_n such that on every input $x \in \{0,1\}^n$ and $\alpha \in \{0,1\}^{a(n)}$, $D_n(x,\alpha) = M(x,\alpha)$. We now let the circuit C_n be the one obtained from D_n by fixing the inputs α to correspond to the actual advice string. It is easily checked that this "hard wiring" can be done by adding only polynomially many additional gates. □

3 The Karp-Lipton-Sipser Theorem

The next question that arises is whether NP is contained in $\mathsf{P}_{/\mathbf{poly}}$. Both possible answers to this question lead to interesting consequences. Since $\mathsf{P} \subseteq \mathsf{P}_{/\mathbf{poly}}$, demonstrating that there is a language in NP outside of $\mathsf{P}_{/\mathbf{poly}}$ will separate P and NP. On the other hand, if NP is contained in $\mathsf{P}_{/\mathbf{poly}}$, then we will show in this section that the polynomial hierarchy collapses to the second level.

As a warm-up observation, note that if NP is contained in the class of P-uniform circuit families, then P = NP. Indeed, we can take a language in NP and construct the corresponding circuit and simulate it, all in polynomial time. Mimicking this approach for $\mathsf{P}_{/\mathbf{poly}}$ at large, however, is tricky because the construction of the circuit is elusive. However, we can achieve a collapse of PH to the second level by using the existential quantifier to "guess" the circuit. This brings us to the Karp-Lipton-Sipser theorem.

Theorem 4. *If* $NP \subseteq \mathsf{P}_{/\mathbf{poly}}$, *then* $\Pi_2 \subseteq \Sigma_2$.

Proof. (Sketch) Note that it is enough to show that Σ_2^{P} contains the Π_2^{P}-complete language Π_2-SAT which consists of all true formulas of the form:

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

Observe that the second part of the formula is a NP-predicate:

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

Since we assume that NP is contained in $\mathsf{P}_{/\mathbf{poly}}$, we know that there exists a polynomial sized circuit family $\{C_n\}_{n\in\mathbb{N}}$ such that for every Boolean formula ϕ and $u \in \{0,1\}^n$, the circuit $C_n(\phi,u)$ evaluates to one if, and only if, there exists a certificate $v \in \{0,1\}^n$ such that $\phi(u,v) = 1$.

It can be shown that we can do more: we can in fact obtain a circuit family $\{D_n\}_{n\in\mathbb{N}}$ such that for every Boolean formula ϕ and $u \in \{0,1\}^n$, the circuit $D_n(\phi,u)$ *outputs* v if there exists a certificate $v \in \{0,1\}^n$ such that $\phi(u,v) = 1$, and outputs 0 otherwise.[1]

So instead of using the NP-predicate above, we can *guess* the circuit D_n (recall that we are only assured of its existence), and feed the output of the circuit as the certificate in the NP-predicate (instead of v). We know that the circuit family D_n is polynomially bounded; and if the size of the circuit D_n is $q(n)$ for some polynomial q then it has a representation that uses at most $10q(n)^2$ bits. This gives us the following equivalent formula:

[1] We leave this as an exercise; also the reader is welcome to refer to these lecture notes for an illustration.

$$\exists w \in \{0,1\}^{c\,q(n)^2} \forall u \in \{0,1\}^n : w \text{ describes a circuit } D \text{ and } \phi(u, D(\phi, u)) = 1.$$

The truth of this predicate can be verified in Σ_2^p, since we can simulate the circuit D in deterministic polynomial time. □

4 Circuit Lower Bounds

Are there functions that cannot be computed by small circuits? Let's get some simple insights by counting:

- The total number of boolean functions on n inputs is 2^{2^n}.
- Since any boolean circuit of size t can be described by a $(t \times t)$ adjacency matrix, the total number of boolean circuits of size t is $2^{ct \log t}$, where c is an appropriate constant.

Since every circuit computes a fixed boolean function, when:

$$2^{ct \log t} < 2^{2^n}$$

we "fall short" of circuits of size t — there then exists a boolean function that cannot be computed by a circuit of size t. The above is true for, say, $t = 2^n/n(c+1)$. This brings us to the following simple lower bound, by using an upper bound for the constant c.

Theorem 5. *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 $2^n/(10n)$.*

5 The Classes NC and AC

The *depth* of a circuit is the length of the longest directed path from an input node to the output node. A circuit of size s and depth d for some problem can be easily turned into a parallel algorithm for the problem using s processors and running in "wall clock" time d. Thus, it is interesting to understand when low-depth circuits for problems exist. From a different point of view, we might expect that lower bounds would be easier to prove for low-depth circuits. These considerations motivate the following definitions.

Definition 6 (The class **NC**, Nick's Class)**.** *For every d, a language L is in $\mathbf{NC}^d$ if L can be decided by a family of circuits $\{C_n\}$ where C_n has poly(n) size and depth $O(\log^d n)$. The class* **NC** *is $\cup_{i \geqslant i} NC^i$.*

By requiring the circuits to be logspace-uniform, we may analogously define the class *uniform* **NC**. A related class is the following.

Definition 7 (The class **AC**, Alternating Circuits)**.** *The class $\mathbf{AC}^i$ is defined similarly to $\mathbf{NC}^i$ except the gates are allowed to have unbounded fan-in, that is, the OR and AND gates can be applied to more than two bits. The class* **AC** *is $\cup_{i \geqslant i} AC^i$.*

Since unbounded (but polynomial) fan-in can be simulated by a binary tree of depth $O(\log n)$, we have the following:

$$\mathbf{NC}^i \subseteq \mathbf{AC}^i \subseteq \mathbf{NC}^{i+1}.$$

The inclusion is known to be strict for $i = 0$. It is not hard to see that Parity is in $\mathbf{NC}^1$, but it can also be shown that Parity is not in $\mathbf{AC}^0$.

Note that $\mathbf{NC}^0$ is extremely limited since the output depends only on a constant number of input bits, while $\mathbf{AC}^0$ does not suffer from this limitation.

5.1 P-Completeness

The question of whether P = **NC** is a central open problem. This is essentially the question of whether every polynomial time algorithm admits an efficient parallel implementation. This motivates the theory of P-completeness, which can be used to study which problems are likely to admit efficient parallel algorithms. A natural definition of a complete language involves suggesting that every language in the class reduces to it. The detail lies in determining how we constrain these reductions. Clearly, polynomial time reductions are not very meaningful in the context of P-completeness. A moment's reflection reveals that log-space reductions capture the properties that we anticipate as desirable from a P-complete language.

Definition 8. *A language is P-complete if it is in P and every language in P is log-space reducible to it.*

Indeed, with the definition above, one can establish the following, which is left as an exercise.

Theorem 6. *Let* L *be a P-complete language. Then,*

- $L \in \mathbf{NC}$ *if and only if* $P = \mathbf{NC}$.
- $L \in \mathbf{L}$ *if and only if* $P = \mathbf{L}$, *where* **L** *is the set of languages decidable in logarithmic space.*

References

[1] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009. 1

[2] M. SIPSER. *Introduction to the Theory of Computation*. PWS, Boston, MA, 1996. 4