Boolean Circuits!

Torin Kovach

April 27, 2021

Contents

Acknowledgments

Thank you to Professor Ada, my mentor TA Sayan, and the entire course staff! Also thank you to my group: Jeremy and Andrew. This has been quite fun! ☕

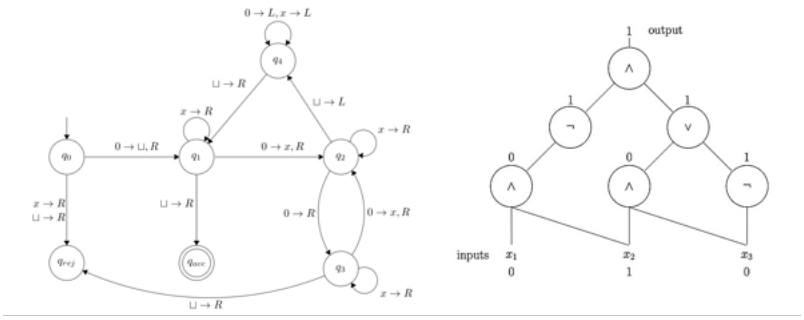
1 Introduction & Motivations

1.1 Why Boolean Circuits?

In 15-251, we've covered multiple different models of computation, such as Deterministic Finite Automata (DFA) and Turing Machines (TM). Boolean circuits can be thought of as yet another model of computation. However, this raises the question – why do we need Boolean circuits? When we began discussing TM's, it was clearly relevant that they allowed us a more powerful computational model than DFA's. However, we know by the Church Turing Thesis that any computation in the universe can be carried out by a TM – what can Boolean circuits do that TM's can't?

We have a few notable reasons. Firstly, Boolean circuits are exceedingly simple to work with compared to our current computational models. Instead of a 5 or 7-tuple, we have a simple directed graph that one can easily look at and reason about. Given a simple problem, it is often far easier to write out a diagram of a Boolean circuit solving it compared to a Turing Machine.

Figure 1: Would you rather reason about a complicated TM (left) or a nice Boolean circuit (right)?



Turing Machines can be thought of as similar to a human writing on a piece of paper – the states correspond to different states of mind, the tape and cells pieces of paper and spaces on the paper, etc. Similarly, Boolean circuits can be thought of as similar to the digital circuits present in all of our computers today. As a result, from a computer or electrical engineering perspective, Boolean circuits could seem "closer" to the method of computation implemented in real devices.

Perhaps most importantly, Boolean circuits can efficiently simulate Turing Machines – that is, any language efficiently decidable by a Turing Machine can efficiently be decided by a family of Boolean circuits (more on this later!). As a result, the contrapositive is true – if a language cannot be efficiently computed by a Boolean circuit, it cannot be efficiently decidable by a Turing machine! This allows us to use Boolean circuits to investigate P vs NP, NP-hardness and completeness, etc. Closer to the end of this document, we will investigate this and find some really cool results!

1.2 A Pesky Problem

While Boolean circuits seem really cool, there still are a few caveats that limit their abilities – otherwise, why would we need Turing Machines? Given a single Boolean circuit, it can only take input of a single length. By contrast, both Turing Machines and DFA's were able to take input of an arbitrary length! Therefore, to decide a language, we can't just use a single Boolean circuit – instead, we require a family of Boolean circuits, each used for input of a specific length. This limitation has both beneficial and problematic consequences, which we will get into later!

2 Definitions, Results, & Proofs

2.1 The Basics

Let's get right into it with the definition of a Boolean circuit!

Definition 2.1. A **Boolean circuit** of n input variables is a directed, acyclic graph, where vertices are referred to as **gates** and edges as **wires**, satisfying the following:

- Gates are either input gates, output gates, AND gates, OR gates, or NOT gates.
- AND and OR gates have in-degree of 2, NOT gates have in-degree of 1, output gates have an in-degree of 1, and input gates have in-degree of 0.
- There are exactly n input gates, labelled $x_1, x_2, \ldots, x_n$.
- There is a single output gate, with out-degree of 0.

Exercise 2.1. Both Turing Machines and Discrete Finite Automata are represented as tuples. How might be represent Boolean Circuits as a tuple?

One (perhaps overly complicated) example would be $B = (n, I, G, x_{\text{output}}, W)$. In this tuple, the gates or vertices of our graph are $I \cup G \cup \{x_{\text{output}}\}$, and the wires or edges are stored as tuples in W. The number $n \in \mathbb{N}$ gives us the number of input gates, and tuple $I = (x_1, x_2, \ldots x_n)$ stores them. G stores the AND, NOT, and OR gates, while x_{output} is the output gate.

Now, we can begin to examine what it means for a Boolean circuit to compute a function. Suppose we are given a string $\ell \in \{0,1\}^n$ and a Boolean circuit B of n input variables for some $n \in \mathbb{N}^+$. Now, B processes input x and produces a result, which we will refer to simply as $B(x)$, by first assigning the i-th bit of x to x_i in B for all $i \in [n]$, where $x_1, x_2, \ldots, x_n$ are the input gates of B. Then, each gate takes in input from wires going into the gate and send output to all the wires going from the gate.

The input gates simply take in their assigned input and send this input unchanged as their output. The AND gates, with in-degree of 2, take input of 2 bits and output 1 if and only if both input bits are 1 (otw. output is 0). The OR gates similarly takes input of 2 bits and output 0 if and only if both input bits are 0 (otw. output is 1). The NOT gates takes a single input bit and returns 0 if the input is 1, and 1 if the input is 0. The output gate simply takes the input and does nothing. Note that as our graph is directed and acyclic, we can naturally process each gate incrementally to determine inputs and output until we reach the output gate. The input given to the output gate (either 0 or 1), is the final result of $B(x)$. Now, we can define what it means for a Boolean circuit to compute a function!

Definition 2.2. Given a function $f\colon \{0,1\}^n \to \{0,1\}$ for some $n \in \mathbb{N}^+$, a Boolean circuit B of n input variables **computes** f if for all $x \in \{0,1\}^n$, $f(x) = B(x)$

Exercise 2.2. Suppose we want to have constant gates, that is a gate CONST1 that always returns 1 and a gate CONST0 that always returns 0. First, we can think about these as functions CONST1$\colon \{0,1\} \to \{0,1\}$ and CONST0$\colon \{0,1\} \to \{0,1\}$. How might we design Boolean circuits to compute these functions?

For CONST1, we use a single NOT gate and a single OR gate, so given bit x we compute x OR (NOT x). This must always be true, so our circuit always returns 1. We construct CONST0 in the same manner but replace our OR with an AND.

Remark. One way that we can think about Boolean circuits are as a generalization of propositional formulas, where the variables of the propositional formula correspond to the input variables of the circuit. Given any propositional formula, we can easily encode it into a Boolean circuit where each $\vee$, $\wedge$, and $\neg$ corresponds to a gate in the Boolean circuit (the exact mechanics of this are left as an exercise).

Exercise 2.3. Given 2 bits of input, the XOR$\colon \{0,1\}^2 \to \{0,1\}$ function returns 1 if a single input bit is 1, and 0 if either none or both of the input bits are 1. The ODD$\colon \{0,1\}^4$ returns 1 if and only if the input string has an odd number of 1's. Write Boolean circuits computing XOR and ODD.

Note that given propositional variables a and b, we can compute a XOR b as $(\neg a \wedge b) \vee (a \wedge \neg b)$. As a result, using two AND gates, two NOT gates, and a single OR gate we can convert this propositional formula into a Boolean circuit deciding XOR. Now, note that given 2 bits, XOR decides if there is an odd number of ones in these bits. Furthermore, given 4 bits, we can split these bits into sets of two bits $S_1 = (x_1, x_2)$ and $S_2 = (x_3, x_4$. Thus, there is an odd number of bits if and only if one of S_1 contains an even number of bits, and the other contains an odd number of bits. Thus, $\text{ODD}(x_1, x_2, x_3, x_4)$ is equivalent to $\text{XOR}(\text{XOR}(x_1, x_2), \text{XOR}(x_3, x_4)$. Using this fact and our circuit computing XOR, we can design a circuit computing ODD.

2.2 Family Matters

Now that we are armed with the idea of a Boolean circuit computing a function, we can generalize this idea to tackle functions that take in lots of different input lengths! This requires one small notational difference from what we've seen in the past:

Remark. For alphabet Σ, we know Σ^n defines all languages of length n. Suppose we have decision problem $f\colon \Sigma^* \to \Sigma$. We can segment this problem into a sequence of functions $(f^0, f^1, f^2, \dots)$. For an arbitrary $n \in \mathbb{N}$, we define $f^n\colon \Sigma^n \to \Sigma$ such that $\forall x \in \Sigma^n, f^n(x) = f(x)$. As $\bigcup_{i\in\mathbb{N}} \Sigma^i = \Sigma^*$, using our sequence of functions we can compute f for all possible input $x \in \Sigma^*$.

Definition 2.3. A **circuit family** is a sequence of Boolean circuits $(B_1, B_2, B_3, \dots)$ such that for all $n \in \mathbb{N}$, B_n is a Boolean circuit of n input gates. Given a decision problem $f\colon \{0,1\}^* \to \{0,1\}$, we say that a circuit family $(B_1, B_2, B_3, \dots)$ **decides** or **computes** f if for all $n \in \mathbb{N}$, f^n is computed by B_n.

Remark. By our definitions above, B_0 is not very well defined, as we have no idea what it would look like to have 0 input gates. Note that as $\{0,1\}^0 = \{\varepsilon\}$, the only possible input to B_0 is the empty string. Thus, we can think of B_0 as a circuit simply always returning 1 or always returning 0, so that for function $f^0\colon \{0,1\}^0 \to \{0,1\}$ we always satisfy $B_0(\varepsilon) = f^0(\varepsilon)$.

Exercise 2.4. Suppose we have a new function ODDER$\colon \{0,1\}^* \to \{0,1\}$ which given input string $x \in \{0,1\}^*$ returns 1 if and only if x contains an odd number of ones. How can we construct a family of circuits that decides ODDER?

Note that we can trivially construct circuits for ODDER^0 and ODDER^1. Now, we note that given a string x which we can divide into substrings x_1 and x_2, x contains an odd number of ones if and only if one of x_1 or x_2 contains an odd number of ones and the other contains an even number of ones. Equivalently, $f(x) = f(x_1)$ XOR $f(x_2)$. Exploiting this, we find that for arbitrary $k \in \mathbb{N}^+$, for all $x \in \{0,1\}^{k+1}$, we can split x into a single character c and x' so $x = x'c$, and as a result $B_{k+1}(x) = \text{XOR}(B_k(x'), B_1(c))$. As we know how to implement XOR in a boolean circuit from exercise 2.3, we can proceed inductively to produce our entire circuit family!

2.3 Getting Complex

As our circuits are represented by directed, acyclic graphs, when evaluating any one circuit on a given input, we never "loop" – that is, we never have to evaluate input for a gate more than once. As a result, we can measure how long a circuit takes to run by measuring how many gates are in it! This allows us to come up with an idea of complexity for Boolean circuits.

Definition 2.4. The **size** of a circuit is its number of gates. The **size complexity** of a circuit family $(B_0, B_1, B_2, \dots)$ is a function $S\colon \mathbb{N} \to \mathbb{N}$ such that for all $i \in \mathbb{N}$, $S(i)$ is equal to the size of B_i. A circuit B is **size minimal** if no smaller circuit exists computing the same function as B. The **circuit complexity** of a decision problem $f\colon \{0,1\}^* \to \{0,1\}$ is the size complexity of a circuit family computing f such that every circuit in the family is size minimal.

Exercise 2.5. Note that in our definition of size complexity, we include the input gates as part of the count total size. Some others do not include the input gates in this measurement. Use the fact that we include the input gates to prove that every decision problem has a circuit complexity in $\Omega(n)$.

We see that every circuit of n input variables has at least n total gates – the input gates themselves. As a result, we can conclude that given the size complexity S of an arbitrary family of circuits, it must be true that for all $n \in \mathbb{N}$, $n \leq S(n)$. As a result, given an arbitrary decision problem, its circuit complexity C must satisfy $n \leq C(n)$ for all $n \in \mathbb{N}$. Therefore, the circuit complexity of any decision problem is in $\Omega(n)$.

Exercise 2.6. Prove that the circuit complexity of ODDER is in $\Theta(n)$.

Our lower bound is supplied by Exercise 2.5. Furthermore, from Exercise 2.4 we can conclude that for all $n \in \mathbb{N}^+$, ODDER^n only requires $n-1$ uses of XOR. As we can represent a single XOR with 5 gates (as shown in Exercise 2.3), we can upper bound the circuit complexity of ODDER by $5n \in O(n)$.

Perhaps even more powerful than a lower bound for the circuit complexity on an arbitrary decision problem is an upper bound! Exploiting the need for a new Boolean circuit for each possible input length, we can find an exponential upper bound on the circuit complexity for any decision problem.

Theorem 2.5. Any decision problem has a circuit complexity in $O(2^n)$.

We can achieve this bound by "hardcoding" each possible string into our circuits. Let $C\colon \mathbb{N} \to \mathbb{N}$ be the circuit complexity of the largest minimal circuit of n input variables – that is, $C(n)$ equals the size of largest possible minimal

circuit computing any function $f\colon \{0,1\}^n \to \{0,1\}$. First, for $n = 1$ we can decide any possible function with at most 4 gates (see Exercise 2.2). Now, consider some specific $n \in \mathbb{N}^+$. It follows there exists some function $f^n\colon \{0,1\}^n \to \{0,1\}$ which takes $C(n)$ total gates for a minimal boolean circuit to compute. Define functions $g_0, g_1\colon \{0,1\}^{n-1} \to \{0,1\}$, such that for all $x \in \{0,1\}^{n-1}$, $g_0(x) = f^n(x \cdot 0)$, and $g_1(x) = f^n(x \cdot 1)$ (here "$\cdot$" implies concatenation). It necessarily follows that for any $x = x_1x_2\dots x_{n-1}x_n \in \{0,1\}^n$, $f(x) = (\neg x_n \wedge g_0(x_1x_2\dots x_{n-1})) \vee (x_n \wedge g_1(x_1x_2\dots x_{n-1}))$. Therefore, given Boolean circuits B_{g_0} and B_{g_1} computing g_0 and g_1, respectively, we can compute f^n with a Boolean circuit B_n implementing B_{g_0}, B_{g_1}, and 6 extra gates (an extra input gate, one not, two ands, one or, and an output gate. As a result, B_n will use at most $2 \cdot C(n-1)+6$ gates, so $C(n) \leq 2 \cdot C(n-1)+6$. This recurrence relation implies our exponential bound, so $S(n) \in O(2^n)$. As a result, any possible decision problem must have a circuit complexity of at worst $O(2^n)$.

This result might seem a little crazy! With Turing Machines, we can construct languages that can't even be decided! However, using circuit families, even the hardest of languages take at most $O(2^n)$ circuit complexity. Perhaps just as important as showing how hard languages can be is showing that there exist languages that are this hard. This is characterized by Shannon's Theorem!

Theorem 2.6. Most languages $L \subseteq \{0,1\}^*$ have minimal circuits computing L of size at least ${}^{2^n}\!/_n$.

Let the function $\texttt{Count}(s,n)$ be a function specifying the number of Boolean circuits with n total input gates and s other gates (so $s+n$ is the total size). We can specify a circuit by specifying the types and gates directed to each of the s non-input gates. Each input gate has 4 possible types (AND, OR, NOT, output), and at most 2 gates directed to it, out of $(s+n-1)$ options. Also note that the order of the gates don't matter. Therefore, we can conclude that:

$$\begin{aligned}
\texttt{Count}(s,n) &\leq \frac{(4(s+n-1))^s}{s!} \\
&\leq \frac{(4s^2(1+\frac{n-1}{s})^2)^s}{{}^{s^s}\!/_{2^s}} && \text{(as } s! \geq ({}^s\!/_2)^s \text{ for } s \geq 0\text{)} \\
&= (8)^s s^s \left(1+\frac{n-1}{s}\right)^{2s} \\
&\leq (32)^s s^s && \text{(as } s \geq n-1\text{)}
\end{aligned}$$

Now, if we have $s = {}^{2^n}\!/_n - n$, so the total circuit size is ${}^{2^n}\!/_n$. Then we have:

$$\begin{aligned}
\texttt{Count}({}^{2^n}\!/_n - n, n) &\leq \texttt{Count}({}^{2^n}\!/_n, n) \\
&= (32)^{({}^{2^n}\!/_n)}({}^{2^n}\!/_n)^{({}^{2^n}\!/_n)} \\
&= \left(\frac{32}{n}\right)^{({}^{2^n}\!/_n)} 2^{(2^n)} \\
&\ll 2^{(2^n)} && \text{(for large enough } n\text{)}
\end{aligned}$$

Note there are $2^{(2^n)}$ total functions $f\colon \{0,1\}^n \to \{0,1\}$, as each function f has 2^n possible inputs and each input has two possible outputs. Therefore, for large enough n the number of circuits of size ${}^{2^n}\!/_n$ is far smaller than the number of functions $f\colon \{0,1\}^n \to \{0,1\}$, so most languages have minimum circuit families of size, and thus circuit complexity of, at least ${}^{2^n}\!/_n$.

Note that Theorem 2.5 implies that the decision problem for **any** language can be computed by a circuit family. This highlights the distinct difference between Turing Machine complexity/decidability and Boolean circuit complexity/decidability. However, there is one extremely important relation between the two:

Theorem 2.7. Let M be a Turing Machine which can be decided in $O(T(n))$ for some $T\colon \mathbb{N} \to \mathbb{N}$. Then the Turing Machine can be simulated by a family of circuits with circuit complexity in $O(T^2(n))$ – that is, the function deciding $L(M)$ has a circuit complexity of $O(T^2(n))$.

While we will not provide the entire proof here, we will provide some motivation. Suppose we have M which can be decided in $O(T(n))$ time, that is at most $c \cdot T(n)$ for some $c \geq 0$. It follows that for some input $x \in \{0,1\}^n$, $M(x)$ moves at most $c \cdot T(n)$ tape cells left and $c \cdot T(n)$ tape cells right. Furthermore, $M(x)$ moves at most $c \cdot T(n)$ steps total. Thus, for each $n \in \mathbb{N}$, we construct a circuit B_n of $T(n)$ rows of gates (one for each step of the computation. Each row is composed of $2 \cdot T(n)$ gates, one for each possible cell M may visit. Each row is wired to the previous row so that it can calculate its configuration using the output of the previous row's configuration. A gate in the final row is designated as the output gate.

Remark. An important note about the process explained in Theorem 2.7 to "convert" a Turing Machine to a circuit family; comuting a single circuit B_n in the family can be done in polynomial time with regard to n.

2.4 Circuits Solve Entscheidungsproblem

Remark. Note that this relationship shown by Theorem 2.7 is not bi-directional. If a circuit family is able to compute a function deciding a language, there is no guarantee that a Turing Machine exists which can decide the language, irregardless of complexity. This is highlighted by the exercise below.

Exercise 2.7. Given a circuit family $F = (C_0, C_1, C_2, \dots)$, we define BUILD_F as the problem taking in an encoding of a natural number n, and returning the encoding of C_n. Show that there exists some circuit family F' such that BUILD'_F is undecidable by Turing Machines.

Consider the language HALTS, which takes in the encoding of a Turing machine M and some string x, and determines whether $M(x)$ halts. We know HALTS to be undecidable by Turing Machines. Furthermore, we know that there must exist some family of circuits F which compute the functions deciding HALTS. If BUILD_F were decidable by a Turing Machine, then we could construct a Turing Machine to decide HALTS by running BUILD_F and simulating the resulting Boolean circuit – that is, HALTS reduces to BUILD_F. As we know HALTS is undecidable by a Turing Machine, it must then be true that BUILD_F is undecidable.

The exercise above shines a light on a big problem with using Boolean circuits as our only computational model – some circuit families would be provably impossible to compute in real life! The "infinite-ness" of Circuit families allows us to decide languages which are undecidable by Turing Machines. We proved there must be languages undecidable by Turing Machines because the set of all languages is uncountable, while the set of all Turing Machines is only countable. In the following theorem, we will attempt to help quantify why Circuit families can decide Turing Machine undecidable languages.

Theorem 2.8. The set of all Boolean circuit families is uncountable.

Fix alphabet $\Sigma = \{0, 1\}$. Let $\mathcal{C}$ be the set of Boolean circuit families, and let $\mathcal{L} = \mathcal{P}(\Sigma^*)$, the set of all languages. By Cantor's theorem, $\mathcal{L}$ is uncountable. We define $f : \mathcal{C} \to \mathcal{L}$ as for all families $B \in \mathcal{C}$, $f(B)$ is equal to the language that B computes. As all languages are computable by a Boolean circuit family (see Theorem 2.5), we can conclude that f is a surjection, and thus $|\mathcal{C}| \geq |\mathcal{L}|$. Therefore, $\mathcal{C}$ is uncountable.

Exercise 2.8. Prove that the set of all Boolean circuit families is uncountable by constructing a surjection with $\{0, 1\}^\infty$. This is left as an exercise to the reader.

While the set of all Boolean circuit families is uncountable, a whole lot of these circuits will be of no practical use to us! For instance, the circuit family detailed in Exercise 2.7 would be beyond difficult to physically implement. However, what if we only took a subset of these families that we could actually use? That way, we can work with Boolean circuits without having to worry about whether these circuits can actually be computed! The definition below does just that:

Definition 2.9. A family of Boolean circuits $B = (B_0, B_1, B_2, \dots)$ is **polynomial-time uniform** if there exists a Turing Machine M_B such that M_B runs in polynomial time on all inputs, and given the encoding of a natural number n, $M_B(n)$ returns an encoding of B_n.

Exercise 2.9. Prove that $\mathcal{PU}$, the set of all polynomial-time uniform circuit families, is countably infinite.

Note that for every $F \in \mathcal{PU}$, there exists a Turing machine M_B which computes members of F. Thus, take the subset of the set of all Turing Machines containing Turing Machines which compute valid Boolean circuits, and denote this set as $\mathcal{TM}$. As the set of all Turing Machines is countable, $\mathcal{TM}$ is countable. We define a function $f : \mathcal{TM} \to \mathcal{PU}$ as follows; given a $M \in \mathcal{TM}$, $f(M)$ returns the Boolean circuit family computed as $(M(\langle 0 \rangle), M(\langle 1 \rangle), M(\langle 2 \rangle), \dots)$. As every $B \in \mathcal{PU}$ has a Turing Machine computing it, f is surjective, so $\mathcal{BC}$ must be countable. As $\mathcal{PU}$ is trivially infinite, $\mathcal{PU}$ is countably infinite!

Recall that the set of all languages, $\mathcal{P}(\Sigma^*)$, is uncountable. Therefore, using the result of Exercise 2.9 we can prove $|\mathcal{PU}| < |\mathcal{P}(\Sigma^*)|$. As a result, in similar fashion to how we proved the existence of undecidable languages by Turing Machines, there must exist languages who's decision problems cannot be computed by any circuit family in $\mathcal{PU}$.

2.5 Tackling the Big Questions

By exploiting Theorem 2.7, we can prove some very powerful things! Not only is this theorem useful in allowing us to reason with Boolean circuits as opposed to Turing Machines, but the contrapositive implies that if there is not a polynomial-time Boolean circuit family computing a decision problem, then there cannot be a polynomial-time Turing Machine computing a decision problem. While we will not use this property here, this is a potential method by which researchers can attack the P vs. NP problem (if an NP-complete problem cannot be decided with polynomial circuit complexity, then it must be true that $\mathsf{P} \neq \mathsf{NP}$)! Perhaps my very favorite exploitation of Theorem 2.7 is a fun alternative proof to the Cook-Levin Theorem!

Theorem 2.10. (Cook-Levin Theorem) The problem CIRCUIT-SAT is NP-complete.

Fix alphabet $\Sigma = \{0, 1\}$. First, note that CIRCUIT-SAT is in NP – building a verifier is relatively easy using the assignment to input gates which satisfies the circuit as a certificate. Now, we prove CIRCUIT-SAT is NP-hard. Take an arbitrary language $L \in \mathsf{NP}$. It follows that L has a polynomial-time verifier V_L. By Theorem 2.7, there exists a circuit family $F = (B_0, B_1, B_2, \dots)$ simulating V_L. We construct a function $f\colon \Sigma^* \to \Sigma^*$ as follows.

Given arbitrary string x, we first calculate $|x| = n$. Now, note there is a polynomial number in n of possible certificate lengths. For each of these possible input lengths, we compute the corresponding circuit from F (note computing each circuit is done in polynomial time, and we compute a polynomial number of them). These produce a set of circuits G, for each of which the first n input gates correspond to the problem input, and the rest the certificate input. We first insert gates for CONST1 and CONST0 directly after these input gates so the problem input is always x (see Exercise 2.2; note this only requires an addition of $2n$ gates per circuit, taking polynomial time). Finally, we "merge" these circuits together into a single circuit C of k input variables, where k is the maximum certificate length plus n, for all $0 \leq i \leq k$ the circuit checking for certificates of length i is given the first $n + i$ input gates as input, and the merged circuit outputs 1 if and only if at least one of the "sub-circuits" outputs 1 (this can be done with a polynomial number of OR gates. Our polynomial-time computable function returns C.

Now, if $f(x)$ is accepted by CIRCUIT-SAT, then it follows that there must exist a certificate $u \in \Sigma^*$ for x such that $V_k(x, u)$ accepts, and thus $x \in L$. Similarly, if $x \in L$, then a certificate $u \in \Sigma^*$ must exist so that $V_k(x, u)$ accepts, and thus the circuit $f(x)$ will be satisfied by u, so $f(x) \in$ CIRCUIT-SAT. Therefore, for any arbitrary $L \in \mathsf{NP}$, $L \leq_m^P$ CIRCUIT-SAT, so we can conclude that CIRCUIT-SAT is NP-complete.

Remark. While we proved that CIRCUIT-SAT is NP-complete, Cook and Levin originally proved that SAT was NP-complete. However, we still complete the "purpose" of this theorem – showing that a language is NP-complete without any other known NP-complete language to work from!

The idea of polynomial-time uniformity is also and important one when reasoning about the P vs. NP problem. This can be seen as a direct result of the theorem below!

Theorem 2.11. Language $L \subseteq \{0, 1\}^*$ is computable by a polynomial-time uniform circuit family if and only if $L \in \mathsf{P}$.

Here, we are proving that $\mathcal{PL}$, the set of all languages computable by a polynomial-time uniform circuit family, is equal to P. First, take arbitrary $L \in \mathsf{P}$. It follows that L can be decided by a Turing Machine V in polynomial time. By Theorem 2.7, we have an algorithm to convert V into circuit family $(B_0, B_1, B_2, \dots)$, computing each circuit in polynomial time. Thus, we design Turing Machine M_V which given the encoding of a natural number n, returns circuit B_n. Therefore, $L \in \mathsf{P}$, so $\mathsf{P} \subseteq \mathcal{PL}$. Now, take arbitrary $K \in \mathcal{PL}$. It follows there exists a Turing Machine M_K which, given the encoding of natural number n, produces a circuit B_n computing the decision problem for K for inputs of length n. We construct a new Turing Machine D_K, which, given a string x, simulates $M_k(|x|)$ to produce circuit $B_{|x|}$, and then returns $B_{|x|}(x)$. This Turing Machine will run in polynomial time and deicide K, so $K \in \mathsf{P}$. Thus, $\mathsf{P} = \mathcal{PL}$.

We can also define an entirely new complexity class based on the idea of Boolean circuits. This class, and a specific property we will prove about this class, has been central in many attempts to prove $\mathsf{P} \neq \mathsf{NP}$.

Definition 2.12. The complexity class $\mathsf{P}_{/\mathsf{poly}}$ is the set of languages that are decidable by polynomial-size circuit families – that is, the set of all languages with decision problems of polynomial-time circuit complexity.

Theorem 2.13. The set of all polynomial-time decidable languages is a proper subset of $\mathsf{P}_{/\mathsf{poly}}$. That is, $\mathsf{P} \subset \mathsf{P}_{/\mathsf{poly}}$.

First, we show that $\mathsf{P} \subseteq \mathsf{P}_{/\mathsf{poly}}$. Take an arbitrary $L \in \mathsf{P}$. By definition of P, we know L has a polynomial-time decider Turing Machine M. By Theorem 2.7, we know that there exist a circuit family $F = (B_0, B_1, B_2, \dots)$ simulating M, and thus computing the decision problem for L. Furthermore, as M runs in worst-case $O(T(n))$ for some polynomial $T(n)$, the size complexity of the elements of the circuit family F is bounded by $O(T^2(n))$, which is also polynomial. Therefore, $L \in \mathsf{P}_{/\mathsf{poly}}$, so $\mathsf{P} \subseteq \mathsf{P}_{/\mathsf{poly}}$.

Next, we show $\mathsf{P}_{/\mathsf{poly}}$ contains an element not in P. We have 2 methods to do so – a non-constructive and a constructive one. We will begin using the former, and provide an outline of the latter. Note that given a natural number n and a value $b \in \{0, 1\}$, I can trivially construct an n-input gate Boolean circuit with a total number of gates bounded by a polynomial in n, which returns b on all possible inputs (for an idea at exactly how to do this, see exercise 2.2). Therefore, given an infinite boolean string $s \in \{0, 1\}^\infty$, I can construct a boolean circuit family $(B_0, B_1, B_2, \dots)$ of polynomial-bounded size complexity such that for all $n \in \mathbb{N}$, on all possible input B_n returns the n-th symbol of s (indexed from 0). Therefore, for any string $s \in \{0, 1\}^\infty$, the language L_s such that for all $n \in \mathbb{N}$, all strings of size n are in the language if the n-th digit of s is 1, and no strings of size n are in the language is the n-th digit of s is 0, is in $\mathsf{P}_{/\mathsf{poly}}$. Thus, we have an injective mapping of $\{0, 1\}^\infty$ to $\mathsf{P}_{/\mathsf{poly}}$, so $\mathsf{P}_{/\mathsf{poly}}$ is uncountable. As P is countable, we

can conclude that it cannot be true that $\mathsf{P} = \mathsf{P}_{/\mathsf{poly}}$, so $\mathsf{P} \subset \mathsf{P}_{/\mathsf{poly}}$.

Now, consider the unary language $\{1\}^*$. It follows as this language is countably infinite, we can encode every single Turing Machine as an element of $\{1\}^*$. By the same construction as the previous paragraph, using the fact that $\{1\}^* \subseteq \{0,1\}^*$ any language that takes the encoding of a Turing Machine as input is in $\mathsf{P}_{/\mathsf{poly}}$. Thus, EMPTY-HALTS, SELF-ACCEPTS, and other undecidable languages are all in $\mathsf{P}_{/\mathsf{poly}}$, so it must be true that $\mathsf{P} \subset \mathsf{P}_{/\mathsf{poly}}$. More generally, by this construction we show that every undecidable unary language is necessarily in $\mathsf{P}_{/\mathsf{poly}}$.

Remark. Now, suppose that we can prove $\mathsf{NP} \not\subseteq \mathsf{P}_{/\mathsf{poly}}$, that is that there exists some language $L \in \mathsf{NP}$ such that $L \notin \mathsf{P}_{/\mathsf{poly}}$. As a result, it would necessarily follow that $\mathsf{P} \neq \mathsf{NP}$. This method has been the focus of many attempts to solve the P vs. NP problem. Actually, this is simply a formalization of the method explained at the beginning of this subsection!

To complete our discussion on Boolean circuits, we will investigate one more use of Boolean circuits that I personally find fascinating. While we often discuss NP-complete problems, we don't very often think of P-complete ones. One potential example in linear programming. However, using Boolean circuits we can come up with an exceedingly more simple P-complete problem!

Definition 2.14. The **Circuit Value Problem** (**CVP**) is given a Boolean circuit and a potential input to the circuit, determine whether the circuit will output 1.

Theorem 2.15. The Circuit Value Problem is P-complete.

Given a circuit of n gates and a set of inputs, by simply simulating the circuit we can trivially decide the CVP problem in polynomial time, as each gate need by evaluated only once. Now, take an arbitrary language $L \in \mathsf{P}$, and its polynomial-time decider M. We define a function $f \colon \Sigma^* \to \Sigma^*$ as follows. Given a string $x \in \Sigma^*$ with $|x| = n$, we use the method of Theorem 2.7 to generate circuit B_n, computing the decision problem for L for inputs of length n. Our function f returns B_n and x. If CVP accepts $f(x)$, then $B_n(x) = 1$, so $x \in L$. Otherwise, $B_n(x) = 0$, so $x \notin L$. As f can be computed in polynomial time, $L \leq_m^P$ CVP, so CVP is P-complete!

3 Exercises

These exercises should hopefully Check Your Understanding© of the coolest computational model – Boolean Circuits!

3.1 Basic Definitions

1. We define Boolean circuits as directed and acyclic graphs. Why would we want these graphs to be acyclic?
2. When we run a Boolean circuit on a given input, how many times is each specific gate evaluated?
3. If I tell you a circuit computes a function $f\colon A \to B$, what can you tell me about the sets A and B?
4. If I tell you a circuit family computes a function $f\colon A \to B$, what can you tell me about the sets A and B?

3.2 Building Circuits

1. The NAND$\colon \{0,1\}^2 \to \{0,1\}$ function returns 0 if and only if both input bits are 1. Come up with two different Boolean circuits computing NAND – one using AND gates and no OR gates, and one using OR gates and no AND gates.
2. The NOR$\colon \{0,1\}^2 \to \{0,1\}$ function returns 0 if and only if both input bits are 0. Come up with two different Boolean circuits computing NAND – one using AND gates and no OR gates, and one using OR gates and no AND gates.
3. Suppose we only have access to a NAND gate – that is a gate which computes NAND. Can we create Boolean circuits computing AND, OR, and NOT?
4. Suppose we only have access to the NOR gate. Can we create Boolean circuits computing AND, OR, and NOT?
5. Use your answer to the previous questions to show that we can convert any Boolean circuit using AND, OR, and NOT gates into a boolean circuit using either only NOR or only NAND gates.

3.3 Complexity

1. True or false. All Boolean circuits have size complexity of $\Omega(n)$.
2. What is the difference between circuit complexity and size complexity?
3. Prove that every decision problem has a circuit complexity in $O(n \cdot 2^n)$, without any reference to Theorem 2.5. Hint: $L = \{01, 00\}$ corresponds to $(\neg x_1 \wedge x_2) \vee (x_1 \wedge x_2)$.
4. Another definition used for the complexity of a Boolean circuit is the total depth of the circuit – that is, the longest path from an input gate to the output gate. Why might this notion of complexity be useful? (Hint: think span!)
5. Suppose we prove that for a family of Boolean circuits to decide 3COL, we must have at least an exponential circuit complexity. What can we conclude about P, NP, and $\mathsf{P}_{\mathsf{poly}}$, if anything?
6. Suppose we prove that 3COL can be decided by family of Boolean circuits with circuit complexity $O(n^{15251})$. What can we conclude about P, NP, and $\mathsf{P}_{\mathsf{poly}}$, if anything?

3.4 Decidability

1. True or false. SELF-ACCEPTS can be decided by a family of Boolean circuits.
2. Does there exist a language that is undecidable by a family of Boolean circuits?
3. We prove that there must exist undecidable languages for Turing Machines by observing that the set of all languages is uncountable, while the set of all Turing Machines is countable. Why does this method not work for Boolean circuit families?
4. By the method explained in Theorem 2.7, can a Boolean circuit family simulate a Turing Machine that runs forever on some inputs? Why or why not?
5. We prove the set of all polynomial-time uniform circuit families is countably infinite. However, let's say we remove the "polynomial-time" constraint to define "uniform" circuit families as those which can be computed by a decider Turing Machine at all. Is the set of all uniform circuit families countably infinite? Why or why not?

4 References

The sources I used in this write-up include both Professor Ada's lecture video, Michael Sipser's *Introduction to the Theory of Computation*, 3rd Edition, the Wikipedia article for Boolean Circuits and circuit complexity, and some materials from U.C. San Diego's Circuit Complexity class. I did not copy-paste or copy verbatim any of the material presented in any of these sources (except for the images on page 2, which I got from Diderot and Wikipedia, respectively). I did however adapt the information, definitions, and proof methods presented in these sources for my own use, writing everything in my own words. I adapted the LaTeX template of this document from one I found on Overleaf.