Unofficial Companion Notes to *Introduction to the Theory of Computation* by Michael Sipser

Kevin Sun

Preface

These notes are primarily written for anyone studying from the book *Introduction to the Theory of Computation* by Michael Sipser, specifically the third edition. (In these notes, whenever I refer to "the book," I am referring to that book.) I think the book is excellent, and I strongly recommend it. When writing these notes, I also consulted other resources, most notably the following, which I also recommend:

- *Introduction to Theoretical Computer Science* by Boaz Barak
- *Models of Computation* by Jeff Erickson

These notes are not meant to be a substitution for the book; instead, I think of them as an unofficial companion to the book. I hope you find them useful.

— Kevin Sun
December 2023

Last updated: January 2024

1 Regular Languages

Much of computer science is concerned with the task of solving a problem using an efficient algorithm. In contrast, our main question throughout these notes is this: which problems *cannot* be solved by an efficient algorithm? In fact, are there problems that cannot be solved at all? To formally address these questions, we'll need a mathematical model that captures our intuition about "problems," "algorithms," etc. This chapter presents a relatively simple one that acts as a nice starting point.

1.1 Finite Automata

A *deterministic finite automaton* (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ consisting of:

1. a finite set Q, where each element of Q is called a "state,"
2. a finite set Σ called the "alphabet," where each element of Σ is called a "symbol" or "character,"
3. a function $\delta\colon Q \times \Sigma \to Q$ called the "transition function,"
4. a state $q_0 \in Q$ called the "start state,"
5. and a set $F \subseteq Q$ of the "accept" states.

Fig. 1 contains a DFA $M_1 = (Q, \Sigma, \delta, q_0, F)$ where $Q = \{q_1, q_2, q_3\}, \Sigma = \{\texttt{0}, \texttt{1}\}$, $q_0 = q_1$, and $F = \{q_2\}$. The transition function δ is defined as follows:

$$\begin{aligned} \delta(q_1, \texttt{0}) &= q_1 & \delta(q_1, \texttt{1}) &= q_2 \\ \delta(q_2, \texttt{0}) &= q_3 & \delta(q_2, \texttt{1}) &= q_2 \\ \delta(q_3, \texttt{0}) &= q_2 & \delta(q_3, \texttt{1}) &= q_2 \end{aligned}$$

In a typical diagram of a DFA, each circle represents a state, each double circle represents an accept state, the arrows are labeled and represent the transition function, and the state with an extra arrow pointing to it is the start state.

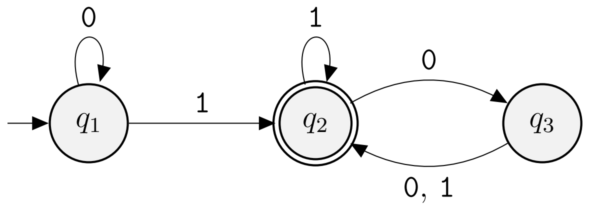

Figure 1: (Figure 1.4 from the book) A DFA M_1 with 3 states. For any string w, M_1 accepts w if and only if w contains at least one 1 and an even number of 0s follow the last 1 in w.

It might not look like it, but every DFA corresponds to an algorithm. The input is any string whose characters are from Σ (including the empty string, which we denote by ε), and the output is either "accept" or "reject." The algorithm is the following: starting at the start state q_0, read each character of the input one at a time from left to right. If we're at some state q_i and reading some character c, we move to the state $\delta(q_i, c)$ and follow the corresponding arrow in the diagram. Once we're done reading the last character, if we're in an accept state then we accept, and otherwise we reject. If we were to implement a DFA in a programming language like Python, the function would basically be a while loop containing a bunch of if statements.[1]

A *language* is simply a set of strings. For any DFA M, we let $L(M)$ denote the set of strings accepted by M and say that M *recognizes* $L(M)$. For example, the DFA defined above recognizes the language

$$L(M_1) = \{w \mid w \text{ contains at least one } \texttt{1} \text{ and an even number of } \texttt{0}\text{s follow the last } \texttt{1} \text{ in } w\}.$$

A language is a *regular language* if there exists a DFA that recognizes it. Every DFA recognizes exactly one language (possibly $\emptyset$), but as you might guess, it is not the case that every language is regular. For example, as we will see later in this chapter, the language $\{\texttt{0}^n\texttt{1}^n \mid n \geq 0\} = \{\varepsilon, \texttt{01}, \texttt{0011}, \ldots\}$ is nonregular. This is a conceptually significant result, since it illustrates a limitation of DFAs — nobody, no matter how clever they are, will ever be able to design a DFA that recognizes that language. But before we get there, let's explore regular languages a little more deeply.

Our first theorem states that the "class" (basically a set whose elements are themselves sets) of regular languages is *closed* under the union operation, i.e., the union of two regular languages is regular.

Theorem 1.1. *If A_1 and A_2 are regular languages, so is $A_1 \cup A_2$.*

Proof (sketch). Since A_1 and A_2 are regular, there exist DFAs M_1 and M_2 that recognize them. Our goal is to design a DFA M_3 that recognizes $A_1 \cup A_2$. In Python, we could implement M_3 as follows: given w, run $M_1(w)$, then run $M_2(w)$, and accept if and only if at least one of the two DFAs accepts. But this isn't a DFA, because DFAs can only read the input once; a DFA cannot "start over" at any point.

Instead, our DFA M_3 will *simulate* M_1 and M_2 by running $M_1(w)$ and $M_2(w)$ "at the same time" and keeping track of two states at once. More specifically, each state in M_3 will represent a pair of states: one from M_1 and one from M_2. If we're at state (s_1, s_2) and reading character c, we move to state $(\delta_1(s_1, c), \delta_2(s_2, c))$.[2] Once we've read the entire input, if we're in state (t_1, t_2), we accept if and only if $t_1 \in F_1$ or $t_1 \in F_2$. $\square$

Let's continue to expand the class of regular languages; we want to see how far we can go with DFAs. Theorem 1.1 tells us that one way to create a regular language is by taking the union of any two regular languages. Another way is to concatenate: if A_1 and A_2 are languages, then their concatenation is denoted by $A_1 \circ A_2$, and it contains the strings that consist of a string in A_1 as a prefix, a string in A_2 as a suffix, and nothing in between. If A_1 and A_2 are regular, then so is $A_1 \circ A_2$, but our proof strategy seems hard to apply. If

[1]Throughout these notes, I will use "Python" to refer to our usual model of computation (with loops, conditionals, functions, arrays, etc.), which recognizes many more languages than DFAs do. Keep in mind that eventually, we will study the "Python" model more formally.

[2]Sometimes, when I believe the context is relatively clear, I'll use notation such as δ_1 without defining it. In this case, for all $i \in \{1, 2\}$, δ_i is the transition function for the DFA M_i.

$w \in A_1 \circ A_2$, then $w = xy$ for some $x \in A_1$ and $y \in A_2$, but if we're just given w, how do we "know" where x ends and y begins?

1.2 Nondeterminism

In this section, we'll extend the capabilities of DFAs to allow for nondeterminism, which helps us prove that the class of regular languages is closed under concatenation. Figure 2 shows an example of a nondeterministic finite automaton (NFA) N_1.

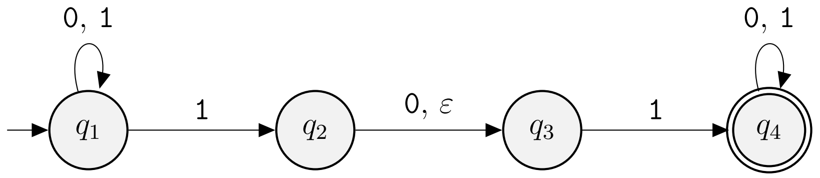

Figure 2: (Figure 1.27 from the book) An NFA N_1 with 4 states. If we run N_1 on the input $w =$ `010110`, we'll see that in the end, there are copies of N_1 in q_1, q_3, and q_4. Since q_4 is an accept state, N_1 accepts w. In general, N_1 accepts a string w if and only if w contains `101` or `11` as a substring.

The NFA "algorithm" is a parallelized version of the DFA algorithm: if an NFA N is at a state q and there's at least one ε-arrow leaving q, it creates one copy per ε-arrow, each copy follows exactly one ε-arrow, and the original copy stays at q.[3] (The NFA "pauses" its reading of the input when this happens.) If N is at state q and reading some character c of the input, it creates one copy per c-arrow, and each copy follows exactly one c-arrow. (The original copy disappears.) If there are no c-arrows leaving q, then that copy of the machine becomes "dead." In the end, the overall algorithm accepts if and only if *at least one* living copy is in an accept state.

Let's define NFAs more formally: a *nondeterministic finite automaton* (NFA) has exactly the same parts as a DFA $(Q, \Sigma, \delta, q_0, F)$, except the transition function δ has domain $Q \times \Sigma_\varepsilon$ (rather than $Q \times \Sigma$) where $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ and codomain $\mathcal{P}(Q)$, where $\mathcal{P}(Q)$ is the power set of Q (the elements of $\mathcal{P}(Q)$ are the subsets of Q, including $\emptyset$ and Q).

Hopefully it's clear that an NFA is like a fancier version of a DFA: an NFA can create multiple copies of itself if it wants to. But surprisingly, NFAs are not able to recognize a bigger class of languages than DFAs. In other words, every NFA can be converted into an equivalent DFA, where we say that two machines are *equivalent* if they recognize the same language.

Theorem 1.2. *For every NFA N, there exists an equivalent DFA M.*

Proof (sketch). In the proof of Theorem 1.1, M_3 keeps track of which two states we'd be in if we ran M_1 and M_2 at the same time on the same input. This proof is similar but more complicated: if N has n states, then M has 2^n states, each representing the subset of states in N that might be the location of some living copy of N. As N runs, the set of states that have a living copy of N changes in some potentially messy — but traceable — way, which defines the transition function for M. $\square$

[3] I'll use the term "ℓ-arrow" to refer to an arrow whose label is ℓ.

Thus, to show that a language is regular, it suffices to describe an NFA that recognizes it. When describing an NFA, we can think of the NFA as "guessing" the "correct" arrow to take whenever there are multiple choices, such that eventually, it arrives at an accept state. Along the way, the NFA makes various copies of itself (i.e., the computation tree grows branches), but it doesn't matter if they die, accept, or reject. (However, we need to make sure that the NFA doesn't accept strings that we don't want it to accept.)

To make this "guessing" concept more concrete, let's apply it to sketch an alternative proof of Theorem 1.1 (i.e., if A_1 and A_2 are regular, so is $A_1 \cup A_2$).

Alternative proof (sketch) of Theorem 1.1. Given NFAs N_1 and N_2, we construct an NFA N by starting with the "union" of N_1 and N_2, adding a new start state s, and adding an ε-arrow from s to the start states of N_1 and N_2. (Notice that this is much simpler than M_3 from the original proof: there's just one additional state and two additional arrows.)

For any input w, $N(w)$ immediately creates a copy of itself that runs $N_1(w)$ and another that runs $N_2(w)$. We can imagine that N is "trying" to accept w, "guesses" which NFA (either N_1 or N_2) will accept w, and takes the corresponding ε-arrow. In reality, the NFA isn't "trying" to do anything. Instead, its computation tree splits into three branches: one stays at s, one moves to N_1, and one moves to N_2. If $w \in A_1 \cup A_2$, then at least one branch will accept, so N will accept w. If $w \notin A_1 \cup A_2$, then none of the branches will accept, so N will reject w. However, this idea that the NFA is "trying" to accept w and "guessing" its way towards an accept state is useful for reasoning about nondeterminism. $\square$

Let's use this "guessing" concept again to show that the class of regular languages is closed under concatenation.

Theorem 1.3. *If A_1 and A_2 are regular languages, so is $A_1 \circ A_2$.*

Proof (sketch). Let N_1 and N_2 be NFAs that recognize A_1 and A_2. The states of our NFA N are $Q_1 \cup Q_2$, the start state is that of N_1, and the accept states are F_2. There is an ε-arrow from each state in F_1 to the start state of N_2. For any input w, these additional ε-arrows allow N to "guess" the location of the "spliting point" in w, i.e., the spot where the prefix from A_1 ends and the suffix from A_2 begins. (In reality, whenever N enters a state in F_1, it creates a copy that "tries" that point in w as the splitting point by switching to N_2.)

More specifically, if $w \in A_1 \circ A_2$, then $w = xy$ for some $x \in A_1$ and $y \in A_2$. Our NFA N runs $N_1(w)$ first, and at some point, it will read the last character of x and enter F_1. At that point, one copy of N will begin running N_2 on y, and N_2 accepts y, so N accepts w. Conversely, for any input w, N accepts w only if N_1 accepts some prefix of w and N_2 accepts the rest of w, which implies $w \in A_1 \circ A_2$. $\square$

Besides union and concatenation, another operation is called *star*: if A is a language, A^* is the language that includes the strings that are the concatenation of strings in A (and no other strings). For example, if $A = \{\texttt{0}, \texttt{1}\}$, then $A^* = \{\varepsilon, \texttt{0}, \texttt{1}, \texttt{01}, \texttt{10}, \ldots\}$ is the set of finite binary strings. Note that for any language A, $\varepsilon \in A^*$.

Theorem 1.4. *If A is a regular language, so is A^*.*

Proof (sketch). Let N be an NFA recognizing A. This proof is similar to the proof of Theorem 1.3, but now, in our design of an NFA N^* that recognizes A^*, we need to allow N^* to "guess" that it should "restart" N whenever it reaches an accept state in N.

More specifically, N^* has the same states and transition function as N, but we add an ε-arrow from each accept state to the start state. The NFA N^* also needs to accept ε (even if N doesn't), and one natural way to do this is to make the starting state of N an accept state in N^*. But this doesn't quite work (Exercise 1.15 from the book asks for a counterexample); a solution that does work is to add a new start state s, which is an accept state, and an ε-arrow from s to the original start state of N. □

1.3 Regular Expressions

Let's briefly turn our attention away from DFAs and NFAs and toward regular expressions. A *regular expression* is a way of specifying a set of strings (i.e., a language) according to some desired pattern. For example, suppose we're searching a text document to find the number of strings of even length. Assuming that we're only working with lowercase characters, we could search for ($\texttt{aa}, \texttt{ab}, \texttt{ac}, \ldots, \texttt{zz}$) and sum their frequencies to get the total, but that would take a long time and we might make a mistake. Instead, if we know the regular expression for "strings of even length," then we'd improve both our speed and accuracy.

The exact definition of a regular expression is recursive, which makes it a bit tricky, so we won't state it formally. Instead, here's an informal version: given an alphabet Σ, a regular expression is a string whose characters are in the set $\Sigma \cup \{\varepsilon, \emptyset, \cup, \circ, ^*, (,)\}$. Every regular expression describes a language, and the symbols $\{\cup, \circ, ^*\}$ correspond to union, concatenation, and star. To keep the regular expression short, we typically omit these symbols, along with parentheses, if the described language is clear.

Despite the difference between a regular expression R and the *language described* by R, we often think of them as the same thing. Here are a few examples from Example 1.53 in the book, assuming $\Sigma = \{\texttt{1}, \texttt{0}\}$:

Regular expression R	Language described by R
$\texttt{0}^*\texttt{1}\texttt{0}^*$	$\{w \mid w$ contains exactly one $\texttt{1}\}$
$\Sigma^*\texttt{1}\Sigma^*$	$\{w \mid w$ contains at least one $\texttt{1}\}$
$\texttt{1}^*(\texttt{0}\texttt{1}\texttt{1}^*)^*$	$\{w \mid$ every $\texttt{0}$ in w is followed by at least one $\texttt{1}\}$
$(\Sigma\Sigma)^*$	$\{w \mid w$ has even length$\}$
$(\texttt{0} \cup \varepsilon)(\texttt{1} \cup \varepsilon)$	$\{\varepsilon, \texttt{0}, \texttt{1}, \texttt{01}\}$

The fact that we've used the term "regular" in two different ways (for languages and expressions) should be telling: This class of languages described by regular expressions is exactly the same as the class of regular languages.

Theorem 1.5. *A language is regular if and only if some regular language describes it.*

Proof (sketch). The backward direction is easier: given a regular expression R that describes some language A, we can define an NFA that recognizes A using a recursive procedure similar to the proofs that the class of regular languages is closed under union, concatenation, and star. For example, if $R = R_1 \circ R_2$ where R_1 and R_2 are regular expressions, we can recursively construct NFAs N_1 and N_2 that recognize R_1 and R_2 and combine them using the procedure described in the proof of Theorem 1.3.

To prove the forward direction, we show that we can convert any DFA M that recognizes a language A into a regular expression that describes A. This proof is quite involved, so we omit it from these notes. $\square$

Theorem 1.5 tells us that for any language A, if we can give a regular expression R that describes A, then A must be regular. In particular, the proof of the theorem shows us how we can convert R to an NFA N that recognizes A, and if we wanted to go further, we can use Theorem 1.2 to convert N into an equivalent DFA M. (Finally, recall that a DFA is basically just a while loop containing a multiple if statements.)

1.4 Nonregular Languages

In this section, we'll see how we can prove that a language is *not* regular. In particular, we'll learn about *fooling sets* and the *pumping lemma*, two techniques for proving that a language is nonregular. The content on the pumping lemma is based on the book, but the content on fooling sets is based on the notes *Models of Computation* by Jeff Erickson.

Technique 1: Fooling Sets

As mentioned earlier in this chapter, the language $L = \{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\} = \{\varepsilon, \mathtt{01}, \mathtt{0011}, \ldots\}$ is nonregular. Here's an explanation: for contradiction, suppose there exists a DFA M that recognizes L, and consider the ending states when we run M on each string in $\mathtt{0}^* = \{\varepsilon, \mathtt{0}, \mathtt{00}, \ldots\}$. Since M only has a finite number of states but $\mathtt{0}^*$ has infinitely many elements, there must be distinct integers i, j such that $M(\mathtt{0}^i)$ and $M(\mathtt{0}^j)$ end in the same state. But this means $M(\mathtt{0}^i\mathtt{1}^i)$ and $M(\mathtt{0}^j\mathtt{1}^i)$ also end in the same state, which contradicts the assumption that M recognizes L because this state cannot simultaneously accept $\mathtt{0}^i\mathtt{1}^i \in L$ and reject $\mathtt{0}^j\mathtt{1}^i \notin L$. This argument shows that $\mathtt{0}^*$ is a "fooling set" for L: the strings in $\mathtt{0}^*$ can "fool" M into not being able to recognize L.

In general, a *fooling set* for a language A is a set F of strings such that, for any distinct strings $x, y \in F$, there is a string z, which we call a "distinguishing suffix," such that exactly one string in $\{xz, yz\}$ is in A.

Theorem 1.6. *If F is a fooling set for a language A, then any DFA M that recognizes A has at least $|F|$ states. In particular, if F is an infinite set, then A is nonregular.*

Proof. We claim that for any strings $x, y \in F$, $M(x)$ and $M(y)$ end in different states; this implies M has at least $|F|$ states. For contradiction, assume $M(x)$ and $M(y)$ end in the same state, and let z be their distinguishing suffix. Since $M(x)$ and $M(y)$ end in the same state, $M(xz)$ and $M(yz)$ also end in the same state. If this state is an accept state, then M accepts both xz and yz; if not, then M rejects both xz and yz. Either way, M cannot "distinguish" between xz and yz (i.e., accept exactly one string in $\{xz, yz\}$), which means M does not actually recognize A.

If F is an infinite set, then M cannot even be a DFA (much less one that recognizes A) because the definition of DFA requires that M has a finite number of states. $\square$

Theorem 1.6 gives us one way to prove that a language A is nonregular: construct an infinite fooling set. For $A = \{\texttt{0}^n\texttt{1}^n \mid n \geq 0\}$, we saw that $\texttt{0}^*$ is a fooling set, where the distinguishing suffix for $x = \texttt{0}^i$ and $y = \texttt{0}^j$ is $z = \texttt{1}^i$.

Technique 2: The Pumping Lemma

Another way to prove that a language is nonregular is by using the pumping lemma.

Theorem 1.7 (Pumping lemma)**.** *If A is a regular language, then there exists a number p (the "pumping length") such that for all $s \in A$ satisfying $|s| \geq p$, s can be split into $s = xyz$ such that (1) for all $i \geq 0$, $xy^iz \in A$, (2) $|y| > 0$, and (3) $|xy| \leq p$.*

Before we sketch the proof, let's see how, at a high level, the pumping lemma helps us prove that a language A is nonregular. For contradiction, assume A is regular. Then A has a pumping length p, and the pumping lemma states that every string s satisfying $|s| \geq p$ can be "pumped," i.e., A must contain infinitely many "longer versions" of s. However, if we can show that some string generated in this way isn't actually in A, then we can conclude that A is nonregular.

Proof (sketch). Let M be a DFA that recognizes A, and let p be the number of states in M. For any $s \in A$ such that $|s| \geq p$, consider the sequence of states Q that M enters when it runs on s. Notice that $|Q| = |s| + 1 > p$, so some state q appears at least twice (by the pigeonhole principle) within the first $p + 1$ terms of Q. We split s into xyz as follows: x is the prefix of s that M reads immediately before entering q, y is the part of s that gets read between the first and second time M enters q, and z is the suffix of s after y.

Now we can check that all three conditions are satisfied: (1) since $M(x)$ and $M(xy)$ lead to the same state and M accepts xyz, (2) since q appears (at least) twice in Q and y is the portion that s that M reads between those appearances, and (3) since q appears (at least) twice in the first $p + 1$ terms of Q. $\square$

Let's apply the pumping lemma to show that our favorite nonregular language $L = \{\texttt{0}^n\texttt{1}^n \mid n \geq 0\}$ is nonregular. If L were regular, then it has a pumping length p. Let $s = \texttt{0}^p\texttt{1}^p$, and let's consider the different ways to split s into xyz:

1. If $y = \texttt{0}^j$ for some j, then $xy^2z \notin L$ because xy^2z has more 0s than 1s.
2. If $y = \texttt{1}^j$, then $xy^2z \notin L$ because xy^2z has fewer 0s than 1s. (Also, we wouldn't have $|xy| \leq p$.)
3. If $y = \texttt{0}^i\texttt{1}^j$ for some i and j, then $xy^2z \notin L$ because it has some 1s before some 0s. (Also, we wouldn't have $|xy| \leq p$.)

So regardless of how we split s, $xy^2z \notin L$, so L is nonregular.

2 Context-Free Languages

2.1 Context-Free Grammars

In Chapter 1, we studied regular languages from two perspectives: regular expressions and finite automata. We'll do the same thing in this chapter but for a broader class of languages known as *context-free languages* (CFLs).

Just as regular expressions describe regular languages, context-free grammars describe context-free languages. But context-free grammars are more powerful, i.e., the class of CFLs contains all regular languages and more, including $\{\texttt{0}^n\texttt{1}^n \mid n \geq 0\}$. They were originally used to study human languages (e.g., verbs, nouns, etc.) but are also used in the design of compilers and interpreters of programming languages.

Here is an example of a context-free grammar G:

$$\begin{aligned} A &\to \texttt{0}A\texttt{1} \\ A &\to B \\ B &\to \texttt{\#} \end{aligned}$$

Each line of G is a *substitution rule* or *production.*[1] Each rule has a *variable* on the left and a string of variables and *terminals* on the right, and one of the variables is the *start variable.* Note that ε can also be on the right-hand side of a rule. For convenience, we often use the "|" symbol to represent the "or" of multiple substitution rules, as in $A \to \texttt{0}A\texttt{1} \mid B$ (instead of the first two rules above).

In G, the variables are $\{A, B\}$, the terminals are $\{\texttt{0}, \texttt{1}, \texttt{\#}\}$, and the start variable is A. The language denoted by G, denoted $L(G)$, is $\{\texttt{0}^n\texttt{\#}\texttt{1}^n \mid n \geq 0\}$. A *derivation* is the sequence of substitutions used to generate a string in $L(G)$. For example, here is a derivation of the string $\texttt{00\#11} \in L(G)$:

$$A \Rightarrow \texttt{0}A\texttt{1} \Rightarrow \texttt{00}A\texttt{11} \Rightarrow \texttt{00}B\texttt{11} \Rightarrow \texttt{00\#11}$$

We can also represent the derivation by a *parse tree* rooted at the start symbol; each node has children according to a substitution rule. The leaves of the parse tree, combined from left to right, represent the generated string; see Figure 3.

Ambiguity

A derivation is a *leftmost derivation* if, at every step, it replaces the leftmost remaining variable. A grammar is *ambiguous* if it can generate some string using at least two leftmost derivations. For example, the grammar below is ambiguous:

$$E \to E\texttt{+}E \mid E\texttt{×}E \mid \texttt{(}E\texttt{)} \mid \texttt{a}$$

For example, it can generate `a+a×a` by first applying either $E \to E\texttt{+}E$ or $E \to E\texttt{×}E$. Some CFLs are inherently ambiguous (i.e., they can only be generated by ambiguous CFGs), while

[1]The term "context-free" refers to the fact that in a context-free grammar, the rules are independent of each other. When deriving strings, we do not need to consider the *context* when determining which rules we can apply. The use of the term "regular" is more arbitrary; it was originally chosen by Stephen Cole Kleene.

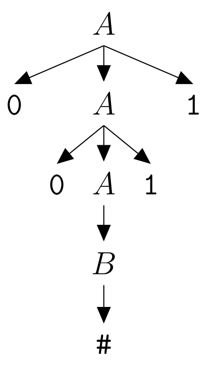

Figure 3: A parse tree for `00#11` in the grammar G.

others are not. For example, the ambiguous grammar above describes the same grammar as the unambiguous one below:

$$\begin{aligned} E &\to E\texttt{+}T \mid T \\ T &\to T\texttt{×}F \mid F \\ F &\to \texttt{(}E\texttt{)} \mid a \end{aligned}$$

To generate `a+a×a`, we must start with $E \to E\texttt{+}T$.

Chomsky Normal Form

Recall that we can convert any NFA into a DFA that recognizes the same language. Similarly, we can convert any CFG into a form called Chomsky normal form. In this form, every rule is of the form $A \to BC$ or $A \to a$ where A, B, C are variables, B and C are not the start variable, and a is a terminal. We are also allowed to include $S \to \varepsilon$, where S is the start variable.

Theorem 2.1. *Every CFL is generated by a CFG in Chomsky normal form.*

Proof (sketch). Given a CFG G, we can modify it to a CFG G' such that G and G' describe the same language and G' is in Chomsky normal form. To start, we add a new start variable S_0 and the rule $S_0 \to S$, where S was start variable of G. This ensures that the start variable is not in the right-hand side of any rule. Next, we replace rules of the form $A \to \varepsilon$ for all $A \neq S$ by adding $R \to uv$ for every rule $R \to uAv$. Along these lines, we can keep replacing and adding rules to G until we reach our goal. □

2.2 Pushdown Automata

Pushdown Automata (PDA) are a computational model that recognizes CFLs, the same way NFAs are a computational model that recognizes regular languages. More specifically, in this section, we consider nondeterministic PDAs because the class of languages they recognize is the class of CFLs. Unlike NFAs, nondeterministic PDAs recognize a larger class of languages than their deterministic counterparts.

A PDA is like an NFA: it has states Q, a transition function δ, a start state q_0, and accept states $F \subseteq Q$. Additionally, a PDA has a *stack*; it can push symbols to it and pop symbols from it. More formally, δ has domain $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$, where Σ is the input alphabet, Γ is the stack alphabet, and $A_\varepsilon = A \cup \{\varepsilon\}$. Given a state, input symbol, and stack symbol, the PDA can both (1) enter a new state and (2) push a symbol onto the stack or replace the symbol on the stack with another symbol (possibly ε, which represents popping from the stack). Since the PDA is nondeterministic, δ specifies a subset of all possible outcomes, i.e., the codomain of δ is $\mathcal{P}(Q \times \Gamma_\varepsilon)$.

Example. Here is the informal description of a PDA that recognizes $L = \{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$: when reading a 0s, push it onto the stack. When reading a 1, pop one 0 off the stack per 1. If we finish reading the input exactly when the stack becomes empty, then accept. Otherwise (i.e., if the stack becomes empty too early, never becomes empty, or a 0 appears after a 1), reject. The formal description is in Figure 4. In a diagram of a PDA, we use "$a, b \to c$" to signify that when the PDA is reading a from the input, it replaces the symbol b on the top of the stack with a c.

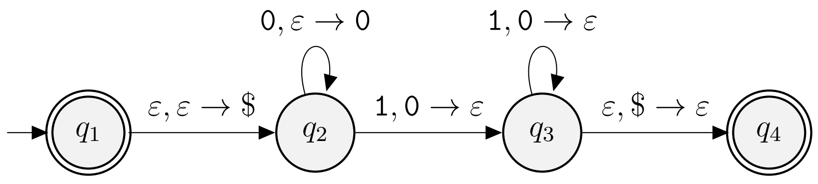

Figure 4: (Figure 2.15 from the book) A PDA that recognizes $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$. We start by pushing a \$ onto the stack, and the PDA "knows" the stack is empty when it sees the \$. The loop at q_2 represents pushing 0s to the stack, and the loop at q_3 represents popping them from the stack.

As we've already implied, PDAs and CFGs are equivalent in power:

Theorem 2.2. *A language is context free if and only if it is recognized by some PDA.*

Proof (sketch). Let G be a CFG; we can convert G to an equivalent PDA P. Given a string w, P needs to determine if there is a derivation for w in G. The PDA starts with the start variable in its stack, (nondeterministically) "guesses" the next substitution rule to make until it arrives a string s of terminal, and accepts if $s = w$. As P guesses, it matches a prefix of w with a prefix of the intermediate string and stores the rest of the intermediate string, starting with the first variable, in the stack for future processing.

Conversely, we can convert any PDA P to an equivalent CFG G. For each pair of states (p, q) in P, G has a variable A_{pq} that generates the set of strings that take P from p to q without changing its stack. By making some simplifying assumptions about P, we can compute A_{pq} using the recursive rules $A_{pp} \to \varepsilon$, $A_{pq} \to aA_{rs}b$ and $A_{pq} \to A_{pr}A_{rq}$ for states r and s "between" p and q. □

Since an NFA can be viewed as a PDA that ignores its stack, every regular language is also a context-free language.

2.3 Non-Context-Free Languages

Recall that the pumping lemma helps us prove that a language is nonregular. There is a similar, more complicated version for CFLs.

Theorem 2.3 (Pumping lemma for CFLs). *If A is a CFL, then there exists a number p (the "pumping length") such that for all $s \in A$ satisfying $|s| \geq p$, s can be split into $s = uvxyz$ such that (1) for all $i \geq 0, uv^ixy^iz \in A$, (2) $|vy| > 0$, and (3) $|vxy| \leq p$.*

Proof (sketch). This proof, like the one for the pumping lemma for regular languages, also uses the pigeonhole principle. Let G be a CFG that describes a language A. Intuitively, consider a very long string $s \in A$. The parse tree T for s is very tall, so on some path from the root to a leaf, some variable R gets repeated. We can do some "surgery" on T by replacing the subtree rooted at the lower appearance of R (which generates x) with the subtree rooted at the upper appearance of R (which generates vxy). The resulting tree is still a valid parse tree in G, so its corresponding string is in A. $\square$

We've used the pumping lemma to show that $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$ is nonregular; we can similarly show that $L = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ is not context free. The proof is quite similar (consider $s = \mathtt{a}^p\mathtt{b}^p\mathtt{c}^p$), so we omit it.

2.4 Deterministic Context-Free Languages

[unfinished]

3 The Church-Turing Thesis

We've seen a few models of computation: DFAs, NFAs, and PDAs. Recall that a DFA is basically a while loop containing if statements, an NFA is a DFA that can branch off in multiple directions at once (and every NFA has an equivalent DFA), and a PDA is an NFA that can use additional memory in the form of a stack. In this chapter, we introduce the Turing machine, a much more powerful model of computation that can do everything a "real computer" (e.g., any Python program) can do.

3.1 Turing Machines

Turing machines were first proposed by Alan Turing in 1936. The "heart" of a Turing machine (TM) is like an DFA (with states, transitions, etc.), but a TM has infinite memory in the form of a *tape*, which initially contains the input followed by infinitely many empty cells. The TM has a "read-write head," and at each step, it can read from one cell in the tape, write to that cell, and move the head left or right.[1]

Formally, a Turing machine M has 7 parts: states Q, a start state, an accept state, a reject state (not equal to the accept state), an input alphabet Σ (that does not contain a special "blank" symbol $\sqcup$), a tape alphabet Γ (where $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$), and a transition function $\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{\text{L}, \text{R}\}$.

Initially, the input w is in the first $|w|$ cells of the tape, the remaining cells all contain $\sqcup$, and the tape head is on the leftmost cell. If $\delta(q, a) = (r, b, \text{L})$, then M goes from state q to state r, replaces the a under the tape head with b, and moves the tape head left. (If the tape head is already on the leftmost cell, it stays put.) If M ever enters the accept or reject state, it immediately halts; if not, we say it "loops."

Configurations. A *configuration* of a TM M is a "snapshot" of its contents as it runs — more specifically, it comprises M's current state, tape contents, and tape head location. We can capture this information in one string as follows: if the current state is q, the contents of the tape before the tape head is u, and the contents of the tape starting at the tape head (ignoring the infinitely many blank symbols at the end) is v, then the configuration is uqv.

Languages. The set of strings accepted by M is the *language recognized by* M, denoted $L(M)$, and we say a language is *Turing-recognizable* if some Turing machine recognizes it. If M halts on all inputs, we call it a *decider*, and we say a language is *Turing-decidable* if some Turing machine decides it. Every Turing-decidable language is Turing-recognizable, but as we'll see, the converse is not true.

Giving a complete example of a Turing machine is quite tedious, so we will resort to informal descriptions. For example, the decider given in the book (Example 3.9) for the language $\{w\texttt{\#}w \mid w \in \{\texttt{0}, \texttt{1}\}^*\}$ has 10 states, and the corresponding diagram has 17 transition arrows. Here is an informal description: Zig-zag across the tape on both sides of # to check

[1] We can think of the tape as an array A with indices $\{1, 2, \ldots\}$, and the head is an index i. At each step, we can read $A[i]$, overwrite $A[i]$, and increase or decrease i by 1.

if the corresponding cells match. Replace matching symbols with an `x` to keep track of progress. When all symbols left of `#` are `x`, check if non-`x` symbols remain right of `#`, and accept/reject accordingly.

3.2 Variants of Turing Machines

There are many variants of Turing machines, including versions with multiple tapes and/or nondeterminism. Yet these variants all have the same power as the model described in the previous section, in the sense that all of them recognize the same class of languages. The fact that this class is not sensitive to changes in the model is one reason why Turing machines have been widely accepted and studied.

For example, suppose we allowed a TM to keep its tape head in its current location (rather than force it to move left or right). This might seem like a special feature, but we can simulate staying put by simply replacing each "stay put" transition with two transitions: the first moves the tape head right, and the second moves the tape head left.

Multitape Turing Machines

Suppose a TM M has k tapes and a read-write head for each tape. We can simulate these tapes on a TM S with one tape as follows: Create a new symbol `#` that separates the contents of the k tapes on our single tape. Furthermore, for each tape symbol a, add a new symbol $\dot{a}$. A dot above the symbol represents that the corresponding tape head is on that symbol. To simulate a single move by M, S scans its tape for the `#`s and $\dot{a}$ symbols and updates the tape accordingly. If S runs out of room between two `#`s (because some tape head in M moved sufficiently right), it shifts all symbols on the tape after this point one cell to the right.

Nondeterministic Turing Machines

Similarly, we can simulate a nondeterministic TM (NTM) N by a deterministic TM D that has three tapes. The definition of an NTM is the expected one: the copies of N form a computation tree, where the children of each node represent the copies determined by the transition function $\delta\colon Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{\text{L, R}\})$. An NTM N accepts w if (and only if) at least one of its computation branches accepts w. To simulate N, D can use breadth-first search in this tree to find a copy of N in the accepting state. Thus, a language A is Turing-recognizable if and only if an NTM recognizes A, and we can similarly show that A is decidable if and only if an NTM decides A.[2]

3.3 The Definition of Algorithm

Suppose we want to determine if we can find integers x, y, z such that

$$6x^3yz^2 + 3xy^2 - x^3 - 10 = 0.$$

[2]The book describes how we can translate multitape TMs and nondeterministic TMs into deterministic single-tape TMs. But it doesn't go into the details on how we can translate, say, a full-fledged Python program into an equivalent Turing machine. For more on that topic, I recommend *Introduction to Theoretical Computer Science* by Boaz Barak.

The solution $(x, y, z) = (5, 3, 0)$ works, but is there an algorithm for any polynomial? This was one of David Hilbert's 23 problems he posed as challenges in 1900. Believe it or not, the answer is no! But before anyone could prove a statement like this, we needed a formal definition of "algorithm." In 1936, Alonzo Church defined algorithms using a system called λ-calculus, and Alan Turing used Turing machines. These definitions were shown to be equivalent, and in fact, the *Church-Turing thesis* refers to the informal idea that these definitions capture our intuitive notion of an "algorithm" as a sequence of clear steps.

The rest of the book focuses on algorithms, with the TMs simply serving as the formal computational model in the definition of algorithm. The input is always a string, which could represent various objects (e.g., graph, grammar, DFA). We use $\langle A \rangle$ to denote the string representation of an object A. The details of the encoding do not matter, since any reasonable encoding can be understood and translated by a TM.

Example. Consider the following language:

$$A = \{\langle G \rangle \mid G \text{ is a connected undirected graph}\}.$$

For any graph G, one encoding $\langle G \rangle$ is to list its vertices in a sequence (with parentheses and commas), followed by its edges in a sequence. For example, we could use the string below to represent a graph with a triangle on $\{1, 2, 3\}$ and an additional edge $\{1, 4\}$:

$$\langle G \rangle = \texttt{(1,2,3,4)((1,2),(2,3),(3,1),(1,4))}$$

Here is a TM that decides A: given $\langle G \rangle$, mark the first node. Then repeat the following until no new nodes are marked: for each node in G, mark it if it is adjacent to a marked vertex. Finally, if all nodes in G are marked then accept; otherwise, reject. This description is quite high-level, but it is sufficient after we feel familiar with more detailed descriptions of TMs (i.e., specifying the states, defining the transition function, and so on).

4 Decidability

Decidability is what we usually consider when using an algorithm to solve a problem. Recall that a language is *Turing-decidable* (or just *Decidable*) if some Turing machine M *decides* it, that is, for every string w, $M(w)$ either outputs "accept" or "reject" — it does not enter an infinite loop. Similarly, when we solve problems (e.g., finding a shortest path in a graph), we often implicitly assume that any algorithm we give should always return something, even if it's not necessarily correct.[1]

4.1 Decidable Languages

The problems we'll consider are rather "meta" in the sense that the inputs are *themselves* algorithms (at an abstract level) that we've already seen. More specifically, we'll study problems concerning finite automata and context-free grammars (which are equivalent to pushdown automata), the two main topics of the previous chapters.

Regular Languages

Consider the following language:

$$A_{\mathsf{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}.$$

A decider for this language is simply the following: given $\langle B, w\rangle$, first check that the input properly represents a DFA. (Recall that a DFA is just a list of states, a transition function δ, etc., so there are many reasonable ways to represent any DFA as a string.) Then *simulate* B on w by tracking B's state on the tape and updating according to δ. Finally, if B accepts w then we accept $\langle B, w\rangle$; otherwise, we reject. Since we can convert any NFA to an equivalent DFA (see Theorem 1.2), the language A_{NFA}, defined analogously, is also decidable.

Here's another decidable language:

$$E_{\mathsf{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}.$$

A TM that decides E_{DFA} is similar to one that determines if a graph is connected: mark the start state of A, keep marking states that are reachable from a marked state until no new states get marked, and return "accept" if an accept state gets marked and "reject" otherwise.

Context-Free Grammars

Consider the language $A_{\mathsf{CFG}} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates string } w\}$. As you can probably guess, it is also decidable.

Theorem 4.1. *A_{CFG} is decidable.*

[1]Proving that a language A is decidable is equivalent to giving an algorithm to a "yes/no" problem. More specifically, determining whether or not a string w is in A amounts to solving the "yes/no" problem of checking whether or not w satisfies the property that defines the set A.

Proof (sketch). To decide A_{CFG}, the TM can't try every possible derivation, since it might get stuck in an infinite loop. Instead, we know that G can be converted to Chomsky normal form, and for any grammar in Chomsky normal form, the number of steps to generate a string of length n is $2n-1$ (Problem 2.26 in the book). So here's a decider for A_{CFG}: convert G to Chomsky normal form, list all derivations with $2|w|-1$ steps, and accept if and only if one of those derivations generate w. (This TM is not very efficient, but we're only concerned about decidability in this chapter — we'll consider efficiency in later chapters.) □

As was the case for DFAs, there's also the "emptiness problem" for CFGs:

$$E_{\mathsf{CFG}} = \{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}.$$

Given a CFG G, we need to determine if the start variable of G can generate a string of terminals. Again, we can't try every derivation, since we might get stuck in an infinite loop. Instead, we'll use the marking strategy "backward" as follows: mark all terminal symbols, keep marking any variable A such that G has a rule $A \to U_1U_2\cdots U_k$ where all U_i have been marked, and in the end, check if the start variable got marked.

Finally, we'll end this section with the following theorem:

Theorem 4.2. *Every context-free language is decidable.*

Proof. Let A be a CFL, and let S be a decider for A_{CFG} (see Theorem 4.1). Here is a TM M that decides A: given w, convert A to a CFG G in Chomsky normal form. If S accepts $\langle G, w\rangle$, then M accepts; otherwise, M rejects. □

To summarize, here are the relationships among the four different classes of languages we've seen so far:

$$\text{Regular} \subseteq \text{Context-Free} \subseteq \text{Decidable} \subseteq \text{Recognizable}.$$

4.2 Undecidability

We've arrived at a philosophically important part of the book. Recall that the Church-Turing thesis states that our understanding of computers/algorithms is captured by the Turing machine model of computation. Yet there are languages that are not decidable by any Turing machine, which implies that there are some problems that algorithms fundamentally cannot solve.

We've seen that A_{DFA} and A_{CFG} are decidable; now let's consider

$$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}.$$

Notice that A_{TM} is Turing-recognizable for the same reason A_{DFA} is decidable: given $\langle M, w\rangle$, our TM simulates M on w, accepts if M accepts, and rejects if M rejects. If M loops on w, then our TM loops as well, but that is acceptable for the purposes of recognizability. However, a TM that potentially loops is unacceptable if our goal is to show that A_{TM} is decidable. The next theorem implies that this goal is impossible.

Theorem 4.3. A_{TM} *is undecidable.*

The proof uses a technique called *diagonalization.* We'll skip the introduction to this technique and jump straight into the proof, but the book contains more details and provides an illustration of diagonalization by proving that the set of rational numbers is countable while the set of real numbers is uncountable.

Proof. For contradiction, assume H is a TM that decides A_{TM}. We construct a TM D that takes as input $\langle M \rangle$ (the description of a TM M) and does the following:

1. Run H on input $\langle M, \langle M \rangle \rangle$.
2. If H accepts, reject; otherwise, accept.

Something weird happens when we call D with input $\langle D \rangle$. First, let's assume D (as defined above) accepts $\langle D \rangle$, which means H rejects $\langle D, \langle D \rangle \rangle$. But if we follow the definition of H, that means D actually doesn't accept $\langle D \rangle$, contradicting our assumption that D accepts $\langle D \rangle$. On the other hand, similarly, if D rejects $\langle D \rangle$, then H must accept $\langle D, \langle D \rangle \rangle$, which means D actually accepts $\langle D \rangle$. Thus, neither "D accepts $\langle D \rangle$" nor "D rejects $\langle D \rangle$" is possible, so our initial assumption that H exists does not hold.

The proof above is complete, but it's interesting to directly see how we're using diagonalization. Suppose we order all possible TMs into a list $(M_1, M_2, \ldots)$. (Note that this is possible because any TM can be encoded in a finite string, so we can start by listing all TMs of length 1, then 2, etc.) Consider a 2-dimensional table where the i-th row and i-th column are labeled M_i and $\langle M_i \rangle$, respectively. The $(M_i, \langle M_j \rangle)$-th entry is "accept" if $H(M_i, \langle M_j \rangle)$ accepts and "X" otherwise; Figure 5 illustrates an example.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\cdots$	$\langle D \rangle$	$\cdots$
M_1	$\underline{\text{A}}$	X	A	$\cdots$	A	
M_2	X	$\underline{\text{X}}$	A	$\cdots$	X	
M_3	A	X	$\underline{\text{X}}$	$\cdots$	X	
$\vdots$						
D	X	A	A	$\cdots$	$\underline{?}$	
$\vdots$						

Figure 5: The results returned by $H(M_i, \langle M_j \rangle)$, where A stands for "accept" and X stands for "reject or loop."

Our TM D, built on H, appears in some row, and the proof considers whether $H(D, \langle D \rangle)$ accepts or rejects/loops. Since D is designed to be the opposite of H on every diagonal entry $(M_i, \langle M_i \rangle)$, the contradiction arises when we try to fill out the $(D, \langle D \rangle)$-th entry — it cannot be the opposite of itself. $\square$

In summary, A_{TM} is Turing-recognizable but not decidable. But there are languages that are not even Turing-recognizable! For any language A, let $\overline{A}$ denote the complement of A (i.e., the set of strings not in A). Furthermore, we say that A is *co-Turing-recognizable* if $\overline{A}$ is Turing-recognizable. In this case, some TM M recognizes $\overline{A}$, so $M(w)$ accepts every string $w \notin A$ and rejects or loops on all $w \in A$.

Theorem 4.4. *A language A is decidable if and only if A is both Turing-recognizable and co-Turing-recognizable.*

Proof. The forward direction is simpler: if A is decidable, then the decider M also recognizes A, so A is Turing-recognizable. Furthermore, $\overline{A}$ is also decidable (the decider for $\overline{A}$ can just output the opposite of M), so $\overline{A}$ is also Turing-recognizable, which means A is co-Turing-recognizable.

For the backward direction, let M_1 and M_2 be recognizers for A and $\overline{A}$, respectively. Here is a TM M that decides A: on input w, run $M_1(w)$ and $M_2(w)$ in parallel. More specifically, alternate between one step of M_1 and one step of M_2. If M_1 accepts w, our TM M accepts; otherwise, M rejects. Since w is either in A or $\overline{A}$, one of the recognizers $\{M_1, M_2\}$ accepts w; since M accepts/rejects w accordingly, M decides A. $\square$

We know that A_{TM} is undecidable (Theorem 4.3), and A_{TM} is Turing-recognizable (just simulate M on w), so Theorem 4.4 implies that $\overline{A_{\mathsf{TM}}}$ is not Turing-recognizable.

5 Reducibility

In the previous chapter, we saw that A_{TM} is undecidable (and $\overline{A_{\mathsf{TM}}}$ is not even Turing-recognizable). In this chapter, we'll see even more undecidable problems, and we'll show that they're undecidable using a technique called *reducibility*. The idea is the following: suppose a problem A reduces to another problem B. In that case, if we could decide (i.e., solve) B, we could also decide A. We are interested in the contrapositive: if A is undecidable, then B must also be undecidable.

5.1 Undecidable Problems from Language Theory

Here's another undecidable language:

$$H_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on } w\}.$$

(The book calls it $HALT_{\mathsf{TM}}$, but we'll use the shorter notation H_{TM}.) Deciding H_{TM} is equivalent to solving the *halting problem*. To prove that H_{TM} is undecidable, we'll use the reduction strategy defined above. In particular, we'll show that the task of deciding A_{TM} reduces to the task of deciding H_{TM}. The following proof is a typical example of reducibility.

Theorem 5.1. *H_{TM} is undecidable.*

Proof. For contradiction, suppose there exists a TM R that decides H_{TM}. Consider the following TM S that takes $\langle M, w\rangle$ as input:

1. Run R on $\langle M, w\rangle$.
2. If R rejects, reject.
3. Otherwise, simulate M on w and output whatever it outputs.

This TM S decides A_{TM}, but A_{TM} is undecidable. Thus, R cannot exist, so H_{TM} is also undecidable. $\square$

By reducing from A_{TM} or any another undecidable problem, we can similarly show the following languages are undecidable:

- $\{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$
- $\{\langle M\rangle \mid M \text{ is a TM and } L(M) \text{ is regular}\}$
- $\{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$

Computation Histories

Another way to prove that A_{TM} is reducible to other languages is by using computation histories. For any TM M and string w, a *accepting computation history* for M on w is a sequence of configurations of M on w, where each configuration can follow from the previous one and the final configuration is in the accept state. (A rejecting computation is defined analogously.) If M doesn't halt on w, then there's no computation history for M on w. So if M is deterministic, there exists at most one computation history for M on w. (Otherwise, there can be many computation histories, each corresponding to a computation branch.)

The reduction we'll give involves a *linear bounded automata* (LBA), which is a Turing machine that is not allowed to move its tape head off the portion of the tape containing the input. Many deciders that we've seen, including the ones for A_{DFA} and A_{CFG}, are LBAs. Theorem 5.9 in the book states that $A_{\mathsf{LBA}} = \{\langle M, w\rangle \mid M \text{ is an LBA that accepts string } w\}$, unlike A_{TM}, is decidable. The idea is that M can only enter a finite number of configurations because its tape head only has $|w|$ possible locations. This allows us to determine M's behavior on w (accept, halt, or loop) by simulating M on w for only a finite number of steps.

Now consider the following language:

$$E_{\mathsf{LBA}} = \{\langle M\rangle \mid M \text{ is an LBA and } L(M) = \emptyset\}.$$

We'll prove that E_{LBA} is undecidable using the computation history method.

Theorem 5.2. *E_{LBA} is undecidable.*

Proof. For contradiction, assume there exists a TM R that decides E_{LBA}. For an TM M and string w, we'll show how to construct an LBA B such that $L(B)$ is the set of accepting computation histories for M on w. If we can construct a B with this property, then we'd have a TM that decides A_{TM}:

1. Construct B using M and w.

2. Run R on $\langle B\rangle$ and output the opposite of R.

Now we construct the LBA B. Given any string x, B needs to determine if x is an accepting configuration history for M on w. Recall that at any point in a TM's calculation, its configuration consists of a state, a location on the tape head, and the contents of the tape. By scanning x, B can determine if the following conditions hold:

1. In x, the first configuration of M is in the initial state of M, w is on the tape, and the tape head is at the beginning of w.

2. Each configuration C_{i+1} follows from the previous one C_i according to the transition function δ of M. We can determine this by zig-zagging across C_{i+1} and C_i and checking that all symbols in the corresponding positions are the same, except for the ones under and adjacent to the tape head in C_i — those should change according to δ. We can keep track of our positions while zig-zagging by replacing any symbol a with $\dot{a}$, as we did to simulate a multitape TM (see Chapter 3).

3. The final configuration of M in x is in the accept state. □

Using the computation history method, we can also prove that the language $\{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \Sigma^*\}$ is undecidable (Theorem 5.13 in the book).

5.2 A Simple Undecidable Problem

So far, we have primarily focused on "meta" problems that themselves are about DFAs, CFGs, TMs, etc. In this section, we'll consider an undecidable problem that is less "meta" — it is called the *Post Correspondence Problem* (PCP). The input is a set of dominos, such as the following (this example is from the book):

$$\left\{ \left[\frac{\texttt{b}}{\texttt{ca}}\right], \left[\frac{\texttt{a}}{\texttt{ab}}\right], \left[\frac{\texttt{ca}}{\texttt{a}}\right], \left[\frac{\texttt{abc}}{\texttt{c}}\right] \right\}$$

Each domino consists of a string on the top and a string on the bottom. Our goal is to select and order a subset (repetition allowed) of the dominos such that the string on the top is equal to the string on the bottom, i.e., the two strings match. In the example above, there is a match containing 5 dominos where the final string is `abcaaabc`.

Theorem 5.3. *PCP is undecidable.*

Proof (sketch). The proof is a reduction from A_{TM} to PCP via computation histories. That is, for any TM M and string w, we will construct a set of dominos P such that a match in P is an accepting computation history for M on w. The strings on the dominos represent the configurations of M, and these strings are defined in a way that correspond to the steps of M. For example, some dominos represent M moving its tape head to the right, while others represent M moving its tape head to the left. Given these various types of dominos in the set P, finding a match in P becomes equivalent to finding an accepting computation history for M on w. □

5.3 Mapping Reducibility

We have already seen a few reductions; they often begin by assuming there exists a TM M that decides some language, and using M to build a TM that "decides" an undecidable like A_{TM}. In this section, we formalize this idea.

First, we say a function $f\colon \Sigma^* \to \Sigma^*$ is *computable* if some TM, on every input w, halts with just $f(w)$ on its tape.[1] The set of computable functions includes pretty much everything we're familiar with: arithmetic operations on integers, slightly modifying the description of a Turing machine, etc. We say a language A is *mapping reducible* to a language B (denoted by $A \leq_{\mathrm{m}} B$) if there exists a computable function $f\colon A \to B$ (called the *reduction* from A to B) such that for every string w, $w \in A$ if and only if $f(w) \in B$.

This definition of reducibility is a stronger notion our intuitive understanding of "reducing" one task to another. That is, if $A \leq_{\mathrm{m}} B$ and B is decidable by some TM M, then A is decidable: given w, compute $f(w)$, and return the output of running M on $f(w)$. However, in the intuitive understanding of "reduction," we're allowed to call the TM that decides B in various ways, not just at the end.

[1] This definition exists for rather technical reasons that the book doesn't discuss extensively. Basically, the purpose is to exclude functions that Turing machines cannot actually compute, such as solving the halting problem.

Undecidability of H_{TM} (alternative proof). To prove undecidability, as we have done, we use the contrapositive: if $A \leq_{\mathrm{m}} B$ and A is undecidable, then B is undecidable. For example, let's show H_{TM} is undecidable again (see Theorem 5.1). We know that A_{TM} is undecidable (see Theorem 4.3), so it suffices to show $A_{\mathsf{TM}} \leq_{\mathrm{m}} H_{\mathsf{TM}}$. The function f takes as input a string representing a TM M and string w, and it outputs a string representing a TM M' and the same string w. The TM M' runs M on x, accepts if M accepts, and loops if M rejects. Notice that M accepts w if and only if M' halts on w, as desired. (If the input to f is not a valid input to A_{TM}, f can output any string not in H_{TM}.)

Unrecognizablity. Similarly, if $A \leq_{\mathrm{m}} B$ and A is not Turing-recognizable, then B is not Turing-recognizable. Now consider the following language:

$$EQ_{\mathsf{TM}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}.$$

Theorem 5.4. *EQ_{TM} is neither Turing-recognizable nor co-Turing recognizable.*

Proof. First, we'll show $A_{\mathsf{TM}} \leq_{\mathrm{m}} \overline{EQ_{\mathsf{TM}}}$, which implies $\overline{A_{\mathsf{TM}}} \leq_{\mathrm{m}} EQ_{\mathsf{TM}}$, which implies EQ_{TM} is not Turing-recognizable. The function f is the following: given $\langle M, w\rangle$, return $\langle M_1, M_2\rangle$ where M_1 always rejects and M_2 runs $M(w)$ and accepts (whatever its input is) if M accepts w. Notice that f is indeed a reduction: If M accepts w, then $w \in L(M_2)$ while $L(M_1) = \emptyset$, so $\langle M_1, M_2\rangle \in \overline{EQ_{\mathsf{TM}}}$. Conversely, if M doesn't accept w, then $L(M_1) = L(M_2) = \emptyset$, so $\langle M_1, M_2\rangle \in EQ_{\mathsf{TM}}$.

Now we show $A_{\mathsf{TM}} \leq_{\mathrm{m}} EQ_{\mathsf{TM}}$, which implies $\overline{EQ_{\mathsf{TM}}}$ is not Turing-recognizable. The function g is identical to f, except M_1 always accepts. Notice that $L(M_1)$ contains all strings, and M accepts w if and only if $L(M_2)$ contains all strings. □

6 Advanced Topics in Computability Theory

[unfinished]

6.1 The Recursion Theorem

[unfinished]

6.2 Decidability of logical theories

[unfinished]

6.3 Turing Reducibility

[unfinished]

6.4 A Definition of Information

[unfinished]

7 Time Complexity

So far, we have been concerned with classifying languages as recognizable and/or decidable under various models of computation. The last part of the book is about complexity theory, the investigation of the time, memory, or other resources required to solve computational problems. We let $\mathcal{N} = \{1, 2, \ldots\}$ denote the set of natural numbers.

7.1 Measuring Complexity

The "running time" of an algorithm is the number of steps it takes in the worst case, as a function of the input length. More formally, if M is a deterministic TM, the *running time* (or *time complexity*) of M is a function $f\colon \mathcal{N} \to \mathcal{N}$, where $f(n)$ is the maximum number of steps that M takes on any input of length n.

Example. Consider the following TM M that decides $A = \{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$: on input w,

1. Scan w and reject if a `0` appears after a `1`.
2. While the tape has at least one `0` and one `1`: scan w and cross off a `0` and a `1`.
3. If there are no `0`s but at least one `1` left, reject. If there are no `1`s but at least one `0` left, reject. If nothing is left, accept.

We analyze the running time of TMs without delving into the implementation details (states, transition functions, etc.). In fact, we will use asymptotic notation, which further simplifies our analyses.[1] Step 1 above takes $O(n)$ steps, where $n = |w|$ denotes the input length. Repositioning the head back to the beginning of the tape takes another $O(n)$ steps. In Step 2, each scan takes $O(n)$ steps, and the number of scans is $O(n)$ because each scan reduces the number of remaining symbols by 2. Finally, Step 3 is a single scan that takes $O(n)$ steps. So the total running time is $O(n) + O(n^2) + O(n) = O(n^2)$.

For any function $t\colon \mathcal{N} \to \mathcal{N}$, we let $\mathrm{TIME}(t(n))$ denote the collection of languages that are decidable by an $O(t(n))$-time TM. Thus, the analysis above shows that $A \in \mathrm{TIME}(n^2)$. The book gives a more sophisticated TM decides A in $O(n \log n)$ time. With a second tape, we can decide A in $O(n)$ time: copy the `0`s to the second tape, and match them with the `1`s.

Relationships Among Models

The time complexity of a language depends on the model of computation. We consider three: single-tape TM, multitape TM, and nondeterministic TM.

Theorem 7.1. *For every function t satisfying $t(n) \geq n$ for all n, every $t(n)$-time multitape TM has an equivalent $O(t^2(n))$-time single-tape TM.*

[1]The book contains the formal definition of asymptotic notation; if you've taken other computer science courses, you might be familiar with it already.

Proof (sketch). In Section 3.2, we showed how a single-tape TM S can simulate a multitape TM M: concatenate all of M's tape contents on S's tape, separate each tape with a special symbol, and track the corresponding tape head locations. The TM S can simulate a single step of M by scanning and adjusting its tape in $O(t(n))$ time, since the total length of all of M's tapes is $O(t(n))$. Since M takes $O(t(n))$ time, S takes $O(t^2(n))$ time. □

Now we consider an analogous result for nondeterministic TMs. (Recall that a nondeterministic TM is a decider if every branch halts on every input.) The running time of a nondeterministic TM is defined as the maximum number of steps it uses on any branch.

Theorem 7.2. *For every function t satisfying $t(n) \geq n$ for all n, every $t(n)$-time nondeterministic TM has an equivalent $2^{O(t(n))}$-time deterministic single-tape TM.*

Proof (sketch). Again, recall the simulator from Section 3.2: if N is a nondeterministic TM, our deterministic TM M can simulate N by trying each node of N's computation tree using breadth-first search. Every branch has length at most $t(n)$, so if each node has at most b children, the number of nodes is $O(b^{t(n)})$. Since M can travel to any node from the root in $O(t(n))$ time, its total running time is $O(t(n)b^{t(n)}) = 2^{O(t(n))}$. (This last equality holds if $b < 2^k$ for some constant k: $t(n)b^{t(n)} < 2^{t(n)}2^{kt(n)} = 2^{O(t(n))}$.) By Theorem 7.1, we can convert M to a single-tape TM whose running time is $\left(2^{O(t(n))}\right)^2 = 2^{O(2t(n))} = 2^{O(t(n))}$. □

7.2 The Class P

In the previous section, we saw that with respect to running times, the advantage of multiple tapes is at most polynomial, while the advantage of nondeterminism is at most exponential. In fact, all "reasonable" deterministic models of computation are equivalent up to polynomials. This motivates the following definition: we let P denote the class of languages that are decidable in polynomial time on a deterministic single-tape TM, i.e., $\mathrm{P} = \cup_k \mathrm{TIME}(n^k)$. Generally speaking, problems in P (i.e., languages that can be decided in polynomial time) tend to be realistically solvable.[2] This includes the a significant fraction of algorithms studied in a typical introductory course on algorithms (e.g., BFS, DFS, Kruskal's, Dijkstra's).

Example. Consider the following language:

$$\mathsf{PATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a path from } s \text{ to } t\}.$$

To show that $\mathsf{PATH} \in \mathrm{P}$, we give a polynomial-time algorithm that decides PATH. Breadth-first search is sufficient: mark s, keep marking out-neighbors of marked vertices until no additional vertices are marked, and check if t is marked. If m is the number of vertices in G, then the algorithm makes at most m marks, and each mark takes m^k time for some constant k. The book excludes many details, but hopefully it's clear that this algorithm runs in polynomial time.

Now let's show that an entire class of languages is in P:

[2]It might seem like the difference between n and n^{100} is quite big, but by ignoring this difference, we free ourselves from needing to track various details (e.g., the exact movement of heads on tapes). This is analogous to ignoring the difference between n and $100n$ in asymptotic notation.

Theorem 7.3. *Every context-free language is in* P.

Proof (sketch). Let A be a CFL; recall that A is decidable (see Theorem 4.2). The decider for A relies on converting A to an equivalent grammar G in Chomsky normal form, listing a number of derivations, and checking if any of them produce the input string. The problem is that the number of derivations could be exponential, but we can turn to dynamic programming. Roughly speaking, this allows us to reduce the running time to polynomial by searching through the list of derivations more efficiently. $\square$

7.3 The Class NP

Many optimizations problems can be solved in exponential time by using brute force, and some problems can be solved in polynomial time. But for many other problems, we do not know if there exists a polynomial-time algorithm that solves the problem. However, it is often the case that we can efficiently verify that a proposed solution is indeed a solution. For example, consider the following language:

$$\mathsf{HamPath} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\},$$

where a *Hamiltonian path* is a path that includes every vertex exactly once. It is relatively easy to verify that a path P is Hamiltonian in polynomial time: just check that P starts at s, ends at t, and includes every vertex exactly once. However, it is not known whether a Hamiltonian path can be found in polynomial time.

This phenomenon — (seemingly) difficult to solve, but easy to verify — is a core concept in complexity theory. More formally, a *verifier* for a language A is an algorithm V such that

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

The running time of V is only a function of $|w|$, and the string c is known as a *certificate* or *proof* of membership in A. The certificate can often be thought of as a "proposed solution" (e.g., for $\mathsf{HamPath}$, c could just be a Hamiltonian path from s to t). NP is the class of languages that have polynomial-time verifiers.

The name "NP" is short for "nondeterministic polynomial-time" because there's an alternative definition based on nondeterministic Turing machines. (Recall that the running time of a nondeterministic TM is the number of steps in the longest computation branch.) In particular, we have the following theorem:

Theorem 7.4. *A language A is in* NP *if and only if there exists a nondeterministic polynomial-time TM that decides A.*

Proof (sketch). Suppose $A \in$ NP, and let V be a polynomial-time verifier for A, so for any string w, V runs in time n^k for some value k. Here is a nondeterministic TM that decides A: on input w, nondeterministically select a string c of length at most n^k, run V on $\langle w, c\rangle$, accept if V accepts, and reject otherwise.

Conversely, if an NTM N decides A, we can design a verifier V that works as follows: on input $\langle w, c\rangle$, simulate N on w, interpreting each symbol in c as a description of the nondeterministic choice that N should make at each step. (The details are similar to simulating a nondeterministic TM using a deterministic TM; see Theorem 3.16 in the book.) Accept if this branch of N accepts; otherwise, reject. $\square$

The class coNP. Every problem $A \in \mathrm{NP}$ has a complement problem $\overline{A}$; we let coNP denote the collection of languages that are complements of languages in NP. For example, the problem $\overline{\textsf{HamPath}}$ requires us to decide if a given graph does *not* have a Hamiltonian path. This problem is not obviously in NP, since it is not clear what kind of certificate could show that a graph does *not* have a Hamiltonian path. People generally believe that $\mathrm{NP} \neq \mathrm{coNP}$, but this has not been proven.

The nondeterministic version of $\mathrm{TIME}(t(n))$ is $\mathrm{NTIME}(t(n))$, the collection of languages that are decidable by an $O(t(n))$-time NTM. So $\mathrm{P} = \mathrm{TIME}(t(n))$, $\mathrm{NP} = \cup_k \mathrm{NTIME}(n^k)$, and $\mathrm{P} \subseteq \mathrm{NP}$. As the book puts it, "The question of whether P = NP is one of the greatest unsolved problems in theoretical computer science and contemporary mathematics." However, it is known that $\mathrm{NP} \subseteq \mathrm{EXPTIME} = \cup_k \mathrm{TIME}(2^{n^k})$, i.e., if a problem is verifiable in polynomial time, then it is solvable in exponential time. This captures the notion that problems in NP (e.g., HamPath) can be solved using brute force.

7.4 NP-completeness

Remark. The book does not define "NP-hard" in the main content (though it does in one of its problems), but it's a useful term, so we'll include it in these notes.

Here is a rough overview of the picture so far: problems in P are easy to solve, while some problems in NP, including HamPath, seem hard to solve. Nobody has been able to prove that HamPath is actually hard, but we *can* definitively state that HamPath is *harder* than some other problem X. If X is known to be seemingly hard, and HamPath is definitively harder than X, then this is evidence that HamPath is also hard.

To compare the hardness of problems, we use the notion of mapping reducibility from Section 5.3, but we restrict ourselves to polynomial time. More formally: A language A is *polynomial-time mapping reducible* (or simply *polynomial-time reducible*) to a language B if there exists a function $f\colon \Sigma^* \to \Sigma^*$ such that, for every string w, $w \in A$ if and only if $f(w) \in B$. This function f is known as a *polynomial-time reduction* from A to B, and there needs to exist a TM that computes f in polynomial time. We write $A \leq_{\mathrm{P}} B$ if A is polynomial-time reducible to B.

Theorem 7.5. *If $A \leq_{\mathrm{P}} B$ and $B \in \mathrm{P}$, then $A \in \mathrm{P}$.*

Proof. Let f be a polynomial-time reduction from A to B, and let M be a polynomial-time algorithm that decides B. The following algorithm decides A in polynomial time: on input w, compute $f(w)$, and output $M(f(w))$. □

A language B is *NP-hard* if $A \leq_{\mathrm{P}} B$ for every problem $A \in \mathrm{NP}$, and B is *NP-complete* if $B \in \mathrm{NP}$ and B is NP-hard. So if B is NP-complete and $B \in \mathrm{NP}$, then $\mathrm{P} = \mathrm{NP}$.

Theorem 7.6. *If $A \leq_{\mathrm{P}} B$ and A is NP-hard, then B is NP-hard.*

Proof. Consider any problem $C \in \mathrm{NP}$; we want to show $C \leq_{\mathrm{P}} B$. Since A is NP-hard, we know there exists a polynomial-time reduction from C to A. Combining this reduction with the one from A to B yields a polynomial-time reduction from C to B. □

Remark. The previous two theorems can be interpreted as follows: suppose $A \leq_{\mathrm{P}} B$. Let's overload notation and also let A (similarly for B) denote a binary "hardness value" of problem A, where $A = 0$ means we know that A is solvable in polynomial time, and $A = 1$ means we don't know if A is solvable in polynomial time. Under this interpretation, the inequality $A \leq B$ must hold. More specifically, Theorem 7.5 states that $B = 0$ implies $A = 0$ (i.e., if B is easy then A is easy), and Theorem 7.6 states that $A = 1$ implies $B = 1$ (i.e., if A is hard then B is hard).

So Theorem 7.6 tells us that to prove a problem B is NP-hard, we can do the following: find a problem A that is known to be NP-hard and prove $A \leq_{\mathrm{P}} B$. But how do we prove that a problem is NP-hard without relying on the NP-hardness of another problem? We address this question in the remainder of this section.

The Cook-Levin Theorem

The *satisfiability problem* is the following: the input is a Boolean formula containing variables and the operations $\wedge$ (AND), $\vee$ (OR) and $\neg$ (NOT); we use $\overline{x}$ to denote the negation of x. For example, $\phi = (\overline{x} \vee y) \vee (x \wedge \overline{z})$ is a Boolean formula containing three variables. A Boolean formula ϕ is *satisfiable* if there exists an assignment of each variable to either 0 or 1 such that ϕ evaluates to 1, where 0 is interpreted as FALSE and 1 is interpreted as TRUE. In our example, ϕ is satisfiable because the assignment $(x, y, z) = (0, 1, 0)$ makes $\phi = 1$. The language SAT is defined as

$$\mathsf{SAT} = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}.$$

The following theorem is named after Stephen Cook and Leonid Levin, two pioneers in complexity theory:

Theorem 7.7. SAT *is* NP-*complete.*

Proof (sketch). Let A be any language in NP; we need to show $A \leq_{\mathrm{P}} \mathsf{SAT}$. That is, we need to give an algorithm that converts every string in A (and no other strings) to a satisfiable Boolean formula. Since $A \in \mathrm{NP}$, there exists an NTM N that decides A in n^k time for some constant k. The high-level intuition is that the operations $\wedge, \vee, \neg$ allow us to build a primitive "computer," which we can use to build a Boolean formula based on N and w.

More specifically, we represent each computation branch of N using a table of size $n^k \times n^k$; each row represents a configuration of N on w. Our formula ϕ has variables based on the cell indices in the table, N's set of states, and N's tape alphabet. It is possible to combine these variables into ϕ such that a satisfying assignment for ϕ corresponds to an accepting tableau for N on w. The details are quite involved, so we omit them. $\square$

We now describe a well-known special case of SAT known as $\mathsf{3SAT}$. A *literal* is either a variable or its negation, and a *clause* is several literals joined by $\vee$s. In $\mathsf{3SAT}$, the input is a Boolean formula that is several clauses, each containing exactly three literals, joined by $\wedge$s. For example, the formula

$$\phi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3 \vee \overline{x_4})$$

contains four variables, two clauses, and is an element of 3SAT. Although 3SAT seems easier to solve than SAT since the set of possible inputs is restricted, this is not actually the case.

Theorem 7.8. 3SAT *is* NP*-complete.*

Proof (sketch). The proof of Theorem 7.7 produces a Boolean formula ϕ that is almost in the correct format (i.e., the $\wedge$ of clauses, where each clause has exactly three literals). We can modify ϕ until the resulting Boolean formula is in the form for 3SAT without affecting the truth value of ϕ. We omit the details, but as an example, suppose ϕ contains $C = (a_1 \vee a_2 \vee a_3 \vee a_4)$ as a clause (where each a_i is a literal), and consider replacing C with $C' = (a_1 \vee a_2 \vee z) \wedge (\overline{z} \vee a_3 \vee a_4)$. It is relatively straightforward to see that C is satisfiable if and only if C' is satisfiable. □

7.5 Additional NP-complete Problems

This section of the book proves that various problems, including HamPath, are NP-complete. Generally speaking, showing that a problem B is in NP is rather straightforward: simply describe a polynomial-time algorithm that verifies that a proposed solution to B is indeed a solution. The more difficult part is proving that B is NP-hard. The book typically does this by showing 3SAT $\leq_{\mathrm{P}}$ B, but any NP-hard problem can play the role of 3SAT. For example, the book shows that the undirected version of HamPath is NP-hard via a reduction from the directed version.

8 Space Complexity

The last chapter was about time complexity; this one is about space. If M is a deterministic TM that halts on all inputs, the *space complexity* of M is the function $f\colon \mathcal{N} \to \mathcal{N}$ where $f(n)$ is the maximum number of tape cells that M scans on any input of length n. If M is nondeterministic, $f(n)$ is the maximum number of tape cells M scans on any branch. We also have the classes SPACE($f(n)$), the collection of languages decided by an $O(f(n))$-space deterministic TM, and NSPACE($f(n)$), the collection of languages decided by an $O(f(n))$-space nondeterministic TM.

Example. Although nobody knows how to solve SAT in polynomial time, SAT is solvable in linear space by the brute-force algorithm: simply try every truth assignment. We can store any assignment in $O(m)$ space, where m is the number of variables in the input, so this TM requires $O(n)$ space, where n is the length of the input.

8.1 Savitch's Theorem

Nondeterministic TMs seem to have a large advantage over deterministic TMs with respect to time. Savitch's theorem, stated below, implies that the same does not hold with respect to space.

Theorem 8.1. *For every function f satisfying $f(n) \geq n$ for all n,* NSPACE($f(n)$) $\subseteq$ SPACE($f^2(n)$).

Proof (sketch). Let N be an $O(f(n))$-space NTM that decides a language A; our goal is to simulate N using a deterministic TM M. One natural approach is for M to try every branch in N's computation tree, one at a time. The problem is that a branch that uses $f(n)$ space may run for $2^{O(f(n))}$ steps, so tracking the information we'd need would require $2^{O(f(n))}$ space.

Instead, we can save space by using a recursive, divide-and-conquer approach. Basically, on input w, M determines if N accepts w by recursively searching for a path P from the starting configuration to an accepting configuration. To find P, M joins two shorter paths, and when computing those shorter paths, M can reuse its tape to save space. □

8.2 The Class PSPACE

The space analog of P is PSPACE $= \cup_k$SPACE(n^k). We can define NPSPACE analogously, but by Savitch's theorem, PSPACE = NPSPACE since the square of a polynomial is also a polynomial. Notice P $\subseteq$ PSPACE and NP $\subseteq$ NPSPACE because any TM can only explore a single new cell in each time step. Furthermore, an $f(n)$-space TM has at most $f(n)2^{O(f(n))}$ configurations, none of which can be repeated if the TM halts, so its running time is $f(n)2^{O(f(n))}$.[1] In summary, we know that

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} = \text{NPSPACE} \subseteq \text{EXPTIME},$$

[1]We must assume $f(n) \geq n$. Otherwise, for example, an $O(1)$-space TM could run for n steps. We'll revisit this assumption in Section 8.4.

and we don't know if any of the containments is actually an equality. In Chapter 9, we'll prove P $\neq$ EXPTIME so at least one of the containments is proper, and most researchers believe that all of them are proper.

8.3 PSPACE-completeness

The notion of completeness also applies to space complexity: a language B is *PSPACE-hard* if every problem in PSPACE is polynomial-time reducible to B, and B is *PSPACE-complete* if B is PSPACE-hard and $B \in$ PSPACE.

The TQBF Problem

The TQBF problem is a generalization of SAT. A *quantified Boolean formula* (QBF) is a Boolean formula that also can contain multiple quantifiers $\forall$ (for all) and $\exists$ (there exists). For example,

$$\phi = \forall x\, \exists y\, [(x \vee y) \wedge (\overline{x} \vee \overline{y})]$$

is a true QBF ($y = \overline{x}$ makes $\phi = 1$). It is also *fully quantified* because every variable appears within the scope of some quantifier. The TQBF problem is the following:

$$\mathsf{TQBF} = \{\langle \phi \rangle \mid \phi \text{ is a true fully quantified Boolean formula}\}.$$

Theorem 8.2. TQBF *is* PSPACE-*complete.*

Proof (sketch). We can decide TQBF in polynomial space by using a recursive brute-force algorithm; the space required is $O(m)$, where m is the number of variables in ϕ.

Now let A be a language decidable by an n^k-space TM M; we want to show $A \leq_{\mathrm{P}} \mathsf{TQBF}$. One idea is to use the approach we used to prove SAT is NP-hard (Theorem 7.7), but the size of the tableau could be exponential in n^k because M can run for exponential time. Instead, we can use a technique related to the divide-and-conquer approach used to prove Savitch's theorem (Theorem 8.1): construct ϕ by assembling several parts in a recursive manner. $\square$

Winning Strategies for Games

At this point, the book introduces a couple of games and shows that they are PSPACE-complete:

- **Formula game:** There is a quantified Boolean formula, and two players alternate assigning variables to values. Player 1's goal is to make the formula TRUE, while Player 2's goal is to make the formula FALSE. It turns out that deciding whether Player 1 has a winning strategy is equivalent to deciding TQBF.

- **Generalized Geography:** There is a directed graph G, and two players alternate picking vertices in G. The chosen vertices must collectively form a path in G (i.e., no vertex can be chosen more than once); the first player to get "stuck" loses. The book proves that this game is PSPACE-hard via a reduction from the formula game described above.

More familiar games, such as chess, checkers, and GO, typically involve only a finite number of states, so finding a winning strategy only requires a finite amount of resources. However, generalizations of these games have been studied, and they have been shown to be PSPACE-hard (or even harder, depending on the details of the generalization).

8.4 The Classes L and NL

So far, we have only considered TMs that use at least n space (where n is the input size) but we can also consider TMs that use less space. (TMs that use less than n time steps cannot even read the entire input.) Our computation model is now a TM with two tapes: the first is a read-only input tape, and the second is the usual read/write work tape. Only cells scanned on the work tape count towards the space complexity. This model captures the scenario in which the input is too large to be stored in our main memory. We'll focus on the following two classes: L = SPACE($\log n$), NL = NSPACE($\log n$).[2]

Example. Consider the language $\{\texttt{0}^k\texttt{1}^k \mid k \geq 0\}$: we can decide this language by checking that no `1` appears before a `0`, then counting the number of `0`s and `1`s. Each counter uses $O(\log n)$ space, so this language is in L.

Example. Recall the following language:

$$\textsf{PATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a path from } s \text{ to } t\}.$$

In Section 7.2, we showed that $\textsf{PATH} \in$ P since we can use breadth-first search, but this algorithm requires linear space. We don't know if $\textsf{PATH}$ is in L, but we can show that $\textsf{PATH} \in$ NL: an NTM can nondeterministically guess the vertices on the path from s to t while keeping track of just the current vertex. It accepts if it reaches t and rejects if hasn't reached t in m steps, where m is the number of vertices in G.

In Section 8.2, we claimed that an $f(n)$-space TM runs in time $f(n)2^{O(f(n))}$ but assumed $f(n) \geq n$. To address this assumption, we define a configuration of our new model to be the following: it is a setting of the state, the work tape, and the positions of the two tape heads. In particular, the input is not part of the configuration. Under this definition, an $f(n)$-space TM has $n2^{O(f(n))}$ configurations; the details are in the book. Savitch's theorem (Theorem 8.1) also holds if we assume $f(n) \geq \log n$ rather than $f(n) \geq n$, and the proof is similar.

8.5 NL-completeness

As mentioned in the previous section, we know that $\textsf{PATH}$ is in NL but we don't know if $\textsf{PATH}$ is in L. In fact, we don't know if there are any problems in NL $\setminus$ L (just as we don't know if there are any problems in NP $\setminus$ P).

[2]The reasons for choosing $\log n$ rather than, say, $\sqrt{n}$, are similar to the reasons for considering polynomial time when defining P and NP: practical considerations, mathematical properties, etc. One way to think of logarithmic space is to consider algorithms that have a fixed number of input pointers, since a pointer requires logarithmic space.

We can define NL-completeness the same way we did for NP-completeness: a problem B is NL-complete if $B \in$ NL and every problem $A \in$ NL reduces to B. However, the reduction from A to B must be a *log-space reduction*.[3] This means that the TM computing the reduction has three tapes: a read-only input tape, a write-only output tape, and a read/write work tape. The head on the write-only tape cannot move leftward, and the work tape may only contain $O(\log n)$ symbols. We use $A \leq_{\mathrm{L}} B$ to denote that A is log-space reducible to B.

Theorem 8.3. *If $A \leq_{\mathrm{L}} B$ and $B \in \mathrm{L}$, then $A \in \mathrm{L}$.*

Proof (sketch). Let f denote the reduction from A to B. We could mimic the proof of Theorem 7.5, but $f(w)$ might be too large to fit in the log space bound. Instead, A's TM M_A can run B's machine M_B on $f(w)$, and while M_A simulates M_B, it recomputes $f(w)$ only when necessary, and it only stores the symbol it needs. That is, whenever M_B takes a step, M_A computes $f(w)$ to find the symbol in $f(w)$ that M_B needs and discards the rest of $f(w)$. This approach allows M_A to reduce its space complexity by raising its time complexity. □

Just as 3SAT is NP-complete, PATH is NL-complete:

Theorem 8.4. PATH *is* NL*-complete.*

Proof (sketch). In the previous section, we showed PATH $\in$ NL, so it suffices to show $A \leq_{\mathrm{L}}$ PATH for any $A \in$ NL. The idea is fairly intuitive: given a TM M that decides A, we can build a graph G whose vertices represent the configurations of M on input w and edges represent valid steps that M can take. Determining if M accepts w is equivalent to finding a path from the vertex representing the start configuration to the vertex representing the accept configuration. □

Corollary 8.5. NL $\subseteq$ P.

Proof (sketch). Suppose $A \in$ NL; Theorem 8.4 implies that A is log-space reducible to PATH. As mentioned in Section 8.4, an $f(n)$-space TM has $n2^{O(f(n))}$ configurations; this implies that a log space reducer runs in polynomial time. Therefore, since we can solve PATH in polynomial time, we can also solve A in polynomial time. □

8.6 NL equals coNL

As mentioned in Section 7.3, people generally believe that NP $\neq$ coNP. But this does not hold for NL and coNL:

Theorem 8.6. NL $\neq$ coNL.

Proof (sketch). Recall that PATH is NL-complete (Theorem 8.4), which implies $\overline{\text{PATH}}$ is coNL-complete. It suffices to show $\overline{\text{PATH}} \in$ NL because that implies coNL $\subseteq$ NL, which implies coNL = NL.

[3] As we'll see, every problem in NL is solvable in polynomial time, so any two problems in NL (excluding $\emptyset$ and Σ^*) are polynomial-time reducible to each other.

First, suppose we know c, the number of vertices in G that are reachable from s. The TM M essentially does the following: nondeterministically guess whether each vertex u is reachable from s, and verify each guess for vertex u by nondeterministically finding a path from s to u. If M ever finds exactly c vertices reachable from s, and t is not one of these c vertices, M accepts. To calculate c, it uses a similar nondeterministic guessing procedure. $\square$

Let's incorporate L and NL into our summary:

$$\mathrm{L} \subseteq \mathrm{NL} = \mathrm{coNL} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} = \mathrm{NPSPACE} \subseteq \mathrm{EXPTIME}.$$

Again (see Section 8.2), we don't know if any of the containments are proper, but in Chapter 9, we'll prove NL $\subsetneq$ PSPACE.

9 Intractability

Recall that P $\subseteq$ NP, and some problems in NP seem intractable (e.g., 3SAT), but nobody has been able to prove this. In this chapter, we'll prove that some problems truly are harder than others.

9.1 Hierarchy Theorems

Space Hierarchy Theorem

A function $f\colon \mathcal{N} \to \mathcal{N}$ that is at least $O(\log n)$ is *space constructible* if the function that maps 1^n to the binary representation of $f(n)$ is computable in $O(f(n))$ space.[1] The following is the space hierarchy theorem:

Theorem 9.1. *For any space computable function f, there exists a language A that is decidable in $O(f(n))$ space but not $o(f(n))$ space.*

Proof (sketch). We'll give an algorithm D that describes the desired language A. In particular, D runs in $O(f(n))$ space and ensures that A is not decidable in $o(f(n))$ space. To design D, we'll use diagonalization (see Section 4.2): for any TM M, A will differ from M's language in the spot $\langle M\rangle$. That is, suppose D receives $\langle M\rangle$ as input. (If the input does not describe a TM, D simply rejects.) Then D runs M on $\langle M\rangle$ within $f(n)$ space, and D accepts if and only if M rejects. There are some missing details, but this basically ensures that D's language is different from M's. □

As a corollary, for any two natural numbers $c_1 < c_2$, we have SPACE(n^{c_1}) $\subsetneq$ SPACE(n^{c_2}). In fact, this statement holds even if c_1 and c_2 are real numbers (and $c_1 \geq 0$). We can also obtain a separation result between two complexity classes we've seen:

Corollary 9.2. NL $\subsetneq$ PSPACE.

Proof. By Savitch's theorem (Theorem 8.1), NL $\subseteq$ SPACE($\log^2 n$), and Theorem 9.1 implies SPACE($\log^2 n$) $\subsetneq$ SPACE(n). □

Since TQBF is PSPACE-complete with respect to log space reducibility, the corollary above implies TQBF $\notin$ NL. The space hierarchy theorem also implies separation between PSPACE and EXPSPACE $= \cup_k$SPACE(2^{n^k}).

Time Hierarchy Theorem

The time hierarchy theorem is slightly weaker (by a logarithmic factor) than its space analog from earlier in this section. We still start with a technical definition: a function $t : \mathcal{N} \to \mathcal{N}$ that is at least $O(n \log n)$ is *time constructible* if the function that maps 1^n to the binary representation of $t(n)$ is computable in $O(t(n))$ time.

Theorem 9.3. *For any time constructible function t, there exists a language A that is decidable in $O(t(n))$ time but not in $o(t(n)/logt(n))$ time.*

[1] As was the case with computable functions, we're working with space constructible functions for rather technical reasons that are not essential to fully understand.

Proof (sketch). The proof is similar to that of the space hierachy theorem: we'll design an $O(t(n))$-time algorithm D that decides the desired language A using the diagonalization method. Keeping track of the number of steps that D has spent results in the logarithmic factor overhead. $\square$

From the time hierarchy theorem, we can conclude $\text{TIME}(n^{c_1}) \subsetneq \text{TIME}(n^{c_2})$ for any real numbers $1 \leq c_1 < c_2$, and $\text{P} \subsetneq \text{EXPTIME}$.

Exponential Space Completeness

[unfinished]

9.2 Relativization

[unfinished]

9.3 Circuit Complexity

[unfinished]

10 Advanced Topics in Complexity Theory

[unfinished]

10.1 Approximation Algorithms

[unfinished]

10.2 Probabilistic Algorithms

[unfinished]

10.3 Alternation

[unfinished]

10.4 Interactive Proof Systems

[unfinished]

10.5 Parallel Computation

[unfinished]

10.6 Cryptography

[unfinished]