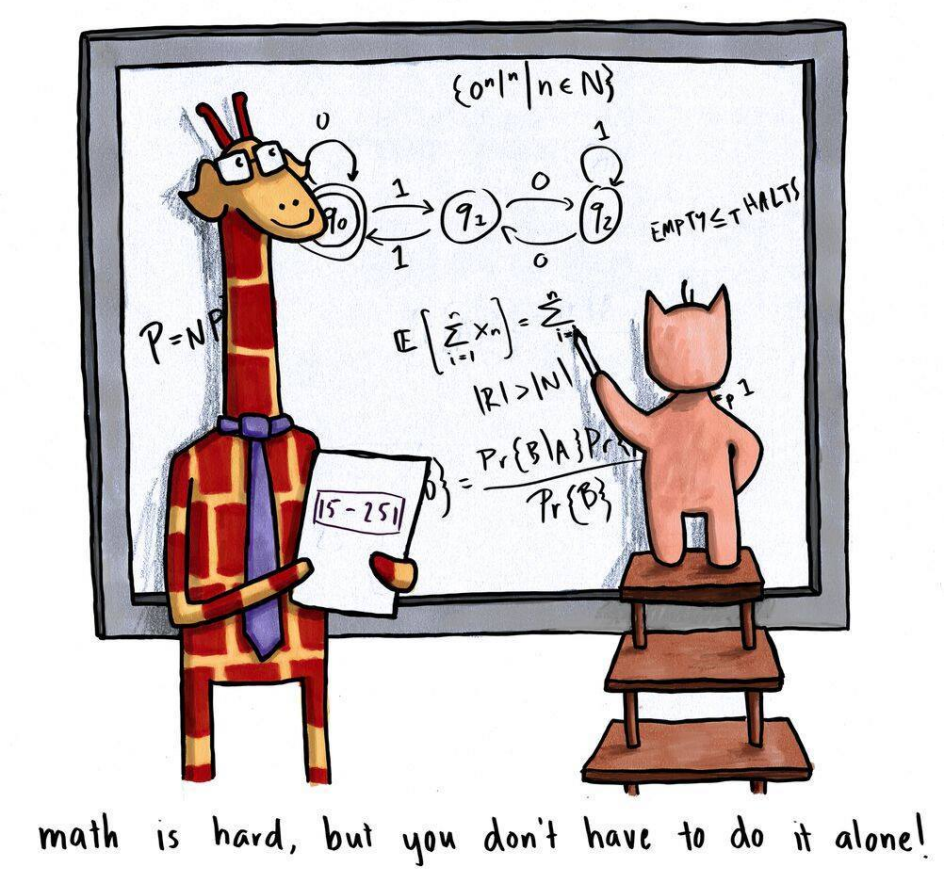
CMU 15-251, Spring 2018
Great Ideas in Theoretical Computer Science

Course Notes: Main File



math is hard, but you don't have to do it alone!

April 4, 2018

Please send comments and corrections to Anil Ada (aada@cs.cmu.edu).

Foreword

These notes are based on the lectures given by Anil Ada and Klaus Sutner for the Spring 2018 edition of the course 15-251 "Great Ideas in Theoretical Computer Science" at Carnegie Mellon University. They are also closely related to the previous editions of the course, and in particular, lectures prepared by Ryan O'Donnell.

WARNING: The purpose of these notes is to complement the lectures. These notes do *not* contain full explanations of all the material covered during lectures. In particular, the intuition and motivation behind many concepts and proofs are explained during the lectures and not in these notes.

There are various versions of the notes that omit certain parts of the notes. Go to the course webpage to access all the available versions.

In the main version of the notes (i.e. the main document), each chapter has a preamble containing the chapter structure and the learning goals. The preamble may also contain some links to concrete applications of the topics being covered. At the end of each chapter, you will find a short quiz for you to complete before coming to recitation, as well as hints to selected exercise problems.

Note that some of the exercise solutions are given in full detail, whereas for others, we give all the main ideas, but not all the details. We hope the distinction will be clear.

Acknowledgements

The course 15-251 was created by Steven Rudich many years ago, and we thank him for creating this awesome course. Here is the webpage of an early version of the course:
http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15251-s04/Site/.
Since then, the course has evolved. The webpage of the current version is here:
http://www.cs.cmu.edu/~15251/.

Thanks to the previous and current instructors of 15-251, who have contributed a lot to the development of the course: Victor Adamchik, Luis von Ahn, Anupam Gupta, Venkatesan Guruswami, Bernhard Haeupler, John Lafferty, Ryan O'Donnell, Ariel Procaccia, Daniel Sleator and Klaus Sutner.

Thanks to Eric Bae, Apoorva Bhagwat, George Cai, Darshan Chakrabarti, Seth Cobb, Teddy Ding, Emilie Guermeur, Ellen Kim, Aditya Krishnan, Xinran Liu, Udit Ranasaria, Matthew Salim, Ticha Sethapakdi, Vanessa Siriwalothakul, Rosie Sun, Natasha Vasthare, Jenny Wang, Ling Xu, Ming Yang, Wynne Yao, Stephanie You, Xingjian Yu and Nancy Zhang for sending valuable comments and corrections on an earlier draft of the notes. And thanks to Parmita Bawankule, Dominic Calkosz and Deborah Chu for sending valuable comments and corrections on the current draft.

Special thanks go to Ji An Yang, a former teaching assistant for 15-251, for writing the solutions to the exercise problems in the chapter on probability theory.

Contents

Chapter 1

Strings and Encodings

PREAMBLE

Chapter structure:

- Section 1.1 (Alphabets and Strings)
  - Definition 1.1 (Alphabet, symbol/character)
  - Definition 1.5 (String/word, empty string)
  - Definition 1.9 (Length of a string)
  - Definition 1.11 (Star operation on alphabets)
  - Definition 1.16 (Reversal of a string)
  - Definition 1.20 (Concatenation of strings)
  - Definition 1.24 (Powers of a string)
  - Definition 1.27 (Substring)
- Section 1.2 (Languages)
  - Definition 1.29 (Language)
  - Definition 1.37 (Reversal of a language)
  - Definition 1.39 (Concatenation of languages)
  - Definition 1.41 (Powers of a language)
  - Definition 1.45 (Star operation on a language)
- Section 1.3 (Encodings)
  - Definition 1.51 (Encoding of a set)
- Section 1.4 (Computational Problems and Decision Problems)
  - Definition 1.61 (Computational problem)
  - Definition 1.64 (Decision problem)

Chapter goals:

In the beginning, our goal is to build up, completely formally/mathematically, the important notions related to computation and algorithms. Our starting point is this chapter, which deals with how to formally represent data and how to formally define the concept of a computational problem.

In theoretical computer science, every kind of data is represented/encoded using finite-length strings. In this chapter, we introduce you to the formal definitions related to strings and encodings of objects with strings. We also present the definitions of "computational problem" and "decision problem".

All the definitions in this chapter are at the foundation of the formal study of computation.

1.1 Alphabets and Strings

Definition 1.1 (Alphabet, symbol/character)**.**
An *alphabet* is a non-empty, finite set, and is usually denoted by Σ. The elements of Σ are called *symbols* or *characters*.

Example 1.2 (Unary alphabet)**.**
A unary alphabet consists of one symbol. A common choice for that symbol is $\texttt{1}$. So an example of a unary alphabet is $\Sigma = \{\texttt{1}\}$.

Example 1.3 (Binary alphabet)**.**
A binary alphabet consists of two symbols. Often we represent those symbols using $\texttt{0}$ and $\texttt{1}$. So an example of a binary alphabet is $\Sigma = \{\texttt{0}, \texttt{1}\}$. Another example of a binary alphabet is $\Sigma = \{\texttt{a}, \texttt{b}\}$ where $\texttt{a}$ and $\texttt{b}$ are the symbols.

Example 1.4 (Ternary alphabet)**.**
A ternary alphabet consists of three symbols. So $\Sigma = \{\texttt{0}, \texttt{1}, \texttt{2}\}$ and $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ are examples of ternary alphabets.

Definition 1.5 (String/word, empty string)**.**
Given an alphabet Σ, a *string* (or *word*) over Σ is a (possibly infinite) sequence of symbols, written as $a_1a_2a_3\ldots$, where each $a_i \in \Sigma$. The string with no symbols is called the *empty string* and is denoted by ϵ.

Example 1.6 (Strings over the unary alphabet)**.**
For $\Sigma = \{\texttt{1}\}$, the following is a list of 6 strings over Σ:

$$\epsilon,\ \texttt{1},\ \texttt{11},\ \texttt{111},\ \texttt{1111},\ \texttt{11111}.$$

Furthermore, the infinite sequence $\texttt{111111}\ldots$ is also a string over Σ.

Example 1.7 (Strings over the binary alphabet)**.**
For $\Sigma = \{\texttt{0}, \texttt{1}\}$, the following is a list of 8 strings over Σ:

$$\epsilon,\ \texttt{0},\ \texttt{1},\ \texttt{00},\ \texttt{01},\ \texttt{10},\ \texttt{11},\ \texttt{000}.$$

The infinite strings $\texttt{000000}\ldots$, $\texttt{111111}\ldots$ and $\texttt{010101}\ldots$ are also examples of strings over Σ.

Note 1.8 (Strings and quotation marks)**.**
In our notation of a string, we do not use quotation marks. For instance, we use the notation $\texttt{1010}$ rather than "$\texttt{1010}$", even though the latter notation using the quotation marks is the standard one in many programming languages. Occasionally, however, we may use quotation marks to distinguish a string like "$\texttt{1010}$" from another type of object with the representation 1010 (e.g. the binary *number* 1010).

Definition 1.9 (Length of a string)**.**
The *length* of a string w, denoted $|w|$, is the the number of symbols in w. If w has an infinite number of symbols, then the length is undefined.

Example 1.10 (Lengths of 01001 and ϵ)**.**
Let $\Sigma = \{\texttt{0}, \texttt{1}\}$. The length of the word $\texttt{01001}$, denoted by $|\texttt{01001}|$, is equal to 5. The length of ϵ is 0.

Definition 1.11 (Star operation on alphabets)**.**
Let Σ be an alphabet. We denote by Σ^* the set of *all* strings over Σ consisting of finitely many symbols. Equivalently, using set notation,

$$\Sigma^* = \{a_1 a_2 \dots a_n : \ n \in \mathbb{N}, \text{and } a_i \in \Sigma \text{ for all } i\}.$$

Example 1.12 ($\{a\}^*$)**.**
For $\Sigma = \{\texttt{a}\}$, Σ^* denotes the set of all finite-length words consisting of a's. So

$$\{\texttt{a}\}^* = \{\epsilon, \texttt{a}, \texttt{aa}, \texttt{aaa}, \texttt{aaaa}, \texttt{aaaaa}, \dots\}.$$

Example 1.13 ($\{0,1\}^*$)**.**
For $\Sigma = \{\texttt{0}, \texttt{1}\}$, Σ^* denotes the set of all finite-length words consisting of 0's and 1's. So

$$\{\texttt{0}, \texttt{1}\}^* = \{\epsilon, \texttt{0}, \texttt{1}, \texttt{00}, \texttt{01}, \texttt{10}, \texttt{11}, \texttt{000}, \texttt{001}, \texttt{010}, \texttt{011}, \texttt{100}, \texttt{101}, \texttt{110}, \texttt{111}, \dots\}.$$

Note 1.14 (Finite vs infinite strings)**.**
We often use the words "string" and "word" to refer to a finite-length string/word. When we want to talk about infinite-length strings, we explicitly use the word "infinite".

Note 1.15 (Size of Σ^*)**.**
By Definition 1.1 (Alphabet, symbol/character), an alphabet Σ cannot be the empty set. This implies that Σ^* is an infinite set since there are infinitely many strings of finite length over a non-empty Σ.

Definition 1.16 (Reversal of a string)**.**
For a string $w = a_1 a_2 \dots a_n$, the *reversal* of w, denoted w^R, is the string $w^R = a_n a_{n-1} \dots a_1$.

Example 1.17 (Reversal of 01001)**.**
The reversal of $\texttt{01001}$ is $\texttt{10010}$.

Example 1.18 (Reversal of 1)**.**
The reversal of $\texttt{1}$ is $\texttt{1}$.

Example 1.19 (Reversal of ϵ)**.**
The reversal of ϵ is ϵ.

Definition 1.20 (Concatenation of strings)**.**
If u and v are two strings in Σ^*, the *concatenation* of u and v, denoted by uv or $u \cdot v$, is the string obtained by joining together u and v.

Example 1.21 (Concatenation of 101 and 001)**.**
If $u = \texttt{101}$ and $v = \texttt{001}$, then $uv = \texttt{101001}$.

Example 1.22 (Concatenation of 101 and ϵ)**.**
If $u = \texttt{101}$ and $v = \epsilon$, then $uv = \texttt{101}$.

Example 1.23 (Concatenation of ϵ and ϵ)**.**
If $u = \epsilon$ and $v = \epsilon$, then $uv = \epsilon$.

Definition 1.24 (Powers of a string)**.**
For a word $u \in \Sigma^*$ and $n \in \mathbb{N}$, the n'th *power* of u, denoted by u^n, is the word obtained by concatenating u with itself n times.

Example 1.25 (Third power of 101)**.**
If $u = 101$ then $u^3 = 101101101$.

Example 1.26 (Zeroth power of a string)**.**
For any string u, $u^0 = \epsilon$.

Definition 1.27 (Substring)**.**
We say that a string u is a *substring* of string w if $w = xuy$ for some strings x and y.

Example 1.28 (101 as a substring)**.**
The string 101 is a substring of 11011 and also a substring of 0101. On the other hand, it is not a substring of 1001.

1.2 Languages

Definition 1.29 (Language)**.**
Any (possibly infinite) subset $L \subseteq \Sigma^*$ is called a *language* over the alphabet Σ.

Example 1.30 (Language of even length strings)**.**
Let Σ be an alphabet. Then $L = \{w \in \Sigma^* : |w| \text{ is even}\}$ is a language.

Example 1.31 (A language with one word)**.**
Let $\Sigma = \{0, 1\}$. Then $L = \{101\}$ is a language.

Example 1.32 (Σ^* as a language)**.**
Let Σ be an alphabet. Then $L = \Sigma^*$ is a language.

Example 1.33 (Empty set as a language)**.**
Let Σ be an alphabet. Then $L = \varnothing$ is a language.

Note 1.34 (Size of a language)**.**
Since a language is a set, the *size of a language* refers to the size of that set. A language can have finite or infinite size. This is not in conflict with the fact that every language consists of finite-length strings.

Note 1.35 ($\varnothing$ vs $\{\epsilon\}$)**.**
The language $\{\epsilon\}$ is not the same language as $\varnothing$. The former has size 1 whereas the latter has size 0.

Exercise 1.36 (Structural induction on words)**.**
Let language $L \subseteq \{0, 1\}^*$ be recursively defined as follows:

- $\epsilon \in L$;
- if $x, y \in L$, then $0x1y0 \in L$.

Show, using (structural) induction, that for any word $w \in L$, the number of 0's in w is exactly twice the number of 1's in w.

Definition 1.37 (Reversal of a language)**.**
Given a language $L \subseteq \Sigma^*$, we define its *reversal*, denoted L^R, as the language

$$L^R = \{w^R \in \Sigma^* : w \in L\}.$$

Example 1.38 (Reversal of $\{\epsilon, 1, 1010\}$)**.**
The reversal of the language $\{\epsilon, 1, 1010\}$ is $\{\epsilon, 1, 0101\}$.

Definition 1.39 (Concatenation of languages)**.**
Given two languages $L_1, L_2 \subseteq \Sigma^*$, we define their *concatenation*, denoted L_1L_2 or $L_1 \cdot L_2$, as the language

$$L_1L_2 = \{uv \in \Sigma^* : u \in L_1, v \in L_2\}.$$

Example 1.40 (Concatenation of $\{\epsilon, 1\}$ and $\{0, 01\}$)**.**
The concatenation of languages $\{\epsilon, 1\}$ and $\{0, 01\}$ is the language

$$\{0, 01, 10, 101\}.$$

Definition 1.41 (Powers of a language)**.**
Given a language $L \subseteq \Sigma^*$ and $n \in \mathbb{N}$, the n'th power of L, denoted L^n, is the language obtained by concatenating L with itself n times, that is,[1]

$$L^n = \underbrace{L \cdot L \cdot L \cdots L}_{n \text{ times}}.$$

Equivalently,

$$L^n = \{u_1u_2 \cdots u_n \in \Sigma^* : u_i \in L \text{ for all } i \in \{1, 2, \ldots, n\}\}.$$

Example 1.42 ($\{1\}^3$)**.**
The 3rd power of $\{1\}$ is the language $\{111\}$.

Example 1.43 ($\{\epsilon, 1\}^3$)**.**
The 3rd power of $\{\epsilon, 1\}$ is the language $\{\epsilon, 1, 11, 111\}$.

Example 1.44 (L^0)**.**
The 0th power of any language L is the language $\{\epsilon\}$.

Definition 1.45 (Star operation on a language)**.**
Given a language $L \subseteq \Sigma^*$, we define the *star* of L, denoted L^*, as the language

$$L^* = \bigcup_{n \in \mathbb{N}} L^n.$$

Equivalently,

$$L^* = \{u_1u_2 \cdots u_n \in \Sigma^* : n \in \mathbb{N}, u_i \in L \text{ for all } i \in \{1, 2, \ldots, n\}\}.$$

Example 1.46 (Σ^*)**.**
Given an alphabet Σ, consider the language $L = \Sigma \subseteq \Sigma^*$. Then L^* is equal to Σ^*.

[1] We can omit parentheses as the order in which the concatenation $\cdot$ is applied does not matter.

Example 1.47 ($\{00\}^*$)**.**
If $L = \{00\}$, then L^* is the language consisting of all words containing an even number of 0's and no other symbol.

Example 1.48 ($(\{00\}^*)^*$)**.**
Let L be the language consisting of all words containing an even number of 0's and no other symbol. Then $L^* = L$.

Exercise 1.49 (Can you distribute star over intersection?)**.**
Prove or disprove: If $L_1, L_2 \subseteq \{a, b\}^*$ are languages, then $(L_1 \cap L_2)^* = L_1^* \cap L_2^*$.

Exercise 1.50 (Can you interchange star and reversal?)**.**
Is it true that for any language L, $(L^*)^R = (L^R)^*$? Prove your answer.

1.3 Encodings

Definition 1.51 (Encoding of a set)**.**
Let A be a set (which is possibly countably infinite[2]), and let Σ be a alphabet. An *encoding* of the elements of A, using Σ, is an injective function $\text{Enc} : A \to \Sigma^*$. We denote the encoding of $a \in A$ by $\langle a \rangle$.[3]

If $w \in \Sigma^*$ is such that there is some $a \in A$ with $w = \langle a \rangle$, then we say w is a *valid encoding* of an element in A.

A set that can be encoded is called *encodable*.[4]

Example 1.52 (Decimal encoding of naturals)**.**
When we (humans) communicate numbers among ourselves, we usually use the base-10 representation, which corresponds to an encoding of $\mathbb{N}$ using the alphabet $\Sigma = \{0, 1, 2, \ldots, 9\}$. For example, we encode the number four as 4 and the number twelve as 12.

Example 1.53 (Binary encoding of naturals)**.**
As you know, every number has a base-2 representation (which is also known as the binary representation). This representation corresponds to an encoding of $\mathbb{N}$ using the alphabet $\Sigma = \{0, 1\}$. For example, four is encoded as 100 and twelve is encoded as 1100.

Example 1.54 (Binary encoding of integers)**.**
An integer is a natural number together with a sign, which is either negative or positive. Let $\text{Enc} : \mathbb{N} \to \{0, 1\}^*$ be any binary encoding of $\mathbb{N}$. Then we can extend this encoding to an encoding of $\mathbb{Z}$, by defining $\text{Enc}' : \mathbb{Z} \to \{0, 1\}^*$ as follows:

$$\text{Enc}'(x) = \begin{cases} 0\text{Enc}(x) & \text{if } x \geq 0, \\ 1\text{Enc}(|x|) & \text{if } x < 0. \end{cases}$$

Effectively, this encoding of integers takes the encoding of natural numbers and precedes it with a bit indicating the integer's sign.

[2] We assume you know what a countable set is, however, we will review this concept in a future lecture.

[3] Note that this angle-bracket notation does not specify the underlying encoding function as the particular choice of encoding function is often unimportant.

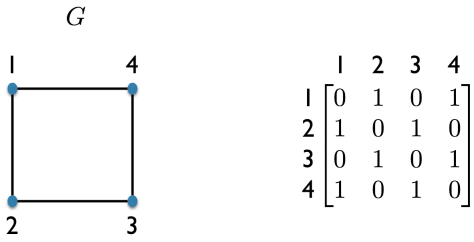
[4] Not every set is encodable. Can you figure out exactly which sets are encodable?

Example 1.55 (Unary encoding of naturals)**.**
It is possible (and straightforward) to encode the natural numbers using the alphabet $\Sigma = \{1\}$ as follows. Let $\text{Enc}(n) = 1^n$ for all $n \in \mathbb{N}$.

Example 1.56 (Ternary encoding of pairs of naturals)**.**
Suppose we want to encode the set $A = \mathbb{N} \times \mathbb{N}$ using the alphabet $\Sigma = \{0, 1, 2\}$. One way to accomplish this is to make use of a binary encoding $\text{Enc}' : \mathbb{N} \to \{0, 1\}^*$ of the natural numbers. With Enc' in hand, we can define $\text{Enc} : \mathbb{N} \times \mathbb{N} \to \{0, 1, 2\}^*$ as follows. For $(x, y) \in \mathbb{N} \times \mathbb{N}$, $\text{Enc}(x, y) = \text{Enc}'(x)2\text{Enc}'(y)$. Here the symbol 2 acts as a separator between the two numbers. To make the separator symbol advertise itself as such, we usually pick a symbol like # rather than 2. So the ternary alphabet is often chosen to be $\Sigma = \{0, 1, \#\}$.

Example 1.57 (Binary encoding of pairs of naturals)**.**
Having a ternary alphabet to encode pairs of naturals was convenient since we could use the third symbol as a separator. It is also relatively straightforward to take that ternary encoding and turn it into a binary encoding, as follows. Encode every element of the ternary alphabet in binary using two bits. For instance, if the ternary alphabet is $\Sigma = \{0, 1, \#\}$, then we could encode 0 as 00, 1 as 01 and # as 11. This mapping allows us to convert any encoded string over the ternary alphabet into a binary encoding. For example, a string like #0#1 would have the binary representation 11001101.

Example 1.58 (Ternary encoding of graphs)**.**
Let A be the set of all undirected graphs.[5] Every graph $G = (V, E)$ can be represented by its $|V|$ by $|V|$ adjacency matrix. In this matrix, every row corresponds to a vertex of the graph, and similarly, every column corresponds to a vertex of the graph. The (i, j)'th entry contains a 1 if $\{i, j\}$ is an edge, and contains a 0 otherwise. Below is an example.



	1	2	3	4
1	0	1	0	1
2	1	0	1	0
3	0	1	0	1
4	1	0	1	0

Such a graph can be encoded using a ternary alphabet as follows. Take the adjacency matrix of the graph, view each row as a binary string, and concatenate all the rows by putting a separator symbol between them. The encoding of the above example would be

$$\langle G \rangle = \texttt{0101\#1010\#0101\#1010}.$$

Example 1.59 (Encoding of Python functions)**.**
Let A be the set of all functions in the programming language Python. Whenever we type up a Python function in a code editor, we are creating a string representation/encoding of the function, where the alphabet is all the Unicode symbols.[6] For example, consider a Python function named absValue, which we can write as

```
def absValue(N):
    if (N < 0): return -N
    else: return N
```

[5] We will define graphs formally in a future chapter, however, we assume you are already familiar with the concept.

[6] https://en.wikipedia.org/wiki/Unicode

By writing out the function, we have already encoded it. More specifically, ⟨absValue⟩ is the string

```
def absValue(N):\n    if (N < 0): return -N\n    else: return N
```

Exercise 1.60 (Unary encoding of integers)**.**
Describe an encoding of $\mathbb{Z}$ using the alphabet $\Sigma = \{1\}$.

1.4 Computational Problems and Decision Problems

Definition 1.61 (Computational problem)**.**
Let Σ be an alphabet. Any function $f : \Sigma^* \to \Sigma^*$ is called a *computational problem* over the alphabet Σ.

Example 1.62 (Addition as a computational problem)**.**
Consider the function $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ defined as $g(x, y) = x + y$. This is a function that expresses the addition problem in naturals. We can view g as a computational problem over an alphabet Σ once we fix an encoding of the domain $\mathbb{N} \times \mathbb{N}$ using Σ and an encoding of the codomain $\mathbb{N}$ using Σ. For convenience, we take $\Sigma = \{0, 1, \#\}$. Let Enc be the encoding of $\mathbb{N} \times \mathbb{N}$ as described in Example 1.56 (Ternary encoding of pairs of naturals). Let Enc′ be the encoding of $\mathbb{N}$ as described in Example 1.53 (Binary encoding of naturals). Note that Enc′ leaves the symbol # unused in the encoding. We now define the computational problem f corresponding to g. If $w \in \Sigma^*$ is a word that corresponds to a valid encoding of a pair of numbers (x, y) (i.e., $\text{Enc}(x, y) = w$), then define $f(w)$ to be $\text{Enc}'(x + y)$. If $w \in \Sigma^*$ is *not* a word that corresponds to a valid encoding of a pair of numbers (i.e., w is not in the image of Enc), then define $f(w)$ to be #. In the codomain, the # symbol serves as an "error" indicator.

IMPORTANT 1.63 (Computational problem as mapping instances to solutions)**.**
A computational problem is often derived from a function $g : I \to S$, where I is a set of objects called *instances* and S is a set of objects called *solutions*. The derivation is done through encodings $\text{Enc} : I \to \Sigma^*$ and $\text{Enc}' : S \to \Sigma^*$. With these encodings, we can create the computational problem $f : \Sigma^* \to \Sigma^*$. In particular, if $w = \langle x \rangle$ for some $x \in I$, then we define $f(w)$ to be $\text{Enc}'(g(x))$.

$$\begin{array}{ccc} I & \xrightarrow{g} & S \\ {\scriptstyle \text{Enc}}\downarrow & & \downarrow{\scriptstyle \text{Enc}'} \\ \Sigma^* & \xrightarrow{f} & \Sigma^* \end{array}$$

One thing we have to be careful about is defining $f(w)$ for a word $w \in \Sigma^*$ that does not correspond to an encoding of an object in I (such a word does not correspond to an instance of the computational problem). To handle this, we can identify one of the strings in Σ^* as an *error* string and define $f(w)$ to be that string.

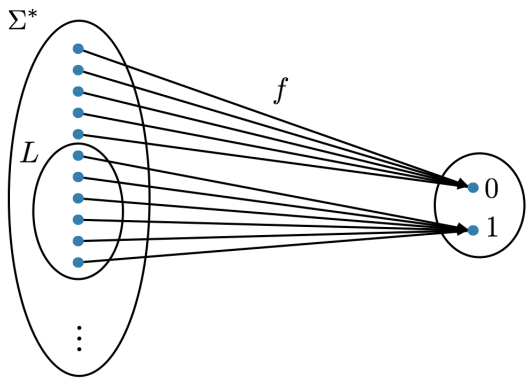
Definition 1.64 (Decision problem)**.**
Let Σ be an alphabet. Any function $f : \Sigma^* \to \{0, 1\}$ is called a *decision problem* over the alphabet Σ. The codomain of the function is not important as long as it has two elements. Other common choices for the codomain are $\{\text{No}, \text{Yes}\}$, $\{\text{False}, \text{True}\}$ and $\{\text{Reject}, \text{Accept}\}$.

Example 1.65 (Primality testing as a decision problem)**.**
Consider the function $g : \mathbb{N} \to \{\text{False}, \text{True}\}$ such that $g(x) = \text{True}$ if and only if x is a prime number. We can view g as a decision problem over an alphabet Σ once we fix an encoding of the domain $\mathbb{N}$ using Σ. Take $\Sigma = \{0, 1\}$. Let Enc be the encoding of $\mathbb{N}$ as described in Example 1.53 (Binary encoding of naturals). We now define the decision problem f corresponding to g. If $w \in \Sigma^*$ is a word that corresponds to an encoding of a prime number, then define $f(w)$ to be True. Otherwise, define $f(w)$ to be False. (Note that in the case of $f(w) = \text{False}$, either w is the encoding of a composite number, or w is not a valid encoding of a natural number.

Note 1.66 (Decision problem as mapping instances to 0 or 1s)**.**
As with a computational problem, a decision problem is often derived from a function $g : I \to \{0, 1\}$, where I is a set of instances. The derivation is done through an encoding $\text{Enc} : I \to \Sigma^*$, which allows us to define the decision problem $f : \Sigma^* \to \{0, 1\}$. Any word $w \in \Sigma^*$ that does not correspond to an encoding of an instance is mapped to 0 by f.

IMPORTANT 1.67 (Correspondence between decision problems and languages)**.**

There is a one-to-one correspondence between decision problems and languages. Let $f : \Sigma^* \to \{0, 1\}$ be some decision problem. Now define $L \subseteq \Sigma^*$ to be the set of all words in Σ^* that f maps to 1. This L is the language corresponding to the decision problem f. Similarly, if you take any language $L \subseteq \Sigma^*$, we can define the corresponding decision problem $f : \Sigma^* \to \{0, 1\}$ as $f(w) = 1$ if and only if $w \in L$. We consider the set of languages and the set of decision problems to be the same set of objects.



Quiz

1. Let L be the set of all strings of length at most 3 over the alphabet $\{\texttt{a}, \texttt{b}, \texttt{c}\}$. What is $|L|$?
2. Let Σ be an alphabet. For which languages $L \subseteq \Sigma^*$ is it true that $L \cdot L$ is infinite?
3. Let Σ be an alphabet. For which languages $L \subseteq \Sigma^*$ is it true that L^* is infinite?
4. True or false: The set of real numbers is encodable.
5. Consider the following problem. The input is an array A of n integers together with a target integer t. The output is a subset $S \subseteq \{0, 1, \ldots, n-1\}$ such that $\sum_{i \in S} A[i] = t$. If no such subset exists, the output is None. Formulate this as a computational problem.

Hints to Selected Exercises

Exercise 1.49 (Can you distribute star over intersection?):
Disprove the statement by providing a counterexample.

Exercise 1.50 (Can you interchange star and reversal?):
Show $(L^*)^R = (L^R)^*$. To do this, you need to argue both $(L^*)^R \subseteq (L^R)^*$ and $(L^R)^* \subseteq (L^*)^R$.

Chapter 2

Deterministic Finite Automata

PREAMBLE

Chapter structure:

- Section 14.1.1 (Basic Definitions)
  - Definition 2.1 (Deterministic Finite Automaton (DFA))
  - Definition 2.3 (Computation path for a DFA)
  - Definition 2.5 (A DFA accepting a string)
  - Definition 2.7 (Extended transition function)
  - Definition 2.9 (Language recognized/accepted by a DFA)
  - Definition 2.14 (Regular language)
- Section 2.2 (Irregular Languages)
  - Theorem 2.17 (0^n1^n is not regular)
  - Theorem 2.18 (A unary non-regular language)
- Section 2.3 (Closure Properties of Regular Languages)
  - Theorem 2.23 (Regular languages are closed under union)
  - Corollary 2.25 (Regular languages are closed under intersection)
  - Theorem 2.30 (Regular languages are closed under concatenation)

Chapter goals:

The goal of this chapter is to introduce you to a simple (and restricted) model of computation known as *deterministic finite automata*. This model is interesting to study in its own right, and has very nice applications, however, our main motivation to study this model is to use it as a stepping stone towards formally defining the notion of an *algorithm* in its full generality. Treating deterministic finite automata as a warm-up, we would like you to get comfortable with how one formally defines a model of computation, and then proves interesting theorems related to the model. Along the way, you will start getting comfortable with using a bit more sophisticated mathematical notation than you might be used to. You will see how mathematical notation helps us express ideas and concepts accurately, succinctly and clearly.

Applications:

- https://cstheory.stackexchange.com/questions/8539/how-practical-is-automata-theory
- http://cap.virginia.edu

2.1 Basic Definitions

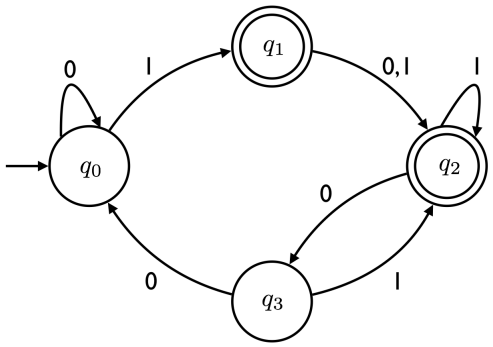
Definition 2.1 (Deterministic Finite Automaton (DFA))**.**
A *deterministic finite automaton* (DFA) M is a 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a non-empty finite set
  (which we refer to as the *set of states*);
- Σ is a non-empty finite set
  (which we refer to as the *alphabet* of the DFA);
- δ is a function of the form $\delta : Q \times \Sigma \to Q$
  (which we refer to as the *transition function*);
- $q_0 \in Q$ is an element of Q
  (which we refer to as the *start state*);
- $F \subseteq Q$ is a subset of Q
  (which we refer to as the *set of accepting states*).

Example 2.2 (A 4-state DFA)**.**
Below is an example of how we draw a DFA:



In this example, $\Sigma = \{0, 1\}$, $Q = \{q_0, q_1, q_2, q_3\}$, $F = \{q_1, q_2\}$. The labeled arrows between the states encode the transition function δ, which can also be represented with a table as below (row $q_i \in Q$ and column $b \in \Sigma$ contains $\delta(q_i, b)$).

	0	1
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_3	q_2
q_3	q_0	q_2

Definition 2.3 (Computation path for a DFA)**.**
Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $w = w_1 w_2 \cdots w_n$ be a string over an alphabet Σ (so $w_i \in \Sigma$ for each $i \in \{1, 2, \ldots, n\}$). Then the *computation path* of M with respect to w is a sequence of states

$$r_0, r_1, r_2, \ldots, r_n,$$

where each $r_i \in Q$, and such that

- $r_0 = q_0$;
- $\delta(r_{i-1}, w_i) = r_i$ for each $i \in \{1, 2, \ldots, n\}$.

We say that the computation path is *accepting* if $r_n \in F$, and *rejecting* otherwise.

Example 2.4 (An example of a computation path)**.**
Let $M = (Q, \Sigma, \delta, q_0, F)$ be the DFA in Example 2.2 (A 4-state DFA) and let $w = 110110$. Then the computation path of M with respect to w is

$$q_0, q_1, q_2, q_3, q_2, q_2, q_3.$$

Since q_3 is not in F, this is a rejecting computation path.

Definition 2.5 (A DFA accepting a string)**.**
We say that DFA $M = (Q, \Sigma, \delta, q_0, F)$ *accepts* a word $w \in \Sigma^*$ if the computation path of M with respect to w is an accepting computation path. Otherwise, we say that M *rejects* the string w.

Example 2.6 (An example of a DFA accepting a string)**.**
Let $M = (Q, \Sigma, \delta, q_0, F)$ be the DFA in Example 2.2 (A 4-state DFA) and let $w = 01101$. Then the computation path of M with respect to w is

$$q_0, q_0, q_1, q_2, q_3, q_2.$$

This is an accepting computation path because the sequence ends with q_2, which is in F. Therefore M accepts w.

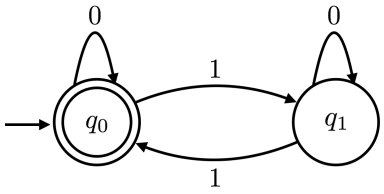
Definition 2.7 (Extended transition function)**.**
Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. The transition function $\delta : Q \times \Sigma \to Q$ can be extended to $\delta^* : Q \times \Sigma^* \to Q$, where $\delta^*(q, w)$ is defined as the state we end up in if we start at q and read the string w. In fact, often the star in the notation is dropped and δ is overloaded to represent both a function $\delta : Q \times \Sigma \to Q$ and a function $\delta : Q \times \Sigma^* \to Q$.

Note 2.8 (Alternative definition of a DFA accepting a string)**.**
Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Using the notation above, we can say that a word w is *accepted* by the DFA M if $\delta(q_0, w) \in F$.

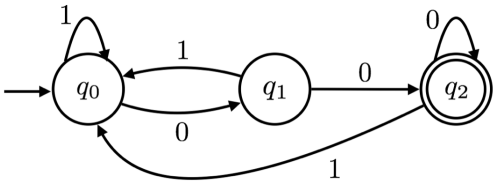
Definition 2.9 (Language recognized/accepted by a DFA)**.**
For a deterministic finite automaton M, we let $L(M)$ denote the set of all strings that M accepts, i.e. $L(M) = \{w \in \Sigma^* : M \text{ accepts } w\}$. We refer to $L(M)$ as the language *recognized* by M (or as the language *accepted* by M, or as the language *decided* by M).[1]

Example 2.10 (Even number of 1's)**.**
The following DFA recognizes the language consisting of all binary strings that contain an even number of 1's.

[1] Here the word "accept" is overloaded since we also use it in the context of a DFA accepting a string. However, this usually does not create any ambiguity. Note that the letter L is also overloaded since we often use it to denote a language $L \subseteq \Sigma^*$. In this definition, you see that it can also denote a function that maps a DFA to a language. Again, this overloading should not create any ambiguity.



Example 2.11 (Ends with 00)**.**
The following DFA recognizes the language consisting of all binary strings that end with 00.



Exercise 2.12 (Draw DFAs)**.**
For each language below (over the alphabet $\Sigma = \{0, 1\}$), draw a DFA recognizing it.

(a) $\{110, 101\}$

(b) $\{0, 1\}^* \backslash \{110, 101\}$

(c) $\{x \in \{0, 1\}^* : x \text{ starts and ends with the same bit}\}$

(d) $\{\epsilon, 110, 110110, 110110110, \ldots\}$

(e) $\{x \in \{0, 1\}^* : x \text{ contains } 110 \text{ as a substring}\}$

Exercise 2.13 (Finite languages are regular)**.**
Let L be a finite language, i.e., it contains a finite number of words . Show that there is a DFA recognizing L.

Definition 2.14 (Regular language)**.**
A language $L \subseteq \Sigma^*$ is called *regular* if there is a deterministic finite automaton M such that $L = L(M)$.

Example 2.15 (Some examples of regular languages)**.**
All the languages in Exercise 2.12 (Draw DFAs) are regular languages.

Exercise 2.16 (Equal number of 01's and 10's)**.**
Is the language

$\{w \in \{0, 1\}^* : w \text{ contains an equal number of occurrences of } 01 \text{ and } 10 \text{ as substrings.}\}$

regular?

2.2 Irregular Languages

Theorem 2.17 (0^n1^n is not regular)**.**
*Let $\Sigma = \{0, 1\}$. The language $L = \{0^n 1^n : n \in \mathbb{N}\}$ is **not** regular.*

Proof. Our goal is to show that $L = \{0^n1^n : n \in \mathbb{N}\}$ is not regular. The proof is by contradiction. So let's assume that L is regular.

Since L is regular, by definition, there is some deterministic finite automaton M that recognizes L. Let k denote the number of states of M. For $n \in \mathbb{N}$, let r_n denote the state that M reaches after reading 0^n (i.e., $r_n = \delta(q_0, 0^n)$). By the pigeonhole principle,[2] we know that there must be a repeat among $r_0, r_1, \ldots, r_k$ (a sequence of $k+1$ states). In other words, there are indices $i, j \in \{0, 1, \ldots, k\}$ with $i \neq j$ such that $r_i = r_j$. This means that the string 0^i and the string 0^j end up in the same state in M. Therefore $0^i w$ and $0^j w$, *for any* string $w \in \{0, 1\}^*$, end up in the same state in M. We'll now reach a contradiction, and conclude the proof, by considering a particular w such that $0^i w$ and $0^j w$ end up in different states.

Consider the string $w = 1^i$. Then since M recognizes L, we know $0^i w = 0^i 1^i$ must end up in an accepting state. On the other hand, since $i \neq j$, $0^j w = 0^j 1^i$ is not in the language, and therefore cannot end up in an accepting state. This is the desired contradiction. □

Theorem 2.18 (A unary non-regular language)**.**
*Let $\Sigma = \{a\}$. The language $L = \{a^{2^n} : n \in \mathbb{N}\}$ is **not** regular.*

Proof. Our goal is to show that $L = \{a^{2^n} : n \in \mathbb{N}\}$ is not regular. The proof is by contradiction. So let's assume that L is regular.

Since L is regular, by definition, there is some deterministic finite automaton M that recognizes L. Let k denote the number of states of M. For $n \in \mathbb{N}$, let r_n denote the state that M reaches after reading a^{2^n} (i.e. $r_n = \delta(q_0, a^{2^n})$). By the pigeonhole principle, we know that there must be a repeat among $r_0, r_1, \ldots, r_k$ (a sequence of $k+1$ states). In other words, there are indices $i, j \in \{0, 1, \ldots, k\}$ with $i < j$ such that $r_i = r_j$. This means that the string a^{2^i} and the string a^{2^j} end up in the same state in M. Therefore $a^{2^i} w$ and $a^{2^j} w$, *for any* string $w \in \{a\}^*$, end up in the same state in M. We'll now reach a contradiction, and conclude the proof, by considering a particular w such that $a^{2^i} w$ ends up in an accepting state but $a^{2^j} w$ ends up in a rejecting state (i.e. they end up in different states).

Consider the string $w = a^{2^i}$. Then $a^{2^i} w = a^{2^i} a^{2^i} = a^{2^{i+1}}$, and therefore must end up in an accepting state. On the other hand, $a^{2^j} w = a^{2^j} a^{2^i} = a^{2^j + 2^i}$. We claim that this word must end up in a rejecting state because $2^j + 2^i$ cannot be written as a power of 2 (i.e., cannot be written as 2^t for some $t \in \mathbb{N}$). To see this, note that since $i < j$, we have

$$2^j < 2^j + 2^i < 2^j + 2^j = 2^{j+1},$$

which implies that if $2^j + 2^i = 2^t$, then $j < t < j+1$. So $2^j + 2^i$ cannot be written as 2^t for $t \in \mathbb{N}$, and therefore $a^{2^j + 2^i}$ leads to a reject state in M as claimed. □

[2] The *pigeonhole principle* states that if n items are put inside m containers, and $n > m$, then there must be at least one container with more than one item. The name *pigeonhole principle* comes from thinking of the items as pigeons, and the containers as holes. The pigeonhole principle is often abbreviated as PHP.

Exercise 2.19 ($a^n b^n c^n$ is not regular)**.**
Let $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$. Prove that $L = \{\texttt{a}^n\texttt{b}^n\texttt{c}^n : n \in \mathbb{N}\}$ is not regular.

Exercise 2.20 ($c^{251}a^n b^{2n}$ is not regular)**.**
Let $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$. Prove that $L = \{\texttt{c}^{251}\texttt{a}^n\texttt{b}^{2n} : n \in \mathbb{N}\}$ is not regular.

2.3 Closure Properties of Regular Languages

Exercise 2.21 (Are regular languages closed under complementation?)**.**
Is it true that if L is regular, than its complement $\Sigma^* \backslash L$ is also regular? In other words, are regular languages *closed* under the complementation operation?

Exercise 2.22 (Are regular languages closed under subsets?)**.**
Is it true that if $L \subseteq \Sigma^*$ is a regular language, then any $L' \subseteq L$ is also a regular language?

Theorem 2.23 (Regular languages are closed under union)**.**
Let Σ be some finite alphabet. If $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Sigma^*$ are regular languages, then the language $L_1 \cup L_2$ is also regular.

Proof. Given regular languages L_1 and L_2, we want to show that $L_1 \cup L_2$ is regular. Since L_1 and L_2 are regular languages, by definition, there are DFAs $M = (Q, \Sigma, \delta, q_0, F)$ and $M' = (Q', \Sigma, \delta', q_0', F')$ that recognize L_1 and L_2 respectively (i.e. $L(M) = L_1$ and $L(M') = L_2$). To show $L_1 \cup L_2$ is regular, we'll construct a DFA $M'' = (Q'', \Sigma, \delta'', q_0'', F'')$ that recognizes $L_1 \cup L_2$. The definition of M'' will make use of M and M'. In particular:

- the set of states is $Q'' = Q \times Q' = \{(q, q') : q \in Q, q' \in Q'\}$;
- the transition function δ'' is defined such that for $(q, q') \in Q''$ and $a \in \Sigma$,
$$\delta''((q, q'), a) = (\delta(q, a), \delta'(q', a));$$
(Note that for $w \in \Sigma^*$, $\delta''((q, q'), w) = (\delta(q, w), \delta'(q', w))$.)
- the initial state is $q_0'' = (q_0, q_0')$;
- the set of accepting states is $F'' = \{(q, q') : q \in F \text{ or } q' \in F'\}$.

This completes the definition of M''. It remains to show that M'' indeed recognizes the language $L_1 \cup L_2$, i.e. $L(M'') = L_1 \cup L_2$. We will first argue that $L_1 \cup L_2 \subseteq L(M'')$ and then argue that $L(M'') \subseteq L_1 \cup L_2$. Both inclusions will follow easily from the definition of M'' and the definition of a DFA accepting a string.

$L_1 \cup L_2 \subseteq L(M'')$: Suppose $w \in L_1 \cup L_2$, which means w either belongs to L_1 or it belongs to L_2. Our goal is to show that $w \in L(M'')$. Without loss of generality, assume w belongs to L_1, or in other words, M accepts w (the argument is essentially identical when w belongs to L_2). So we know that $\delta(q_0, w) \in F$. By the definition of δ'', $\delta''((q_0, q_0'), w) = (\delta(q_0, w), \delta'(q_0', w))$. And since $\delta(q_0, w) \in F$, $(\delta(q_0, w), \delta'(q_0', w)) \in F''$ (by the definition of F''). So w is accepted by M'' as desired.

$L(M'') \subseteq L_1 \cup L_2$: Suppose that $w \in L(M'')$. Our goal is to show that $w \in L_1$ or $w \in L_2$. Since w is accepted by M'', we know that $\delta''((q_0, q_0'), w) = (\delta(q_0, w), \delta'(q_0', w)) \in F''$. By the definition of F'', this means that either $\delta(q_0, w) \in F$ or $\delta'(q_0', w) \in F'$, i.e., w is accepted by M or M'. This implies that either $w \in L(M) = L_1$ or $w \in L(M') = L_2$, as desired. □

Note 2.24 (On proof write-up)**.**
Observe that the proof of Theorem 2.23 (Regular languages are closed under union) contains very little information about how one comes up with such a proof or what is the "right" intuitive interpretation of the construction. Many proofs in the literature are actually written in this manner, which can be frustrating for the reader. We have explained the intuition and the cognitive process that goes into discovering the above proof during class. Therefore we chose not to include any of these details in the above write-up. However, we do encourage you to include a "proof idea" component in your write-ups when you believe that the intuition is not very transparent.

Corollary 2.25 (Regular languages are closed under intersection)**.**
Let Σ be some finite alphabet. If $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Sigma^*$ are regular languages, then the language $L_1 \cap L_2$ is also regular.

Proof. We want to show that regular languages are closed under the intersection operation. We know that regular languages are closed under union (Theorem 2.23 (Regular languages are closed under union)) and closed under complementation (Exercise 2.21 (Are regular languages closed under complementation?)). The result then follows since $A \cap B = \overline{\overline{A} \cup \overline{B}}$. □

Exercise 2.26 (Direct proof that regular languages are closed under difference)**.**

Give a direct proof (without using the fact that regular languages are closed under complementation, union and intersection) that if L_1 and L_2 are regular languages, then $L_1 \backslash L_2$ is also regular.

Exercise 2.27 (Finite vs infinite union)**.**

(a) Suppose $L_1, \ldots, L_k$ are all regular languages. Is it true that their union $\bigcup_{i=0}^{k} L_i$ must be a regular language?

(b) Suppose $L_0, L_1, L_2, \ldots$ is an infinite sequence of regular languages. Is it true that their union $\bigcup_{i \geq 0} L_i$ must be a regular language?

Exercise 2.28 (Union of irregular languages)**.**
Suppose L_1 and L_2 are not regular languages. Is it always true that $L_1 \cup L_2$ is not a regular language?

Exercise 2.29 (Regularity of suffixes and prefixes)**.**
Suppose $L \subseteq \Sigma^*$ is a regular language. Show that the following languages are also regular:

$$\text{SUFFIXES}(L) = \{x \in \Sigma^* : yx \in L \text{ for some } y \in \Sigma^*\},$$
$$\text{PREFIXES}(L) = \{y \in \Sigma^* : yx \in L \text{ for some } x \in \Sigma^*\}.$$

Theorem 2.30 (Regular languages are closed under concatenation)**.**
If $L_1, L_2 \subseteq \Sigma^*$ are regular languages, then the language $L_1 L_2$ is also regular.

Proof. Given regular languages L_1 and L_2, we want to show that L_1L_2 is regular. Since L_1 and L_2 are regular languages, by definition, there are DFAs $M = (Q, \Sigma, \delta, q_0, F)$ and $M' = (Q', \Sigma, \delta', q_0', F')$ that recognize L_1 and L_2 respectively. To show L_1L_2 is regular, we'll construct a DFA $M'' = (Q'', \Sigma, \delta'', q_0'', F'')$ that recognizes L_1L_2. The definition of M'' will make use of M and M'.

Before we formally define M'', we will introduce a few key concepts and explain the intuition behind the construction.

We know that $w \in L_1L_2$ if and only if there is a way to write w as uv where $u \in L_1$ and $v \in L_2$. With this in mind, we first introduce the notion of a *thread*. Given a word $w = w_1w_2 \dots w_n \in \Sigma^*$, a *thread* with respect to w is a sequence of states

$$r_0, r_1, r_2, \dots, r_i, s_{i+1}, s_{i+2}, \dots, s_n,$$

where $r_0, r_1, \dots, r_i$ is an accepting computation path of M with respect to $w_1w_2 \dots w_i$, [3] and $q_0', s_{i+1}, s_{i+2}, \dots, s_n$ is a computation path (not necessarily accepting) of M' with respect to $w_{i+1}w_{i+2} \dots w_n$. A thread like this corresponds to simulating M on $w_1w_2 \dots w_i$ (at which point we require that an accepting state of M is reached), and then simulating M' on $w_{i+1}w_{i+2} \dots w_n$. For each way of writing w as uv where $u \in L_1$, there is a corresponding thread for it. Note that $w \in L_1L_2$ if and only if there is a thread in which $s_n \in F'$. Our goal is to construct the DFA M'' such that it keeps track of all possible threads, and if one of the threads ends with a state in F', then M'' accepts.

At first, it might seem like one cannot keep track of all possible threads using only *constant* number of states. However this is not the case. Let's identify a thread with its sequence of s_j's (i.e. the sequence of states from Q' corresponding to the simulation of M'). Consider two threads (for the sake of example, let's take $n = 10$):

$$\begin{array}{r} s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10} \\ s_5', s_6', s_7', s_8', s_9', s_{10}' \end{array}$$

If, say, $s_i = s_i' = q' \in Q'$ for some i, then $s_j = s_j'$ for all $j > i$ (in particular, $s_{10} = s_{10}'$). At the end, all we care about is whether s_{10} or s_{10}' is an accepting state of M'. So at index i, we do not need to remember that there are two copies of q'; it suffices to keep track of one copy. In general, at any index i, when we look at all the possible threads, we want to keep track of the unique states that appear at that index, and not worry about duplicates. Since we do not need to keep track of duplicated states, what we need to remember is a *subset* of Q' (recall that a set cannot have duplicated elements).

The construction of M'' we present below keeps track of all the threads using constant number of states. Indeed, the set of states is[4]

$$Q'' = Q \times \mathcal{P}(Q') = \{(q, S) : q \in Q, S \subseteq Q'\},$$

where the first component keeps track of which state we are at in M, and the second component keeps track of all the unique states of M' that we can be at if we are following one of the possible threads.

Before we present the formal definition of M'', we introduce one more definition. Recall that the transition function of M' is $\delta' : Q' \times \Sigma \to Q'$. Using δ' we define a new function $\delta_{\mathcal{P}}' : \mathcal{P}(Q') \times \Sigma \to \mathcal{P}(Q')$ as follows. For $S \subseteq Q'$ and $a \in \Sigma$, $\delta_{\mathcal{P}}'(S, a)$ is defined to be the set of all possible states that we can end up at if we start in a state in S and read the symbol a. In other words,

$$\delta_{\mathcal{P}}'(S, a) = \{\delta'(q', a) : q' \in S\}.$$

[3] This means $r_0 = q_0$, $r_i \in F$, and when the symbol w_j is read, M transitions from state r_{j-1} to state r_j. See Definition 2.3 (Computation path for a DFA).

[4] Recall that for any set Q, the set of all subsets of Q is called the *power set* of Q, and is denoted by $\mathcal{P}(Q)$.

It is appropriate to view $\delta'_{\mathcal{P}}$ as an extension/generalization of δ'.

Here is the formal definition of M'':

- The set of states is $Q'' = Q \times \mathcal{P}(Q') = \{(q, S) : q \in Q, S \subseteq Q'\}$.

  (The first coordinate keeps track of which state we are at in the first machine M, and the second coordinate keeps track of the set of states we can be at in the second machine M' if we follow one of the possible threads.)

- The transition function δ'' is defined such that for $(q, S) \in Q''$ and $a \in \Sigma$,

$$\delta''((q, S), a) = \begin{cases} (\delta(q, a), \delta'_{\mathcal{P}}(S, a)) & \text{if } \delta(q, a) \notin F, \\ (\delta(q, a), \delta'_{\mathcal{P}}(S, a) \cup \{q'_0\}) & \text{if } \delta(q, a) \in F. \end{cases}$$

  (The first coordinate is updated according to the transition rule of the first machine. The second coordinate is updated according to the transition rule of the second machine. Since for the second machine, we are keeping track of all possible states we could be at, the extended transition function $\delta'_{\mathcal{P}}$ gives us all possible states we can go to when reading a character a. Note that if after applying δ to the first coordinate, we get a state that is an accepting state of the first machine, a new thread must be created and kept track of. This is accomplished by adding q'_0 to the second coordinate.)

- The initial state is

$$q''_0 = \begin{cases} (q_0, \emptyset) & \text{if } q_0 \notin F, \\ (q_0, \{q'_0\}) & \text{if } q_0 \in F. \end{cases}$$

  (Initially, if $q_0 \notin F$, then there are no threads to keep track of, so the second coordinate is the empty set. On the other hand, if $q_0 \in F$, then there is already a thread that we need to keep track of – the one corresponding to running the whole input word w on the second machine – so we add q'_0 to the second coordinate to keep track of this thread.)

- The set of accepting states is $F'' = \{(q, S) : q \in Q, S \subseteq Q', S \cap F' \neq \emptyset\}$.

  (In other words, M'' accepts if and only if there is a state in the second coordinate that is an accepting state of the second machine M'. So M'' accepts if and only if one of the possible threads ends in an accepting state of M'.)

This completes the definition of M''.

To see that M'' indeed recognizes the language L_1L_2, i.e. $L(M'') = L_1L_2$, note that by construction, M'' with input w, does indeed keep track of all the possible threads. And it accepts w if and only if one of those threads ends in an accepting state of M'. The result follows since $w \in L_1L_2$ if and only if there is a thread with respect to w that ends in an accepting state of M'. □

Exercise 2.31 (Regular languages are closed under star)**.**
Critique the following argument that claims to establish that regular languages are closed under the concatenation operation, that is, if L is a regular language, then so is L^*.

> Let L be a regular language. We know that by definition $L^* = \bigcup_{n \in \mathbb{N}} L^n$, where $L^n = \{u_1u_2 \dots u_n : u_i \in L \text{ for all } i\}$. We know that for all n, L^n must be regular using Theorem 2.30 (Regular languages are closed under concatenation). And since L^n is regular for all n, we know L^* must be regular using Theorem 2.23 (Regular languages are closed under union).

Quiz

1. Fix some alphabet Σ. How many DFAs are there with exactly one state?
2. Let $L \subseteq \{\texttt{a}\}^*$ be a language consisting of all strings of $\texttt{a}$'s of odd length except length 15251. Is L regular?
3. Let L be the set of all strings in $\{\texttt{0}, \texttt{1}\}^*$ that contain at least 15251 $\texttt{0}$'s and at most 15251 $\texttt{1}$'s. Is L regular?
4. True or false: Let $L_1 \oplus L_2$ denote the set of all words in either L_1 or L_2, but not both. If L_1 and L_2 are regular, then so is $L_1 \oplus L_2$.
5. True or false: For languages L and L', if $L \subseteq L'$ and L is non-regular, then L' is non-regular.
6. True or false: If $L \subseteq \Sigma^*$ is non-regular, then $\overline{L} = \Sigma^* \setminus L$ is non-regular.

Hints to Selected Exercises

Exercise 2.13 (Finite languages are regular):
First think about whether languages of size 1 are regular? Are languages of size 2 regular? (The first part of Exercise (Draw DFAs) might help.) Can you generalize your idea to any finite size language?

Exercise 2.16 (Equal number of 01's and 10's):
Yes, it is.

Exercise 2.21 (Are regular languages closed under complementation?):
The answer is yes. How can you construct a DFA recognizing $\Sigma^* \setminus L$ given that you have a DFA recognizing L?

Exercise 2.22 (Are regular languages closed under subsets?):
The answer is no. Find a counter-example.

Exercise 2.27 (Finite vs infinite union):
The answer for the first part is yes, and the second part is no.

Exercise 2.28 (Union of irregular languages):
The answer is no. Find a counter-example.

Chapter 3

Turing Machines

PREAMBLE

Chapter structure:

- Section 14.1.1 (Basic Definitions)
  - Definition 3.1 (Turing machine)
  - Definition 3.6 (A TM accepting or rejecting a string)
  - Definition 3.9 (Decider Turing machine)
  - Definition 3.10 (Language accepted and decided by a TM)
  - Definition 3.11 (Decidable language)
  - Definition 3.18 (Universal Turing machine)
- Section 3.2 (Decidable Languages)
  - Definition 3.22 (Languages related to encodings of DFAs)
  - Theorem 3.23 ($\text{ACCEPTS}_{\text{DFA}}$ and $\text{SELF-ACCEPTS}_{\text{DFA}}$ are decidable)
  - Theorem 3.24 ($\text{EMPTY}_{\text{DFA}}$ is decidable)
  - Theorem 3.25 (EQ_{DFA} is decidable)

Chapter goals:

In this chapter, our main goal is to introduce the definition of a Turing machine, which is the standard mathematical model for any kind of computational device. As such, this definition is very foundational. As we discuss in lecture, the physical Church-Turing thesis asserts that any kind of physical device or phenomenon, when viewed as a computational process mapping input data to output data, can be simulated by some Turing machine. Thus, rigorously studying Turing machines does not just give us insights about what our laptops can or cannot do, but also tells us what the universe can and cannot do computationally.

This chapter kicks things off with examples of decidable languages (i.e. decision problems that we can compute). Next chapter, we will start exploring the limitations of computation. Some of the examples we cover in this chapter will serve as a warm up to other examples we will discuss in the next chapter in the context of uncomputability.

3.1 Basic Definitions

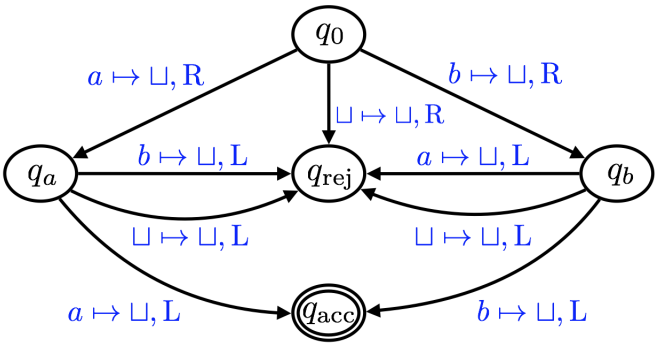
Definition 3.1 (Turing machine)**.**
A Turing machine (TM) M is a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}),$$

where

- Q is a non-empty finite set
  (which we refer to as the *set of states*);
- Σ is a non-empty finite set that does not contain the *blank symbol* $\sqcup$
  (which we refer to as the *input alphabet*);
- Γ is a finite set such that $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$
  (which we refer to as the *tape alphabet*);
- δ is a function of the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$
  (which we refer to as the *transition function*);
- $q_0 \in Q$ is an element of Q
  (which we refer to as the *initial state* or *starting state*);
- $q_{\text{acc}} \in Q$ is an element of Q
  (which we refer to as the *accepting state*);
- $q_{\text{rej}} \in Q$ is an element of Q such that $q_{\text{rej}} \neq q_{\text{acc}}$
  (which we refer to as the *rejecting state*).

Example 3.2 (A 5-state TM)**.**
Below is an example of how we draw a TM:



In this example, $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $\Gamma = \{\mathtt{a}, \mathtt{b}, \sqcup\}$, $Q = \{q_0, q_a, q_b, q_{\text{acc}}, q_{\text{rej}}\}$. The labeled arrows between the states encode the transition function δ. As an example, the arrow from state q_0 to q_a represents $\delta(q_0, \mathtt{a}) = (q_a, \sqcup, \mathrm{R})$. The above picture is called the *state diagram* of the Turing machine.

Note 3.3 (Equivalence of Turing machines)**.**
We'll consider two Turing machines to be equivalent/same if they are the same machine up to renaming the elements of the sets Q, Σ and Γ.

Note 3.4 (No transition out of accepting and rejecting states)**.**
In the transition function δ of a TM, we don't really care about how we define the output of δ when the input state is q_{acc} or q_{rej} because once the computation reaches one of these states, it stops. We explain this below in Definition 3.6 (A TM accepting or rejecting a string).

IMPORTANT 3.5 (A Turing machine uses a tape)**.**
A Turing Machine is always accompanied by a *tape* that is used as memory. The tape is just a sequence of *cells* that can hold any symbol from the tape alphabet. The tape can be defined so that it is infinite in two directions (so we could imagine indexing the cells using the integers $\mathbb{Z}$), or it could be infinite in one direction, to the right (so we could imagine indexing the cells using the natural numbers $\mathbb{N}$). Initially, an input $w_1 \dots w_n \in \Sigma^*$ is put on the tape so that symbol w_i is placed on the cell with index $i-1$. In these notes, we assume our tape is infinite in two directions.

Definition 3.6 (A TM accepting or rejecting a string)**.**
Let M be a Turing machine where Q is the set of states, $\sqcup$ is the blank symbol, and Γ is the tape alphabet.[1] To understand how M's computation proceeds we generally need to keep track of three things: (i) the state M is in; (ii) the contents of the tape; (iii) where the tape head is. These three things are collectively known as the "configuration" of the TM. More formally: a *configuration* for M is defined to be a string $uqv \in (\Gamma \cup Q)^*$, where $u, v \in \Gamma^*$ and $q \in Q$. This represents that the tape has contents $\cdots \sqcup\sqcup\sqcup uv \sqcup\sqcup\sqcup \cdots$, the head is pointing at the leftmost symbol of v, and the state is q. We say the configuration is *accepting* if q is M's accept state and that it's *rejecting* if q is M's reject state.[2]

Suppose that M reaches a certain configuration α (which is not accepting or rejecting). Knowing just this configuration and M's transition function δ, one can determine the configuration β that M will reach at the next step of the computation. (As an exercise, make this statement precise.) We write

$$\alpha \vdash_M \beta$$

and say that "α yields β (in M)". If it's obvious what M we're talking about, we drop the subscript M and just write $\alpha \vdash \beta$.

Given an input $x \in \Sigma^*$ we say that $M(x)$ *halts* if there exists a sequence of configurations (called the *computation trace*) $\alpha_0, \alpha_1, \dots, \alpha_T$ such that:

(i) $\alpha_0 = q_0 x$, where q_0 is M's initial state;

(ii) $\alpha_t \vdash_M \alpha_{t+1}$ for all $t = 0, 1, 2, \dots, T-1$;

(iii) α_T is either an accepting configuration (in which case we say $M(x)$ *accepts*) or a rejecting configuration (in which case we say $M(x)$ *rejects*).

Otherwise, we say $M(x)$ *loops*.

IMPORTANT 3.7 (Turing machines can loop forever)**.**
Given any DFA and any input string, the DFA always halts and makes a decision to either reject or accept the string. The same is not true for Turing machines. It is possible that a Turing machine does not make a decision when given an input string, and instead, loops forever. So given a TM M and an input string x, there are 3 options when we run M on x:

[1] Supernerd note: we will always assume Q and Γ are disjoint sets.

[2] There are some technicalities: The string u cannot start with $\sqcup$ and the string v cannot end with $\sqcup$. This is so that the configuration is always unique. Also, if $v = \epsilon$ it means the head is pointing at the $\sqcup$ immediately to the right of u.

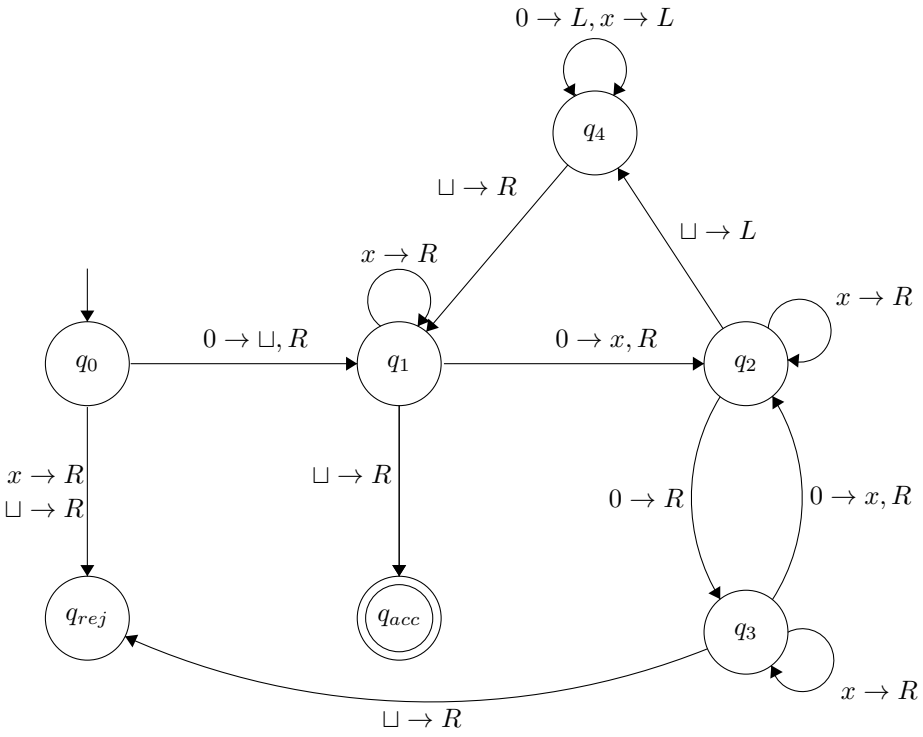
- M accepts x;
- M rejects x;
- M loops forever.

This is an important distinction between DFAs and TMs.

Exercise 3.8 (Practice with configurations)**.**

(a) Suppose $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ is a Turing machine. We want you to formally define $\alpha \vdash_M \beta$. More precisely, suppose $\alpha = uqv$, where $q \in Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}$. Precisely describe β.

(b) Let M denote the Turing machine shown below, which has input alphabet $\Sigma = \{0\}$ and tape alphabet $\Gamma = \{0, x, \sqcup\}$. (Note on notation: A transition label usually has two symbols, one corresponding to the symbol being read, and the other corresponding to the symbol being written. If a transition label has one symbol, the interpretation is that the symbol being read and written is exactly the same.)



We want you to prove that M accepts the input 0000 using the definition on the previous page. More precisely, we want you to write out the computation trace

$$\alpha_0 \vdash_M \alpha_1 \vdash_M \cdots \vdash_M \alpha_T$$

for $M(0000)$. You do not have to justify it; just make sure to get T and $\alpha_0, \ldots, \alpha_T$ correct!

Definition 3.9 (Decider Turing machine)**.**
A Turing machine is called a *decider* if it halts on all inputs.

Definition 3.10 (Language accepted and decided by a TM)**.**
Let M be a Turing machine (not necessarily a decider). We denote by $L(M)$ the set of all strings that M accepts, and we call $L(M)$ the language *accepted* by M. When M is a decider, we say that M *decides* the language $L(M)$.

Definition 3.11 (Decidable language)**.**
A language L is called *decidable* (or *computable*) if $L = L(M)$ for some decider Turing machine M.

Exercise 3.12 (A simple decidable language)**.**
Give a description of the language decided by the TM shown in the example corresponding to Definition 3.1 (Turing machine).

Exercise 3.13 (Drawing TM state diagrams)**.**
For each language below, draw the state diagram of a TM that decides the language. You can use any finite tape alphabet Γ containing the elements of Σ and the symbol $\sqcup$.

(a) $L = \{0^n 1^n : n \in \mathbb{N}\}$, where $\Sigma = \{0, 1\}$.

(b) $L = \{0^n : n \text{ is a nonnegative integer power of } 2\}$, where $\Sigma = \{0\}$.

IMPORTANT 3.14 (The Church-Turing Thesis)**.**
The Church-Turing Thesis (CTT)[3] states that any computation that can be conducted in this universe (constrained by the laws of physics of course), can be carried out by a TM. There are a couple of important things to highlight. First, CTT says nothing about the efficiency of the simulation.[4] Second, CTT is not a mathematical statement, but a physical claim about the universe we live in (similar to claiming that the speed of light is constant). The implications of CTT is far-reaching. For example, CTT claims that any computation that can be carried out by a human can be carried out by a TM. Other implications are discussed in lecture.

Note 3.15 (Low-level, medium-level, high-level descriptions of TMs)**.**
A low-level description of a TM is given by specifying the 7-tuple in its definition. This information is often presented using a picture of its state diagram. A medium-level description includes an English description of the movement and behavior of the tape head, as well as how the contents of the tape is changing, as the computation is being carried out. A high-level description is pseudocode or an algorithm written in English. Usually, an algorithm is written in a way so that a human could read it, understand it, and carry out its steps. By CTT, there is a TM that can carry out the same computation. Unless explicitly stated otherwise, you can present a TM using a high-level description.

[3] The statement we are using here is often called the Physical Church-Turing Thesis and is more general than the original Church-Turing Thesis. In the original Church-Turing Thesis, computation is considered to correspond to a human following step-by-step instructions.

[4] As an example, quantum computers can be simulated by TMs, but in certain cases, we believe that the simulation can be exponentially slower.

Note 3.16 (Encodings of machines)**.**
In Chapter 1 we saw that we can use the notation $\langle \cdot \rangle$ to denote an encoding of objects belonging to any countable set. For example, if D is a DFA, we can write $\langle D \rangle$ to denote the encoding of D as a string. If M is a TM, we can write $\langle M \rangle$ to denote the encoding of M. There are many ways one can encode DFAs and TMs. We will not be describing a specific encoding scheme as this detail will not be important for us.[5]

Recall that when we want to encode a tuple of objects, we use the comma sign. For example, if M_1 and M_2 are two Turing machines, we write $\langle M_1, M_2 \rangle$ to denote the encoding of the tuple (M_1, M_2). As another example, if M is a TM and $x \in \Sigma^*$, we can write $\langle M, x \rangle$ to denote the encoding of the tuple (M, x).

IMPORTANT 3.17 (Code is data)**.**
The fact that we can encode different types of objects with strings has the corollary that a Turing machine, or any piece of code, can be viewed as a string, and therefore as data. This means code can take as input other code (in fact, code can take itself as the input). This point of view has several important implications, one of which is the fact that we can come up with a Turing machine, which given as input the description of any Turing machine, can simulate it. This simulator Turing machine is called a universal Turing machine.

Definition 3.18 (Universal Turing machine)**.**
Let Σ be some finite alphabet. A *universal Turing machine* U is a Turing machine that takes $\langle M, x \rangle$ as input, where M is a TM and x is a word in Σ^*, and has the following high-level description:

> M: Turing machine. x: string in Σ^*.
> $U(\langle M, x \rangle)$:
> 1 Simulate M on input x (i.e. run $M(x)$).
> 2 If it accepts, accept.
> 3 If it rejects, reject.

Note that if $M(x)$ loops forever, then U loops forever as well. To make sure M always halts, we can add a third input, an integer k, and have the universal machine simulate the input TM for at most k steps.

IMPORTANT 3.19 (Checking the input type)**.**
When we give a high-level description of a TM, we often assume that the input given is of the correct form/type. For example, with the Universal TM above, we assumed that the input was the encoding $\langle M, x \rangle$ where M is a TM and x is an input string for M. But technically, the input to the universal TM could be any finite-length string. What do we do if the input string does not correspond to a valid encoding of an expected type of input object?

Even though this is not explicitly written, we will implicitly assume that the first thing our machine does is check whether the input is a valid encoding of an object with the expected type. If it is not, the machine rejects. If it is, then it will carry on with the specified instructions.

The important thing to keep in mind is that in our descriptions of Turing machines, this step of checking whether the input string has the correct form (i.e. that it is a valid encoding) will never be explicitly written, and we don't expect you to explicitly write it either. That being said, be aware that this check is implicitly there.

[5] As an example, if P is some Python program, we can take $\langle P \rangle$ to be the string that represents the source code of the program. A DFA or a TM can also be viewed as a piece of code (as discussed in lecture). So we could define an encoded DFA or TM to be the string that represents that code.

3.2 Decidable Languages

Exercise 3.20 (Decidability is closed under intersection and union)**.**
Let L and K be decidable languages. Show that $L \cap K$ and $L \cup K$ are also decidable by presenting high-level descriptions of TMs deciding them.

Exercise 3.21 (Decidable language based on pi)**.**
Let $L \subseteq \{3\}^*$ be defined as follows: $x \in L$ if and only if x appears somewhere in the decimal expansion of π. For example, the strings ϵ, 3, and 33 are all definitely in L, because

$$\pi = 3.14159265358979323846264 33\ldots$$

Prove that L is decidable. No knowledge in number theory is required to solve this question.

Definition 3.22 (Languages related to encodings of DFAs)**.**
Fix some alphabet Σ. We define the following languages:

$$\begin{aligned}
\text{ACCEPTS}_{\text{DFA}} &= \{\langle D, x\rangle : D \text{ is a DFA that accepts the string } x\},\\
\text{SELF-ACCEPTS}_{\text{DFA}} &= \{\langle D\rangle : D \text{ is a DFA that accepts the string } \langle D\rangle\},\\
\text{EMPTY}_{\text{DFA}} &= \{\langle D\rangle : D \text{ is a DFA with } L(D) = \emptyset\},\\
\text{EQ}_{\text{DFA}} &= \{\langle D_1, D_2\rangle : D_1 \text{ and } D_2 \text{ are DFAs with } L(D_1) = L(D_2)\}.
\end{aligned}$$

Theorem 3.23 ($\text{ACCEPTS}_{\text{DFA}}$ and $\text{SELF-ACCEPTS}_{\text{DFA}}$ are decidable)**.**
The languages $\text{ACCEPTS}_{\text{DFA}}$ *and* $\text{SELF-ACCEPTS}_{\text{DFA}}$ *are decidable.*

Proof. Our goal is to show that $\text{ACCEPTS}_{\text{DFA}}$ and $\text{SELF-ACCEPTS}_{\text{DFA}}$ are decidable languages. To show that these languages are decidable, we will give high-level descriptions of TMs deciding them.

For $\text{ACCEPTS}_{\text{DFA}}$, the decider is essentially the same as a universal TM:

> D: DFA. x: string.
> $M(\langle D, x\rangle)$:
> 1 Simulate D on input x (i.e. run $D(x)$).
> 2 If it accepts, accept.
> 3 If it rejects, reject.

It is clear that this correctly decides $\text{ACCEPTS}_{\text{DFA}}$.

For $\text{SELF-ACCEPTS}_{\text{DFA}}$, we just need to slightly modify the above machine:

> D: DFA.
> $M(\langle D\rangle)$:
> 1 Simulate D on input $\langle D\rangle$ (i.e. run $D(\langle D\rangle)$).
> 2 If it accepts, accept.
> 3 If it rejects, reject.

Again, it is clear that this correctly decides $\text{SELF-ACCEPTS}_{\text{DFA}}$. □

Theorem 3.24 ($\text{EMPTY}_{\text{DFA}}$ is decidable)**.**
The language $\text{EMPTY}_{\text{DFA}}$ *is decidable.*

Proof. Our goal is to show $\text{EMPTY}_{\text{DFA}}$ is decidable and we will do so by constructing a decider for $\text{EMPTY}_{\text{DFA}}$.

A decider for $\text{EMPTY}_{\text{DFA}}$ takes as input $\langle D \rangle$ for some DFA $D = (Q, \Sigma, \delta, q_0, F)$, and needs to determine if $L(D) = \emptyset$. In other words, it needs to determine if there is any string that D accepts. If we view the DFA as a directed graph,[6] where the states of the DFA correspond to the nodes in the graph and transitions correspond to edges, notice that the DFA accepts some string if and only if there is a directed path from q_0 to some state in F. Therefore, the following decider decides $\text{EMPTY}_{\text{DFA}}$ correctly.

> D: DFA.
> $M(\langle D \rangle)$:
> 1. Build a directed graph from $\langle D \rangle$.
> 2. Run a graph search algorithm starting from the starting state of D.
> 3. If a node corresponding to an accepting state is reached, reject.
> 4. Else, accept.

□

Theorem 3.25 (EQ_{DFA} is decidable)**.**
The language EQ_{DFA} *is decidable.*

Proof. Our goal is to show that EQ_{DFA} is decidable. We will do so by constructing a decider for EQ_{DFA}.

Our argument is going to use the fact that $\text{EMPTY}_{\text{DFA}}$ is decidable (Theorem 3.24 ($\text{EMPTY}_{\text{DFA}}$ is decidable)). In particular, the decider we present for EQ_{DFA} will use the decider for $\text{EMPTY}_{\text{DFA}}$ as a subroutine. Let M denote a decider TM for $\text{EMPTY}_{\text{DFA}}$.

A decider for EQ_{DFA} takes as input $\langle D_1, D_2 \rangle$, where D_1 and D_2 are DFAs. It needs to determine if $L(D_1) = L(D_2)$ (i.e. accept if $L(D_1) = L(D_2)$ and reject otherwise). We can determine if $L(D_1) = L(D_2)$ by looking at their *symmetric difference*[7]

$$(L(D_1) \cap \overline{L(D_2)}) \cup (\overline{L(D_1)} \cap L(D_2)).$$

Note that $L(D_1) = L(D_2)$ if and only if the symmetric difference is empty. Our decider for EQ_{DFA} will construct a DFA D such that $L(D) = (L(D_1) \cap \overline{L(D_2)}) \cup (\overline{L(D_1)} \cap L(D_2))$, and then run $M(\langle D \rangle)$ to determine if $L(D) = \emptyset$. This then tells us if $L(D_1) = L(D_2)$.

To give a bit more detail, observe that given D_1 and D_2, we can

- construct DFAs $\overline{D_1}$ and $\overline{D_2}$ that accept $\overline{L(D_1)}$ and $\overline{L(D_2)}$ respectively (see Exercise 2.21 (Are regular languages closed under complementation?));
- construct a DFA that accepts $L(D_1) \cap \overline{L(D_2)}$ by using the (constructive) proof that regular languages are closed under the intersection operation;[8]

[6] Even though we have not formally defined the notion of a graph yet, we do assume you are familiar with the concept from a prerequisite course and that you have seen some simple graph search algorithms like Breadth-First Search or Depth-First Search.

[7] The symmetric difference of sets A and B is the set of all elements that belong to either A or B, but not both. In set notation, it corresponds to $(A \cap \overline{B}) \cup (\overline{A} \cap B)$.

[8] The constructive proof gives us a way to construct the DFA accepting $L(D_1) \cap \overline{L(D_2)}$ given D_1 and $\overline{D_2}$.

- construct a DFA that accepts $\overline{L(D_1)} \cap L(D_2)$ by using the proof that regular languages are closed under the intersection operation;
- construct a DFA, call it D, that accepts $(L(D_1) \cap \overline{L(D_2)}) \cup (\overline{L(D_1)} \cap L(D_2))$ by using the constructive proof that regular languages are closed under the union operation.

The decider for $\mathrm{EQ}_{\mathrm{DFA}}$ is as follows.

D_1: DFA. D_2: DFA.
$M'(\langle D_1, D_2 \rangle)$:
1. Construct DFA D as described above.
2. Run $M(\langle D \rangle)$.
3. If it accepts, accept.
4. If it rejects, reject.

By our discussion above, the decider works correctly. □

IMPORTANT 3.26 (Decidability through reductions)**.**
Suppose L and K are two languages and K is decidable. We say that solving L *reduces* to solving K if given a decider M_K for K, we can construct a decider for L that uses M_K as a subroutine, thereby establishing L is also decidable. For example, the proof of Theorem 3.25 ($\mathrm{EQ}_{\mathrm{DFA}}$ is decidable) shows that solving $\mathrm{EQ}_{\mathrm{DFA}}$ reduces to solving $\mathrm{EMPTY}_{\mathrm{DFA}}$. Reduction is a powerful tool to expand the landscape of decidable languages.

Exercise 3.27 (Practice with decidability through reductions)**.**

(a) Let $L = \{\langle D_1, D_2 \rangle : D_1 \text{ and } D_2 \text{ are DFAs with } L(D_1) \subsetneq L(D_2)\}$.[9] Show that L is decidable.

(b) Let $K = \{\langle D \rangle : D \text{ is a DFA that accepts } w^R \text{ whenever it accepts } w\}$, where w^R denotes the *reversal* of w. Show that K is decidable. For this question, you can use the fact given a DFA D, there is an algorithm to construct a DFA D' such that $L(D') = L(D)^R = \{w^R : w \in L(D)\}$.

[9] Note on notation: for sets A and B, we write $A \subsetneq B$ if $A \subseteq B$ and $A \neq B$.

Quiz

1. True or false: A TM can have an infinite number of states.
2. True or false: It is possible that in the definition of a TM, $\Sigma = \Gamma$, where Σ is the input alphabet, and Γ is the tape alphabet.
3. True or false: On every input, any TM either accepts or rejects.
4. True or false: Consider a TM such that the starting state q_0 is also the accepting state q_{accept}. It is possible that this TM does not halt on some inputs.
5. Is the following statement true, false, or hard to determine with the knowledge we have so far? $\emptyset$ is decidable.
6. Is the following statement true, false, or hard to determine with the knowledge we have so far? Σ^* is decidable.
7. True or false: $L \subseteq \Sigma^*$ is *un*decidable if and only if $\Sigma^* \backslash L$ is *un*decidable.
8. Is the following statement true, false, or hard to determine with the knowledge we have so far? The language $\{\langle M \rangle : M \text{ is a TM with } L(M) = \emptyset\}$ is decidable.

Hints to Selected Exercises

Exercise 3.21 (Decidable language based on pi):
Case on the different possibilities that L could be. For example, one option is that $L = \{3\}^*$, but there are other options too. Show that in all cases L is decidable.

Exercise 3.27 (Practice with decidability through reductions):
Part (a): You may want to use a decider for $\text{EMPTY}_{\text{DFA}}$ and a decider for EQ_{DFA}.
Part (b): You may want to use a decider for EQ_{DFA} together with the given fact in the description of the problem (i.e. given any DFA D, there is an algorithm to construct DFA D' such that $L(D') = L(D)^R$).

Chapter 4

Countable and Uncountable Sets

PREAMBLE

Chapter structure:

- Section 14.1.1 (Basic Definitions)
  - Definition 4.1 (Injection, surjection, and bijection)
  - Theorem 4.2 (Relationships between different types of functions)
  - Definition 4.4 (Comparison of cardinality of sets)
  - Definition 4.6 (Countable and uncountable sets)
  - Theorem 4.7 (Characterization of countably infinite sets)
- Section 4.2 (Countable Sets)
  - Proposition 4.10 ($\mathbb{Z} \times \mathbb{Z}$ is countable)
  - Proposition 4.11 ($\mathbb{Q}$ is countable)
  - Proposition 4.12 (Σ^* is countable)
  - Proposition 4.14 (The set of Turing machines is countable)
  - Proposition 4.15 (The set of polynomials with rational coefficients is countable)
- Section 4.3 (Uncountable Sets)
  - Theorem 4.17 (Cantor's Theorem)
  - Corollary 4.18 ($\mathcal{P}(\mathbb{N})$ is uncountable)
  - Corollary 4.19 (The set of languages is uncountable)
  - Definition 4.20 (Σ^∞)
  - Theorem 4.21 ($\{0,1\}^\infty$ is uncountable)

Chapter goals:

In this chapter, we would like to remind you the concepts of countable and uncountable sets, as well as the general techniques involved in countability and uncountability proofs. Even though it may seem like we are diverging from the main discussion on Turing machines and decidability, we'll see in the next chapter that the concepts in this chapter are intimately related to the concepts of decidability and undecidability. Countable and uncountable sets, together with the diagonalization proof technique for showing a set is uncountable, have major applications in proving the limits of computation.

4.1 Basic Definitions

Definition 4.1 (Injection, surjection, and bijection)**.**
Let A and B be two (possibly infinite) sets.

- A function $f : A \to B$ is called *injective* if for any $a, a' \in A$ such that $a \neq a'$, we have $f(a) \neq f(a')$. We write $A \hookrightarrow B$ if there exists an injective function from A to B.
- A function $f : A \to B$ is called *surjective* if for all $b \in B$, there exists an $a \in A$ such that $f(a) = b$. We write $A \twoheadrightarrow B$ if there exists a surjective function from A to B.
- A function $f : A \to B$ is called *bijective* (or *one-to-one correspondence*) if it is both injective and surjective. We write $A \leftrightarrow B$ if there exists a bijective function from A to B.

Theorem 4.2 (Relationships between different types of functions)**.**
Let A, B and C be three (possibly infinite) sets. Then,

(a) $A \hookrightarrow B$ if and only if $B \twoheadrightarrow A$;

(b) if $A \hookrightarrow B$ and $B \hookrightarrow C$, then $A \hookrightarrow C$;

(c) $A \leftrightarrow B$ if and only if $A \hookrightarrow B$ and $B \hookrightarrow A$.

Exercise 4.3 (Exercise with injections and surjections)**.**
Prove parts (a) and (b) of the above theorem.

Definition 4.4 (Comparison of cardinality of sets)**.**
Let A and B be two (possibly infinite) sets.

- We write $|A| = |B|$ if $A \leftrightarrow B$.
- We write $|A| \leq |B|$ if $A \hookrightarrow B$, or equivalently, if $B \twoheadrightarrow A$.[1]
- We write $|A| < |B|$ if it is not the case that $|A| \geq |B|$.[2]

Note 4.5 (Sanity checks for comparing cardinality of sets)**.**
Theorem 4.2 (Relationships between different types of functions) justifies the use of the notation $=, \leq, \geq, <$ and $>$. The properties we would expect to hold for this type of notation indeed do hold. For example, $|A| \leq |B|$ and $|B| \leq |A|$ if and only if $|A| = |B|$. If $|A| \leq |B| \leq |C|$, then $|A| \leq |C|$. If $|A| \leq |B| < |C|$, then $|A| < |C|$, and so on.

Definition 4.6 (Countable and uncountable sets)**.**

- A set A is called *countable* if $|A| \leq |\mathbb{N}|$.
- A set A is called *countably infinite* if it is countable and infinite.
- A set A is called *uncountable* if it is not countable, i.e. $|A| > |\mathbb{N}|$.

[1] Even though not explicitly stated, $|B| \geq |A|$ has the same meaning as $|A| \leq |B|$.
[2] Similar to above, $|B| > |A|$ has the same meaning as $|A| < |B|$.

Theorem 4.7 (Characterization of countably infinite sets)**.**
A set A is countably infinite if and only if $|A| = |\mathbb{N}|$.

Exercise 4.8 (Proof of the characterization of countably infinite sets)**.**
Prove the above theorem.

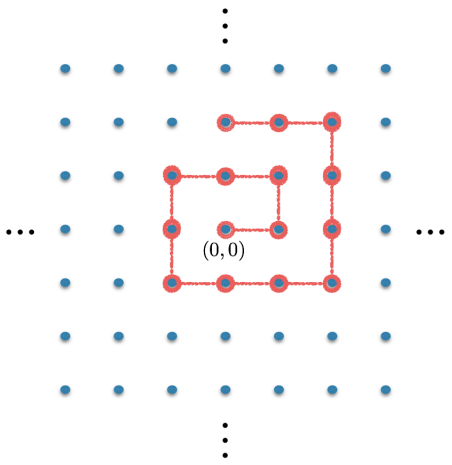
Note 4.9 (Only two options for countable sets)**.**
The above theorem implies that if A is countable, there are two options: either A is finite, or $|A| = |\mathbb{N}|$.

4.2 Countable Sets

Proposition 4.10 ($\mathbb{Z} \times \mathbb{Z}$ is countable)**.**
The set $\mathbb{Z} \times \mathbb{Z}$ is countable.

Proof. We want to show that $\mathbb{Z} \times \mathbb{Z}$ is countable. We will do so by listing all the elements of $\mathbb{Z} \times \mathbb{Z}$ such that every element eventually appears in the list. This implies that there is a surjective function f from $\mathbb{N}$ to $\mathbb{Z} \times \mathbb{Z}$: $f(i)$ is defined to be the i'th element in the list. Since there is a surjection from $\mathbb{N}$ to $\mathbb{Z} \times \mathbb{Z}$, $|\mathbb{Z} \times \mathbb{Z}| \leq |\mathbb{N}|$, and $\mathbb{Z} \times \mathbb{Z}$ is countable.[3]

We now describe how to list the elements of $\mathbb{Z} \times \mathbb{Z}$. Consider the plot of $\mathbb{Z} \times \mathbb{Z}$ on a 2-dimensional grid. Starting at $(0, 0)$ we list the elements of $\mathbb{Z} \times \mathbb{Z}$ using a spiral shape, as shown below.



(The picture shows only a small part of the spiral.) Since we have a way to list all the elements such that every element eventually appears in the list, we are done. □

Proposition 4.11 ($\mathbb{Q}$ is countable)**.**
The set of rational numbers $\mathbb{Q}$ is countable.

[3] Note that it is not a requirement that we give an explicit formula for $f(i)$. In fact, sometimes in such proofs, an explicit formula may not exist. This does not make the proof any less rigorous.
Also note that this proof highlights the fact that the notion of countable is equivalent to the notion of *listable*, which can be informally defined as the ability to list the elements of the set so that every element eventually appears in the list.

Proof. We want to show $\mathbb{Q}$ is countable. We will make use of the previous proposition to establish this. In particular, every element of $\mathbb{Q}$ can be written as a fraction a/b where $a, b \in \mathbb{Z}$. In other words, there is a surjection from $\mathbb{Z} \times \mathbb{Z}$ to $\mathbb{Q}$ that maps (a, b) to a/b (if $b = 0$, map (a, b) to say 0). This shows that $|\mathbb{Q}| \leq |\mathbb{Z} \times \mathbb{Z}|$. Since $\mathbb{Z} \times \mathbb{Z}$ is countable, i.e. $|\mathbb{Z} \times \mathbb{Z}| \leq |\mathbb{N}|$, $\mathbb{Q}$ is also countable, i.e. $|\mathbb{Q}| \leq |\mathbb{N}|$. □

Proposition 4.12 (Σ^* is countable)**.**
Let Σ be a finite set. Then Σ^* is countable.

Proof. Recall that Σ^* denotes the set of all words/strings over the alphabet Σ with finitely many symbols. We want to show Σ^* is countable. We will do so by presenting a way to list all the elements of Σ^* such that eventually all the elements appear in the list.

For each $n = 0, 1, 2, \ldots$, let Σ^n denote the set of words in Σ^* that have length exactly n. Note that Σ^n is a finite set for each n, and Σ^* is a union of these sets: $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$. This gives us a way to list the elements of Σ^* so that any element of Σ^* eventually appears in the list. First list the elements of Σ^0, then list the elements of Σ^1, then list the elements of Σ^2, and so on. This way of listing the elements gives us a surjective function f from $\mathbb{N}$ to Σ^*: $f(i)$ is defined to be the i'th element in the list. Since there is a surjection from $\mathbb{N}$ to Σ^*, $|\Sigma^*| \leq |\mathbb{N}|$, and Σ^* is countable. □

IMPORTANT 4.13 (The CS method of showing countability)**.**
One of the most powerful techniques for showing that a set A is countable is to show that A is encodable (i.e., there is an injective function Enc : $A \to \Sigma^*$ for some finite alphabet Σ). This is because if A is encodable, then $|A| \leq |\Sigma^*| \leq |\mathbb{N}|$. So if the set A is such that you can "write down" each element of A using a finite number of symbols, then A is countable. We call this method the "CS method" of showing countability.

Proposition 4.14 (The set of Turing machines is countable)**.**
The set of all Turing machines $\{M : M \text{ is a TM}\}$ is countable.

Proof. Let $T = \{M : M \text{ is a TM}\}$. We want to show that T is countable. We will do so by using the CS method of showing a set is countable.

Given any Turing machine, there is a way to encode it with a finite length string because each component of the 7-tuple has a finite description. In particular, the mapping $M \mapsto \langle M \rangle$, where $\langle M \rangle \in \Sigma^*$, for some finite alphabet Σ, is an injective map (two distinct Turing machines cannot have the same encoding). Therefore $|T| \leq |\Sigma^*|$. And since Σ^* is countable (Proposition 4.12 (Σ^* is countable)), i.e., $|\Sigma^*| \leq |\mathbb{N}|$, the result follows. □

Proposition 4.15 (The set of polynomials with rational coefficients is countable)**.**
The set of all polynomials in one variable with rational coefficients is countable.

Proof. Let $\mathbb{Q}[x]$ denote the set of all polynomials in one variable with rational coefficients. We want to show that $\mathbb{Q}[x]$ is countable and we will do so using the CS method. Let

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, /, x\}.$$

Then observe that every element of $\mathbb{Q}[x]$ can be written as a string over this alphabet. For example,

```
2x3 - 1/34x2 + 99/100x1 + 22/7
```

represents the polynomial

$$2x^3 - 1/34x^2 + 99/100x + 22/7.$$

This implies that there is a surjective map from Σ^* to $\mathbb{Q}[x]$. And therefore $|\mathbb{Q}[x]| \leq |\Sigma^*|$. Since Σ^* is countable, i.e. $|\Sigma^*| \leq |\mathbb{N}|$, $\mathbb{Q}[x]$ is also countable. □

Exercise 4.16 (Practice with countability proofs)**.**
Show that the following sets are countable.

(a) $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$.

(b) The set of all functions $f : A \to \mathbb{N}$, where A is a finite set.

4.3 Uncountable Sets

Theorem 4.17 (Cantor's Theorem)**.**
For any set A, $|\mathcal{P}(A)| > |A|$.

Proof. We want to show that for any (possibly infinite) set A, we have $|\mathcal{P}(A)| > |A|$. The proof that we present here is called the *diagonalization argument*. The proof is by contradiction. So assume that there is some set A such that $|\mathcal{P}(A)| \leq |A|$. By definition, this means that there is a surjective function from A to $\mathcal{P}(A)$. Let $f : A \to \mathcal{P}(A)$ be such a surjection. So for any $S \in \mathcal{P}(A)$, there exists an $s \in A$ such that $f(s) = S$. Now consider the set

$$S = \{a \in A : a \notin f(a)\}.$$

Since S is a subset of A, $S \in \mathcal{P}(A)$. So there is an $s \in A$ such that $f(s) = S$. But then if $s \notin S$, by the definition of S, s is in $f(s) = S$, which is a contradiction. If $s \in S$, then by the definition of S, s is not in $f(s) = S$, which is also a contradiction. So either way, we get a contradiction, as desired. □

Corollary 4.18 ($\mathcal{P}(\mathbb{N})$ is uncountable)**.**
The set $\mathcal{P}(\mathbb{N})$ is uncountable.

Corollary 4.19 (The set of languages is uncountable)**.**
Let Σ be a finite set with $|\Sigma| > 0$. Then $\mathcal{P}(\Sigma^*)$ is uncountable.

Proof. We want to show that $\mathcal{P}(\Sigma^*)$ is uncountable, where Σ is a non-empty finite set. For such a Σ, note that Σ^* is a countably infinite set (Proposition 4.12 (Σ^* is countable)). So by Theorem 4.7 (Characterization of countably infinite sets), we know $|\Sigma^*| = |\mathbb{N}|$. Theorem 4.17 (Cantor's Theorem) implies that $|\Sigma^*| < |\mathcal{P}(\Sigma^*)|$. So we have $|\mathbb{N}| = |\Sigma^*| < |\mathcal{P}(\Sigma^*)|$, which shows, by the definition of uncountable sets, that $\mathcal{P}(\Sigma^*)$ is uncountable. □

Definition 4.20 (Σ^∞)**.**
Let Σ be some finite alphabet. We denote by Σ^∞ the set of all *infinite* length words over the alphabet Σ. Note that $\Sigma^* \cap \Sigma^\infty = \emptyset$.

Theorem 4.21 ($\{0,1\}^\infty$ is uncountable)**.**
The set $\{0,1\}^\infty$ is uncountable.

Proof. Our goal is to show that $\{0,1\}^\infty$ is uncountable. One can prove this simply by observing that $\{0,1\}^\infty \leftrightarrow \mathcal{P}(\mathbb{N})$, and using Corollary 4.18 ($\mathcal{P}(\mathbb{N})$ is uncountable). Here, we will give a direct proof using a diagonalization argument. The proof is by contradiction, so assume that $\{0,1\}^\infty$ is countable. By definition, this means that $|\{0,1\}^\infty| \leq |\mathbb{N}|$, i.e. there is a surjective map f from $\mathbb{N}$ to $\{0,1\}^\infty$. Consider the table in which the i'th row corresponds to $f(i)$. Below is an example.

$f(1)$	$\boxed{0}$	0	0	0	0	$\cdots$
$f(2)$	1	$\boxed{1}$	1	1	1	$\cdots$
$f(3)$	0	1	$\boxed{0}$	1	0	$\cdots$
$f(4)$	1	0	1	$\boxed{0}$	1	$\cdots$
$f(5)$	0	0	1	1	$\boxed{0}$	$\cdots$
$\vdots$			$\vdots$			

(The elements in the diagonal are highlighted.) Using f, we construct an element a of $\{0,1\}^\infty$ as follows. If the i'th symbol of $f(i)$ is 1, then the i'th symbol of a is defined to be 0. And if the i'th symbol of $f(i)$ is 0, then the i'th symbol of a is defined to be 1. Notice that the i'th symbol of $f(i)$, for $i = 1, 2, 3, \ldots$ corresponds to the diagonal elements in the above table. So we are creating this element a of $\{0,1\}^\infty$ by taking the diagonal elements, and flipping their value.

Now notice that the way a is constructed implies that it cannot appear as a row in this table. This is because a differs from $f(1)$ in the first symbol, it differs from $f(2)$ in the second symbol, it differs from $f(3)$ in the third symbol, and so on. So it differs from every row of the table and hence cannot appear as a row in the table. This leads to the desired contradiction because f is a surjective function, which means every element of $\{0,1\}^\infty$, including a, *must* appear in the table. □

Exercise 4.22 (Uncountable sets are closed under supersets)**.**
Prove that if A is uncountable and $A \subseteq B$, then B is also uncountable.

IMPORTANT 4.23 (Uncountability through $\{0,1\}^\infty$)**.**
One of the most powerful techniques for showing that a set A is uncountable is to show that $|A| \geq |\{0,1\}^\infty|$, i.e. there is a surjection from A to $\{0,1\}^\infty$, or equivalently, there is an injection from $\{0,1\}^\infty$ to A. One strategy for establishing this is to identify a subset of A that is in one-to-one correspondence with $\{0,1\}^\infty$.

Exercise 4.24 (Practice with uncountability proofs)**.**
Show that the following sets are uncountable.

(a) The set of all bijective functions from $\mathbb{N}$ to $\mathbb{N}$.

(b) $\{x_1x_2x_3\ldots \in \{1,2\}^\infty : \text{ for all } n \geq 1,\ \sum_{i=1}^{n} x_i \not\equiv 0 \mod 4\}$

Quiz

1. True or false: $\{0,1\}^* \cap \{0,1\}^\infty = \emptyset$.
2. True of false: $|\{0,1,2\}^*| = |\mathbb{Q} \times \mathbb{Q}|$.
3. True of false: $|\mathcal{P}(\{0,1\}^\infty)| = |\mathcal{P}(\mathcal{P}(\{0,1\}^\infty))|$.
4. True of false: The set of all non-regular languages is countable.
5. True or false: There is a surjection from $\{0,1\}^\infty$ to $\{0,1,2,3\}^\infty$.

Hints to Selected Exercises

Exercise 4.8 (Proof of the characterization of countably infinite sets):
One of the directions should be relatively straightforward. For the other direction, given A which is countable and infinite, try to find a way to order the elements of A. Then the bijection with $\mathbb{N}$ can be: n maps to the n'th element of A in the defined order.

Exercise 4.16 (Practice with countability proofs):
Use the CS method for both parts.

Exercise 4.24 (Practice with uncountability proofs):
In both cases, try to identify a subset of the set that is in one-to-one correspondence with $\{0, 1\}^\infty$.

Chapter 5

Undecidable Languages

PREAMBLE

Chapter structure:

- Section 5.1 (Existence of Undecidable Languages)
  - Theorem 5.1 (Almost all languages are undecidable)
- Section 5.2 (Examples of Undecidable Languages)
  - Definition 5.3 (Halting problem)
  - Theorem 5.4 (Turing's Theorem)
  - Definition 5.6 (Languages related to encodings of TMs)
  - Theorem 5.7 ($\mathrm{ACCEPTS}$ is undecidable)
  - Theorem 5.8 (EMPTY is undecidable)
  - Theorem 5.10 (EQ is undecidable)
- Section 5.3 (Undecidability Proofs by Reductions)
  - Theorem 5.15 ($\mathrm{HALTS} \leq \mathrm{EMPTY}$)
  - Theorem 5.16 ($\mathrm{EMPTY} \leq \mathrm{HALTS}$)

Chapter goals:

In this chapter, we formally prove that almost all languages are undecidable using the countability and uncountability concepts from the previous chapter. We also present (with proofs) several explicit examples of undecidable languages. By the Church-Turing Thesis, these results highlight the inherint limitations of computation.

An important tool in showing that a language is undecidable is the concept of a *reduction*. We present this technique in this chapter. Reductions play an extremely important role in computer science. In fact, we will revisit them in a future chapter (in the context of the famous P vs NP problem.)

Our goal in this chapter is for you to get comfortable with undecidability proofs and the concept of reductions, as they are at the core of the study of computation.

5.1 Existence of Undecidable Languages

Theorem 5.1 (Almost all languages are undecidable)**.**
Fix some alphabet Σ. There are languages $L \subseteq \Sigma^*$ that are not decidable.

Proof. To prove the result, we simply observe that the set of all languages is uncountable whereas the set of decidable languages is countable. First, consider the set of all languages. Since a language L is defined to be a subset of Σ^*, the set of all languages is $\mathcal{P}(\Sigma^*)$. By Corollary 4.19 (The set of languages is uncountable), we know that this set is uncountable. Now consider the set of all decidable languages, which we'll denote by D. Let T be the set of all TMs. By Proposition 4.14 (The set of Turing machines is countable), we know that T is countable. Furthermore, the mapping $M \mapsto L(M)$ can be viewed as a surjection from T to D (if M is not a decider, just map it to $\emptyset$). So $|D| \leq |T|$. Since T is countable, this shows D is countable and completes the proof. □

Note 5.2 (Constructive vs non-constructive proofs)**.**
The argument above is called *non-constructive* because it does not present an explicit undecidable language. A *constructive* argument would prove the undecidability of an explicit language. We present such an argument below (Theorem 5.4 (Turing's Theorem)).

5.2 Examples of Undecidable Languages

Definition 5.3 (Halting problem)**.**
The *halting problem* is defined as the decision problem corresponding to the language HALTS $= \{\langle M, x\rangle : M \text{ is a TM which halts on input } x\}$.

Theorem 5.4 (Turing's Theorem)**.**
The language HALTS *is undecidable.*

Proof. Our goal is to show that HALTS is undecidable. The proof is by contradiction, so assume that HALTS is decidable. By definition, this means that there is a decider TM, call it M_{HALTS}, that decides HALTS. We construct a new TM, which we'll call M_{TURING}, that uses M_{HALTS} as a subroutine. The description of M_{TURING} is as follows:

> M: TM.
> $M_{\text{TURING}}(\langle M\rangle)$:
> 1 Run $M_{\text{HALTS}}(\langle M, M\rangle)$.
> 2 If it accepts, go into an infinite loop.
> 3 If it rejects, accept.

We get the desired contradiction once we consider what happens when we feed M_{TURING} as input to itself, i.e. when we run $M_{\text{TURING}}(\langle M_{\text{TURING}}\rangle)$.

If $M_{\text{HALTS}}(\langle M_{\text{TURING}}, M_{\text{TURING}}\rangle)$ accepts, then $M_{\text{TURING}}(\langle M_{\text{TURING}}\rangle)$ is supposed to halt by the definition of M_{HALTS}. However, from the description of M_{TURING} above, we see that it goes into an infinite loop. This is a contradiction. The other option is that $M_{\text{HALTS}}(\langle M_{\text{TURING}}, M_{\text{TURING}}\rangle)$ rejects. Then $M_{\text{TURING}}(\langle M_{\text{TURING}}\rangle)$ is supposed to lead to an infinite loop. But from the description of M_{TURING} above, we see that it accepts, and therefore halts. This is a contradiction as well. □

Note 5.5 (Diagonalization argument for undecidability)**.**
The above proof is called a *diagonalization argument* as it is very similar to the proof of Cantor's theorem (Theorem 4.17 (Cantor's Theorem)). As in the proof of Theorem 4.21 ($\{0,1\}^\infty$ is uncountable), we can present the above proof using a table and flipping its diagonal elements to get the desired contradiction. We do so below.

Reproof: The proof is by contradiction, so assume that HALTS is decidable. By definition, this means that there is a decider TM, call it M_{HALTS}, that decides HALTS.

The set of all Turing machines is countable (Proposition 4.14 (The set of Turing machines is countable)). Let $M_1, M_2, \ldots$ be a listing of **all** Turing machines in some arbitrary order. We now consider a table in which row i corresponds to M_i and column i corresponds to $\langle M_i \rangle$. At entry corresponding to row i and column j, we indicate whether $M_i(\langle M_j \rangle)$ halts or loops forever. If it loops forever, we put a ∞ symbol, and if it halts, we put H.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	$\boxed{\infty}$	∞	H	∞	
M_2	H	$\boxed{H}$	H	∞	
M_3	∞	∞	$\boxed{H}$	H	$\cdots$
M_4	∞	H	H	$\boxed{\infty}$	
$\vdots$		$\vdots$			

We now create a new row in this table by taking the diagonal elements of the table and flipping their value (an ∞ is flipped to an H, and an H is flipped to an ∞). Notice that M_{TURING} constructed in the previous proof corresponds exactly to this new row we have created. We are able to construct M_{TURING} (and therefore the row it corresponds to) because we have a decider for HALTS. The contradiction is reached because on the one hand, M_{TURING} should appear as a row in the table since all the Turing machines are listed. On the other hand, the row of M_{TURING} differs from every row in the table (by construction, it differs from row i in the i'th column), and therefore cannot be in the table.

Definition 5.6 (Languages related to encodings of TMs)**.**
We define the following languages:

$$\text{ACCEPTS} = \{\langle M, x \rangle : M \text{ is a TM that accepts the input } x\},$$

$$\text{EMPTY} = \{\langle M \rangle : M \text{ is a TM with } L(M) = \emptyset\},$$

$$\text{EQ} = \{\langle M_1, M_2 \rangle : M_1 \text{ and } M_2 \text{ are TMs with } L(M_1) = L(M_2)\}.$$

Theorem 5.7 (ACCEPTS is undecidable)**.**
The language ACCEPTS *is undecidable.*

Proof. We want to show that ACCEPTS is undecidable. The proof is by contradiction, so assume ACCEPTS is decidable and let M_{ACCEPTS} be a decider for it. We will use this decider to come up with a decider for HALTS. Since HALTS is undecidable (Theorem 5.4 (Turing's Theorem)), this argument will allow us to reach a contradiction.

Here is our decider for HALTS:

M: TM. x: string.
$M_{\text{HALTS}}(\langle M, x\rangle)$:
1 Run $M_{\text{ACCEPTS}}(\langle M, x\rangle)$.
2 If it accepts, accept.
3 Construct string $\langle M'\rangle$ by flipping the accept and reject states of $\langle M\rangle$.
4 Run $M_{\text{ACCEPTS}}(\langle M', x\rangle)$.
5 If it accepts, accept.
6 If it rejects, reject.

We now argue that this machine indeed decides HALTS. To do this, we'll show that no matter what input is given to our machine, it always gives the correct answer.

First let's assume we get any input $\langle M, x\rangle$ such that $\langle M, x\rangle \in$ HALTS. In this case our machine is supposed to accept. Since $M(x)$ halts, we know that $M(x)$ either ends up in the accepting state, or it ends up in the rejecting state. If it ends up in the accepting state, then $M_{\text{ACCEPTS}}(\langle M, x\rangle)$ accepts (on line 1 of our machine's description), and so our program accepts and gives the correct answer on line 2. If on the other hand, $M(x)$ ends up in the rejecting state, then $M'(x)$ ends up in the accepting state. Therefore $M_{\text{ACCEPTS}}(\langle M', x\rangle)$ accepts (on line 4 of our machine's description), and so our program accepts and gives the correct answer on line 5.

Now let's assume we get any input $\langle M, x\rangle$ such that $\langle M, x\rangle \notin$ HALTS. In this case our machine is supposed to reject. Since $M(x)$ does not halt, it never reaches the accepting or the rejecting state. By the construction of M', this also implies that $M'(x)$ never reaches the accepting or the rejecting state. Therefore first $M_{\text{ACCEPTS}}(\langle M, x\rangle)$ (on line 1 of our machine's description) will reject. And then $M_{\text{ACCEPTS}}(\langle M', x\rangle)$ (on line 4 of our machine's description) will reject. Thus our program will reject as well, and give the correct answer on line 6.

We have shown that no matter what the input is, our machine gives the correct answer and decides HALTS. This is the desired contradiction and we conclude that ACCEPTS is undecidable. □

Theorem 5.8 (EMPTY is undecidable)**.**
The language EMPTY *is undecidable.*

Proof. We want to show that EMPTY is undecidable. The proof is by contradiction, so suppose EMPTY is decidable, and let M_{EMPTY} be a decider for it. Using this decider, we will construct a decider for ACCEPTS. However, we know that ACCEPTS is undecidable (Theorem 5.7 (ACCEPTS is undecidable)), so this argument will allow us to reach a contradiction.

We construct a TM that decides ACCEPTS as follows.

M: TM. x: string.
$M_{\text{ACCEPTS}}(\langle M, x\rangle)$:
1 Construct the following string, which we call $\langle M'\rangle$.
2 "$M'(y)$:
3 Run $M(x)$.
4 If it accepts, accept.
5 If it rejects, reject."
6 Run $M_{\text{EMPTY}}(\langle M'\rangle)$.

7 If it accepts, reject.
8 If it rejects, accept.

We now argue that this machine indeed decides ACCEPTS. To do this, we'll show that no matter what input is given to our machine, it always gives the correct answer.

First let's assume we get an input $\langle M, x\rangle$ such that $\langle M, x\rangle \in$ ACCEPTS, i.e. $x \in L(M)$. Then observe that $L(M') = \Sigma^*$, because for any input y, $M'(y)$ will accept. When we run $M_{\text{EMPTY}}(\langle M'\rangle)$ on line 6, it rejects, and so our machine accepts and gives the correct answer.

Now assume that we get an input $\langle M, x\rangle$ such that $\langle M, x\rangle \notin$ ACCEPTS, i.e. $x \notin L(M)$. Then either $M(x)$ rejects, or loops forever. If it rejects, then $M'(y)$ rejects for any y. If it loops forever, then $M'(y)$ gets stuck on line 3 for any y. In both cases, $L(M') = \emptyset$. When we run $M_{\text{EMPTY}}(\langle M'\rangle)$ on line 6, it accepts, and so our machine rejects and gives the correct answer.

Our machine always gives the correct answer, so we are done. □

Note 5.9 (Creating an encoding of a machine vs running it)**.**
In the proof above, we have defined the decider M_{ACCEPTS} in which we create the encoding of the machine M', denoted $\langle M'\rangle$. Note that creating $\langle M'\rangle$ is very different from actually running the machine M'. In particular, even if $M(x)$ loops forever, $M_{\text{ACCEPTS}}(\langle M, x\rangle)$ does **not** loop forever because

(i) M_{ACCEPTS} does not run M', and

(ii) M_{EMPTY} is assumed to be a decider, which means it always halts.

Theorem 5.10 (EQ is undecidable)**.**
The language EQ *is undecidable.*

Proof. The proof is by contradiction, so assume EQ is decidable, and let M_{EQ} be a decider for it. Using this decider, we will construct a decider for EMPTY. However, EMPTY is undecidable (Theorem 5.8 (EMPTY is undecidable)), so this argument allows us to reach the desired contradiction.

We construct a TM that decides EMPTY as follows.

M: TM.
$M_{\text{EMPTY}}(\langle M\rangle)$:
1 Construct the string $\langle M'\rangle$ where M' is a TM that rejects every input.
2 Run $M_{\text{EQ}}(\langle M, M'\rangle)$.
3 If it accepts, accept.
4 If it rejects, reject.

It is not difficult to see that this machine indeed decides EMPTY. Notice that $L(M') = \emptyset$. So when we run $M_{\text{EQ}}(\langle M, M'\rangle)$ on line 2, we are deciding whether $L(M) = L(M')$, i.e. whether $L(M) = \emptyset$. □

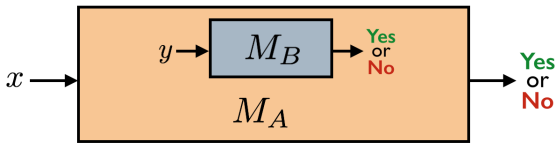
Exercise 5.11 (Practice with undecidability proofs)**.**
Show that the following languages are undecidable.

(a) EMPTY-HALTS $= \{\langle M\rangle : M \text{ is a TM and } M(\epsilon) \text{ halts}\}$.

(b) FINITE $= \{\langle M\rangle : M \text{ is a TM that accepts finitely many strings.}\}$

5.3 Undecidability Proofs by Reductions

IMPORTANT 5.12 (Undecidability proofs by reduction)**.**
In the last section, we have used the same proof technique over and over again. It will be convenient to abstract away this technique and give it a name. Fix some alphabet Σ. Let A and B be two languages. We say that A *reduces* to B, written $A \leq B$, if we are able to do the following: assume B is decidable (for the sake of argument), and then show that A is decidable by using the decider for B as a black-box subroutine. Here the languages A and B may or may not be decidable to begin with. But observe that if $A \leq B$ and B is decidable, then A is also decidable. Equivalently, taking the contrapositive, if $A \leq B$ and A is undecidable, then B is also undecidable. So when $A \leq B$, we think of B as being at least as hard as A with respect to decidability (which justifies using the less-than-or-equal-to sign).



Note 5.13 (Turing reductions)**.**
In the literature, the above idea is formalized using the notion of a *Turing reduction* (with the corresponding symbol $\leq_T$). In order to define it formally, we need to define Turing machines that have access to an *oracle*. This level of detail will not be important for us, so we choose to omit the formal definition in our notes.

Note 5.14 (Already established reductions)**.**
The proofs of Theorem 5.7 (ACCEPTS is undecidable), Theorem 5.8 (EMPTY is undecidable), and Theorem 5.10 (EQ is undecidable) correspond to HALTS $\leq$ ACCEPTS, ACCEPTS $\leq$ EMPTY and EMPTY $\leq$ EQ respectively.

Theorem 5.15 (HALTS $\leq$ EMPTY)**.**
HALTS $\leq$ EMPTY.

Proof. (This can be considered as an alternative proof of Theorem 5.8 (EMPTY is undecidable).) We want to show that deciding HALTS reduces to deciding EMPTY. For this, we assume EMPTY is decidable. Let M_{EMPTY} be a decider for EMPTY. We need to construct a TM that decides HALTS. We do so now.

M: TM. x: string.
$M_{\text{HALTS}}(\langle M, x\rangle)$:
1 Construct the following string, which we call $\langle M'\rangle$.
2 "$M'(y)$:
3 Run $M(x)$.
4 Ignore the output and accept."
5 Run $M_{\text{EMPTY}}(\langle M'\rangle)$.
6 If it accepts, reject.
7 If it rejects, accept.

We now argue that this machine indeed decides HALTS. First consider an input $\langle M, x\rangle$ such that $\langle M, x\rangle \in$ HALTS. Then $L(M') = \Sigma^*$ since in this case M' accepts every string. So when we run $M_{\text{EMPTY}}(\langle M'\rangle)$ on line 8, it rejects, and our machine accepts and gives the correct answer.

Now consider an input $\langle M, x\rangle$ such that $\langle M, x\rangle \notin$ HALTS. Then notice that whatever input is given to M', it gets stuck in an infinite loop when it runs $M(x)$. Therefore $L(M') = \emptyset$. So when we run $M_{\text{EMPTY}}(\langle M'\rangle)$ on line 8, it accepts, and our machine rejects and gives the correct answer. □

Theorem 5.16 (EMPTY $\leq$ HALTS)**.**
EMPTY $\leq$ HALTS.

Proof. We want to show that deciding EMPTY reduces to deciding HALTS. For this, we assume HALTS is decidable. Let M_{HALTS} be a decider for HALTS. Using it, we need to construct a decider for EMPTY. We do so now.

```
M: TM.
M_EMPTY(⟨M⟩):
1 Construct the following string, which we call ⟨M'⟩.
2 "M'(x):
3   For t = 1, 2, 3, ...:
4     For each y with |y| ≤ t:
5       Simulate M(y) for at most t steps.
6       If it accepts, accept."
7 Run M_HALTS(⟨M', ϵ⟩).
8 If it accepts, reject.
9 If it rejects, accept.
```

We now argue that this machine indeed decides EMPTY. First consider an input $\langle M\rangle$ such that $\langle M\rangle \in$ EMPTY. Observe that the only way M' halts is if $M(y)$ accepts for some string y. This cannot happen since $L(M) = \emptyset$. So $M'(x)$, for *any* x, does not halt (note that M' ignores its input). This means that when we run $M_{\text{HALTS}}(\langle M', \epsilon\rangle)$, it rejects, and so our decider above accepts, as desired.

Now consider an input $\langle M\rangle$ such that $\langle M\rangle \notin$ EMPTY. This means that there is some word y such that $M(y)$ accepts. Note that M', by construction, does an exhaustive search, so if such a y exists, then M' will eventually find it, and accept. So $M'(x)$ halts for any x. When we run $M_{\text{HALTS}}(\langle M', \epsilon\rangle)$, it accepts, and our machine rejects and gives the correct answer. □

Exercise 5.17 (Practice with reduction definition)**.**
Let $A, B \subseteq \{0, 1\}^*$ be languages. Prove or disprove the following claims.

(a) If $A \leq B$ then $B \leq A$.

(b) If $A \leq B$ and B is regular, then A is regular.

Exercise 5.18 (Practice with reduction proofs)**.**
Show the following.

(a) ACCEPTS $\leq$ HALTS.

(b) HALTS $\leq$ EQ.

Quiz

1. True or false: For languages K and L, if $K \leq L$, then L is undecidable.
2. True or false: The set of undecidable languages is countable.
3. True or false: If a language L is undecidable, then L is infinite.
4. True or false: $\Sigma^* \leq \emptyset$.
5. True or false: HALTS $\leq \Sigma^*$.

Hints to Selected Exercises

Exercise 5.11 (Practice with undecidability proofs):
As usual, the proofs will be by contradiction. In both cases, show how to decide HALTS given a decider for the language in question.

Exercise 5.17 (Practice with reduction definition):
Both parts are false.

Chapter 6

Time Complexity

PREAMBLE

Chapter structure:

- Section 6.1 (Big-O, Big-Omega and Theta)
  - Definition 6.1 (Big-O)
  - Definition 6.3 (Big-Omega)
  - Definition 6.5 (Theta)
  - Proposition 6.6 (Logarithms in different bases)
- Section 6.2 (Worst-Case Running Time of Algorithms)
  - Definition 6.9 (Worst-case running time of an algorithm)
  - Definition 6.13 (Names for common growth rates)
  - Proposition 6.16 (Intrinsic complexity of $\{0^k 1^k : k \in \mathbb{N}\}$)
- Section 6.3 (Complexity of Algorithms with Integer Inputs)
  - Definition 6.20 (Integer addition and integer multiplication problems)
  - Theorem 6.24 (Karatsuba algorithm for integer multiplication)

Chapter goals:

So far, we have formally defined what a computational/decision problem is, what an algorithm is, and saw that most (decision) problems are undecidable. We also saw some explicit and interesting examples of undecidable problems. Nevertheless, it turns out that many problems that we care about are actually decidable. So the next natural thing to study is the computational complexity of problems. If a problem is decidable, but the most efficient algorithm solving it takes vigintillion computational steps even for reasonably sized inputs, then practically speaking, that problem is still undecidable.

Our goal in this chapter is to introduce the right language to express and analyze the running time of algorithms, which in return helps us determine which problems are practically decidable, and which problems are (or seem to be) practically undecidable.

Most of the ideas in this chapter will probably be familiar to you at some level as most introductory computer science courses do talk about (asymptotic) running time of algorithms. Nevertheless, it is important to specify some of the details that we will be using as we slowly start approaching the famous P vs NP question, which is a question about the computational complexity of problems.

6.1 Big-O, Big-Omega and Theta

Definition 6.1 (Big-O)**.**
For $f : \mathbb{R}^+ \to \mathbb{R}^+$ and $g : \mathbb{R}^+ \to \mathbb{R}^+$, we write $f(n) = O(g(n))$ if there exist constants $C > 0$ and $n_0 > 0$ such that for all $n \geq n_0$,

$$f(n) \leq Cg(n).$$

In this case, we say that $f(n)$ *is big-O of* $g(n)$.

Exercise 6.2 (Practice with big-O)**.**
Show that $3n^2 + 10n + 30$ is $O(n^2)$.

Definition 6.3 (Big-Omega)**.**
For $f : \mathbb{R}^+ \to \mathbb{R}^+$ and $g : \mathbb{R}^+ \to \mathbb{R}^+$, we write $f(n) = \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0$,

$$f(n) \geq cg(n).$$

In this case, we say that $f(n)$ *is big-Omega of* $g(n)$.

Exercise 6.4 (Practice with big-Omega)**.**
Show that $n!$[2] is $\Omega(n^n)$.

Definition 6.5 (Theta)**.**
For $f : \mathbb{R}^+ \to \mathbb{R}^+$ and $g : \mathbb{R}^+ \to \mathbb{R}^+$, we write $f(n) = \Theta(g(n))$ if

$$f(n) = O(g(n)) \qquad \text{and} \qquad f(n) = \Omega(g(n)).$$

This is equivalent to saying that there exists constants $c, C, n_0 > 0$ such that for all $n \geq n_0$,

$$cg(n) \leq f(n) \leq Cg(n).$$

In this case, we say that $f(n)$ *is Theta of* $g(n)$.[1]

Proposition 6.6 (Logarithms in different bases)**.**
For any constant $b > 1$*,*

$$\log_b n = \Theta(\log n).$$

Proof. It is well known that $\log_b n = \frac{\log_a n}{\log_a b}$. In particular $\log_b n = \frac{\log_2 n}{\log_2 b}$. Then taking $c = C = \frac{1}{\log_2 b}$ and $n_0 = 1$, we see that $c \log_2 n \leq \log_b n \leq C \log_2 n$ for all $n \geq n_0$. Therefore $\log_b n = \Theta(\log_2 n)$. □

Note 6.7 (Does the base of a logarithm matter?)**.**
Since the base of a logarithm only changes the value of the log function by a constant factor, it is usually not relevant in big-O, big-Omega or Theta notation. So most of the time, when you see a $\log$ function present inside $O(\cdot)$, $\Omega(\cdot)$, or $\Theta(\cdot)$, the base will be ignored. E.g. instead of writing $\ln n = \Theta(\log_2 n)$, we actually write $\ln n = \Theta(\log n)$. That being said, if the $\log$ appears in the exponent, the base matters. For example, $n^{\log_2 5}$ is asymptotically different from $n^{\log_3 5}$.

Exercise 6.8 (Practice with Theta)**.**
Show that $\log_2(n!) = \Theta(n \log n)$.

[1] The reason we don't call it big-Theta is that there is no separate notion of little-theta, whereas little-o $o(\cdot)$ and little-omega $\omega(\cdot)$ have meanings separate from big-O and big-Omega. We don't cover little-o and little-omega in this course.

6.2 Worst-Case Running Time of Algorithms

Definition 6.9 (Worst-case running time of an algorithm)**.**
Suppose we are using some computational model in which what constitutes a step in an algorithm is understood. Suppose also that for any input x, we have an explicit definition of its length. The *worst-case running time* of an algorithm A is a function $T_A : \mathbb{N} \to \mathbb{N}$ defined by

$$T_A(n) = \max_{\substack{\text{instances/inputs } x \\ \text{of length } n}} \text{number of steps } A \text{ takes on input } x.$$

We drop the subscript A and just write $T(n)$ when A is clear from the context.

IMPORTANT 6.10 (Input length)**.**
We use n to denote the input length. Unless specified otherwise, n is defined to be the number of bits in a reasonable binary encoding of the input. It is also common to define n in other ways. For example, if the input is an array or a list, n can denote the number of elements.

IMPORTANT 6.11 (Our model when measuring running time)**.**
In the Turing machine model, a step in the computation corresponds to one application of the transition function of the machine. However, when measuring running time, often we will not be considering the Turing machine model.

If we don't specify a particular computational model, by default, our model will be closely related to the Random Access Machine (RAM) model. Compared to TMs, this model aligns better with the architecture of the computers we use today. We will not define this model formally, but instead point out two properties of importance.

First, given a string or an array, accessing any index counts as 1 step.

Second, arithmetic operations count as 1 step as long as the numbers involved are "small". We say that a number y is *small* if it can be upper bounded by a polynomial in n, the input length. That is, y is small if there is some constant k such that y is $O(n^k)$. As an example, suppose we have an algorithm A that contains a line like $x = y + z$, where y and z are variables that hold integer values. Then we can count this line as a single step if y and z are both small. Note that whether a number is small or not is determined by the length of the input to the algorithm A.

We say that a number is *large*, if it is not small, i.e., if it cannot be upper bounded by a polynomial in n. In cases where we are doing arithmetic operations involving large numbers, we have to consider the algorithms used for the arithmetic operations and figure out their running time. For example, in the line $x = y+z$, if y or z is a large number, we need to specify what algorithm is being used to do the addition and what its running time is. A large number should be treated as a string of digits/characters. Arithmetic operations on large numbers should be treated as string manipulation operations and their running time should be figured out accordingly.

Note 6.12 (Asymptotic complexity)**.**
The expression of the running time of an algorithm using big-O, big-Omega or Theta notation is referred to as *asymptotic complexity* estimate of the algorithm.

Definition 6.13 (Names for common growth rates)**.**

Constant time:	$T(n) = O(1)$.
Logarithmic time:	$T(n) = O(\log n)$.
Linear time:	$T(n) = O(n)$.
Quadratic time:	$T(n) = O(n^2)$.
Polynomial time:	$T(n) = O(n^k)$ for some constant $k > 0$.
Exponential time:	$T(n) = O(2^{n^k})$ for some constant $k > 0$.

Exercise 6.14 (Composing polynomial time algorithms)**.**
Suppose that we have an algorithm A that runs another algorithm A' once as a subroutine. We know that the running time of A' is $O(n^k)$, $k \geq 1$, and the work done by A is $O(n^t)$, $t \geq 1$, if we ignore the subroutine A' (i.e., we don't count the steps taken by A'). What kind of upper bound can we give for the total running-time of A (which includes the work done by A')?

Note 6.15 (Intrinsic complexity)**.**
The *intrinsic complexity* of a computational problem refers to the asymptotic time complexity of the most efficient algorithm that computes the problem.[2]

Proposition 6.16 (Intrinsic complexity of $\{0^k1^k : k \in \mathbb{N}\}$)**.**
The intrinsic complexity of $L = \{0^k1^k : k \in \mathbb{N}\}$ *is* $\Theta(n)$.

Proof. We want to show that the intrinsic complexity of $L = \{0^k1^k : k \in \mathbb{N}\}$ is $\Theta(n)$. The proof has two parts. First, we need to argue that the intrinsic complexity is $O(n)$. Then, we need to argue that the intrinsic complexity is $\Omega(n)$.

To show that L has intrinsic complexity $O(n)$, all we need to do is present an algorithm that decides L in time $O(n)$. We leave this as an exercise to the reader.

To show that L has intrinsic complexity $\Omega(n)$, we show that no matter what algorithm is used to decide L, the number of steps it takes must be at least n. We prove this by contradiction, so assume that there is some algorithm A that decides L using $n-1$ steps or less. Consider the input $x = 0^k1^k$ (where $n = 2k$). Since A uses at most $n - 1$ steps, there is at least one index j with the property that A does not access $x[j]$. Let x' be the input that is the same as x, except the j'th coordinate is reversed. Since A does not access the j'th coordinate, it has no way of distinguishing between x and x'. In other words, A behaves exactly the same when the input is x or x'. But this contradicts the assumption that A correctly decides L because A should accept x and reject x'. □

Exercise 6.17 (TM complexity of $\{0^k1^k : k \in \mathbb{N}\}$)**.**
In the TM model, a *step* corresponds to one application of the transition function. Show that $L = \{0^k1^k : k \in \mathbb{N}\}$ can be decided by a TM in time $O(n \log n)$. Is this statement directly implied by Proposition 6.16 (Intrinsic complexity of $\{0^k1^k : k \in \mathbb{N}\}$)?

Exercise 6.18 (Is polynomial time decidability closed under concatenation?)**.**
Assume the languages L_1 and L_2 are decidable in polynomial time. Prove or give a counter-example: L_1L_2 is decidable in polynomial time.

[2] For certain computational problems, the intrinsic complexity may not be well-defined. In some cases, there can be a sequence of algorithms that solve a certain computational problem, where each algorithm in the sequence is asymptotically more efficient than the one before.

6.3 Complexity of Algorithms with Integer Inputs

IMPORTANT 6.19 (Integer inputs are large numbers)**.**
Given a computational problem with an integer input x, notice that x is a large number (if x is n bits long, its value can be about 2^n, so it cannot be upper bounded by a polynomial in n). Therefore arithmetic operations involving x cannot be treated as 1-step operations. Computational problems with integer input(s) are the most common examples in which we have to deal with large numbers, and in these situations, one should be particularly careful about analyzing running time.

Definition 6.20 (Integer addition and integer multiplication problems)**.**
In the *integer addition problem*, we are given two n-bit numbers x and y, and the output is their sum $x + y$. In the *integer multiplication problem*, we are given two n-bit numbers x and y, and the output is their product xy.

Note 6.21 (Algorithms for integer addition)**.**
Consider the following algorithm for the integer addition problem (we'll assume the inputs are natural numbers for simplicity).

```
x: natural number. y: natural number.
Addition(⟨x, y⟩):
1 For i = 1 to x:
2     y = y + 1.
3 Return y.
```

This algorithm has a loop that repeats x many times. Since x is an n-bit number, the worst-case complexity of this algorithm is $\Omega(2^n)$.

In comparison, the following well-known algorithm for integer addition has time complexity $O(n)$.

```
x: natural number. y: natural number.
Addition(⟨x, y⟩):
1 carry = 0.
2 For i = 0 to n − 1:
3     columnSum = x[i] + y[i] + carry.
4     z[i] = columnSum%2.
5     carry = (columnSum − z[i])/2.
6 z[n] = carry.
7 Return z.
```

Note that the arithmetic operations inside the loop are all $O(1)$ time since the numbers involved are all bounded (i.e., their values do not depend on n). Since the loop repeats n times, the overall complexity is $O(n)$.

It is easy to see that the intrinsic complexity of integer addition is $\Omega(n)$ since it takes at least n steps to write down the output, which is either n or $n + 1$ bits long. Therefore we can conclude that the intrinsic complexity of integer addition is $\Theta(n)$. The same is true for integer subtraction.

Exercise 6.22 (Running time of the factoring problem)**.**
Consider the following problem: Given as input a positive integer N, output a non-trivial factor[3] of N if one exists, and output False otherwise. Give a lower bound using the $\Omega(\cdot)$ notation for the running-time of the following algorithm solving the problem:

```
N: natural number.
Non-Trivial-Factor(⟨N⟩):
1 For i = 2 to N − 1:
2     If N%i == 0: Return i.
3 Return False.
```

Note 6.23 (Grade-school algorithms for multiplication and division)**.**
The grade-school algorithms for the integer multiplication and division problems have time complexity $O(n^2)$.

Theorem 6.24 (Karatsuba algorithm for integer multiplication)**.**
The integer multiplication problem can be solved in time $O(n^{1.59})$.

Proof. Our goal is to present an algorithm with running time $O(n^{1.59})$ that solves the integer multiplication problem where the input is two n-bit numbers x and y. For simplicity, we assume that n is a power of 2. The argument can be adapted to handle other values of n. We first present the algorithm (which is recursive), and then put an upper bound on its running time complexity.

Let x and y be the input integers, each n bits long. Observe that we can write x as $a \cdot 2^{n/2} + b$, where a is the number corresponding to the left $n/2$ bits of x and b is the number corresponding to the right $n/2$ bits of x. For example, if $x = 1011$ then $a = 10$ and $b = 11$. We can similarly write y as $c \cdot 2^{n/2} + d$. Then the product of x and y can be written as:

$$\begin{aligned} x \cdot y &= (a \cdot 2^{n/2} + b)(c \cdot 2^{n/2} + d) \\ &= ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd \end{aligned}$$

This implies that we can compute xy by recursively computing the products ac, ad, bc and bd. Note that multiplying a number by 2^n is just putting n 0's at the end of the number, so this has cost $O(n)$. Similarly all the addition operations are $O(n)$ time.

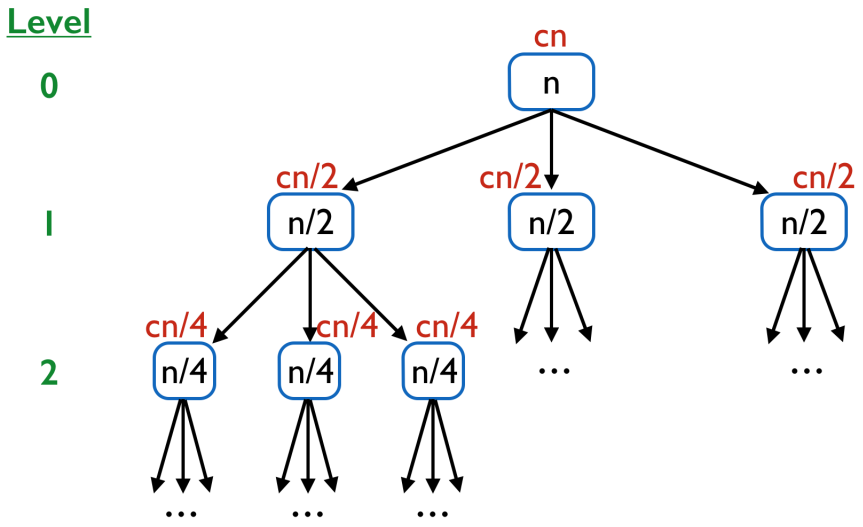
Our algorithm is slightly different than the one described above. We will do the calculation with only 3 recursive calls rather than 4. We recursively compute the products ac, bd and $(a-b)(c-d)$. Observe that $(a-b)(c-d) - ac - bd = -(ad+bc)$. Now that we have ac, bd and $(ad+bc)$ at hand, we can compute xy (of course we still have to do the appropriate addition operations and padding with 0's). This completes the description of the algorithm.

Let $T(n)$ be the time complexity of our algorithm. Observe that the recursive relation that $T(n)$ satisfies is

$$T(1) = k, \qquad T(n) \leq 3T(n/2) + cn \quad \text{for } n > 1,$$

where k and c are some constants. The base case corresponds to 1-bit integers, which can be calculated in constant time. In $T(n) \leq 3T(n/2) + cn$, the $3T(n/2)$ comes from the 3 recursive calls we make to compute the products ac, bd and $(a-b)(c-d)$. The cn comes from the work we have to do for the addition operations and the padding with 0's. To solve the recursion, i.e., to figure out the formula for $T(n)$, we draw the associated *recursion tree*.

[3] A non-trivial factor is a factor that is not equal to 1 or the number itself.



The root (top) of the tree corresponds to the original input with n-digit numbers and is therefore labeled with an n. This branches off into 3 nodes, one corresponding to each recursive call. These nodes are labeled with $n/2$ since they correspond to recursive calls in which the number of bits is halved. Those nodes further branch off into 3 nodes, and so on, until at the very bottom, we end up with nodes corresponding to inputs with $n = 1$. The work being done for each node of the tree is provided with a label on top of the node. For example, at the root (top), we do at most cn work before we do our recursive calls. This is why we put a cn on top of that node. Similarly, every other node can be labeled, and the total work done by the algorithm is the sum of all the labels.

We now calculate the total work done. We can divide the nodes of the tree into *levels* according to how far a node is from the root. So the root corresponds to level 0, the nodes it branches off to correspond to level 1, and so on. Observe that level j has exactly 3^j nodes. The nodes that are at level j each do $cn/2^j$ work. Therefore, the total work done at level j is $cn3^j/2^j$. The last level corresponds to level $\log_2 n$. Thus, we have:

$$T(n) \leq \sum_{j=0}^{\log_2 n} cn(3^j/2^j) = cn \sum_{j=0}^{\log_2 n} (3^j/2^j)$$

Using the formula for geometric sums, we can say that there is some constant C such that:

$$T(n) \leq Cn(3^{\log_2 n}/2^{\log_2 n}) = Cn(n^{\log_2 3}/n^{\log_2 2}) = Cn^{\log_2 3}.$$

So we conclude that $T(n) = O(n^{1.59})$. □

Exercise 6.25 (251st root)**.**
Consider the following computational problem. Given as input a number $A \in \mathbb{N}$, output $\lfloor A^{1/251} \rfloor$. Determine whether this problem can be computed in worst-case polynomial-time, i.e. $O(n^k)$ time for some constant k, where n denotes the number of bits in the binary representation of the input A. If you think the problem can be solved in polynomial time, give an algorithm in pseudocode, explain briefly why it gives the correct answer, and argue carefully why the running time is polynomial. If you think the problem cannot be solved in polynomial time, then provide a proof.

Quiz

1. True or false: $n^{\log_2 5} = \Theta(n^{\log_3 5})$.
2. True or false: $n^{\log_2 n} = \Omega(n^{15251})$.
3. True or false: $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.
4. True or false: Let $\Sigma = \{0, 1\}$ and let $L = \{0^n : n \in \mathbb{N}^+\}$. There is a Turing Machine A deciding L whose running time T_A satisfies "$T_A(n)$ is $O(n)$".
5. True or false: Continuing previous question, every Turing Machine B that decides L has running time T_B satisfying "$T_B(n)$ is $\Omega(n)$".
6. What is the running time of the following algorithm in terms of n, the input length, using the big-O notation?

```
def isPrime(N):
    if (N < 2):
        return False
    if (N == 2):
        return True
    if (N mod 2 == 0):
        return False
    maxFactor = ceiling(N**0.5)
    for factor in range(3,maxFactor+1,2):
        if (N mod factor == 0):
            return False
    return True
```

Hints to Selected Exercises

Exercise 6.17 (TM complexity of $\{0^k 1^k : k \in \mathbb{N}\}$):
Think about $\log n$ iterations with each iteration being $O(n)$ steps.

Exercise 6.18 (Is polynomial time decidability closed under concatenation?):
The statement is true.

Exercise 6.22 (Running time of the factoring problem):
It is not a polynomial-time algorithm.

Exercise 6.25 (251st root):
Binary search.

Chapter 7

Stable Matchings

PREAMBLE

Chapter structure:

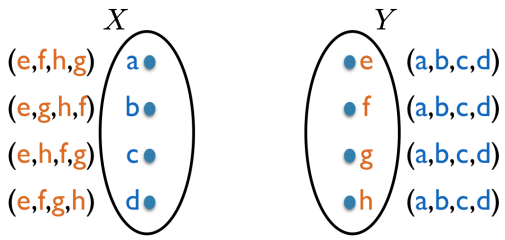
- Section 7.1 (Stable Matchings)
  - Definition 7.1 (Stable matching problem)
  - Theorem 7.2 (Gale-Shapley proposal algorithm)
  - Definition 7.3 (Best and worst valid partners)
  - Theorem 7.4 (Gale-Shapley is male optimal)

Chapter goals:

In this chapter, we analyze and solve a problem with real-world applications. This serves as another example of the power of algorithms and illustrates the impact that algorithm design can have. This chapter is also a nice prologue for Graph Theory, which we will cover next.

7.1 Stable Matchings

Definition 7.1 (Stable matching problem)**.**
An instance of the *stable matching problem* is a tuple of sets (X, Y) with $|X| = |Y|$, and a *preference list* for each element of X and Y. A preference list for an element in X is an ordering of the elements in Y, and a preference list for an element in Y is an ordering of the elements of X. Below is an example of an instance of the stable matching problem:



The output of the stable matching problem is a *stable matching*, which is a subset S of $\{(x, y) : x \in X, y \in Y\}$ with the following properties:

(i) The matching is a *perfect matching*, which means every $x \in X$ and every $y \in Y$ appear exactly once in S. If $(x, y) \in S$, we say x and y are matched.

(ii) There are no *unstable* pairs. A pair (x, y) where $x \in X$ and $y \in Y$ is called *unstable* if $(x, y) \notin S$, but they both prefer each other to the elements they are matched to.

Theorem 7.2 (Gale-Shapley proposal algorithm)**.**
There is a polynomial time algorithm which, given an instance of the stable matching problem, always returns a stable matching.

Proof. We first describe the algorithm (which is called the Gale-Shapley algorithm). For the sake of clear exposition, we refer to the elements of X as men, and the elements of Y as women.

While there is a male m in X not matched, do the following:

- Let m be an arbitrary unmatched man.
- Let w be the highest ranked woman on m's preference list to whom m has not "proposed" yet.
- Let m "propose" to w.
- If w is unmatched or w prefers m over her current partner, match m and w. (The previous partner of w, if there was any, is now unmatched.)

The theorem will follow once we show the following 3 things:

(a) the number of iterations in the algorithm is at most n^2, where $n = |X| = |Y|$;

(b) the algorithm always outputs a perfect matching;

(c) there are no unstable pairs with respect to this matching.

Part (a) implies that the algorithm is polynomial time. Parts (b) and (c) imply that the matching returned by the algorithm is a stable matching.

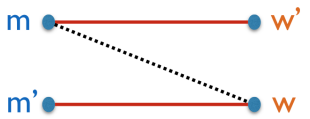
Proof of (a): Notice that the number of iterations in the algorithm is equal to the total number of proposals made. No man proposes to a woman more than once, so each man makes at most n proposals. There are n men in total, so the total number of proposals is at most n^2.

Proof of (b): The proof is by contradiction, so suppose the algorithm does not output a perfect matching. This means that some man, call it m, is not matched to any woman. The proof can be broken down as follows:

$$\begin{aligned} m \text{ is not matched at the end} &\implies \text{all women must be matched at the end} \\ &\implies \text{all men must be matched at the end.} \end{aligned}$$

This obviously leads to the desired contradiction. The second implication is simple: since there are an equal number of men and women, the only way all the women can be matched at the end is if all the men are matched. To show the first implication, notice that since m is not matched at the end, he got rejected by all the women he proposed to. Either he got rejected because the woman preferred her current partner, or he got rejected by a woman that he was already matched with. Either way, all the women that m proposed to must have been matched to someone at some point in the algorithm. But once a woman is matched, she never goes back to being unmatched. So at the end of the algorithm, all the women must be matched.

Proof of (c): We first make a crucial observation. As the algorithm proceeds, a man can only go down in his preference list, and a woman can only go up in her preference list. Now consider any pair (m, w) where $m \in X$, $w \in Y$, and m and w are not matched by the algorithm. We want to show that this pair is not unstable. Let w' be the woman that m is matched to, and let m' be the man that w is matched to.



There are two cases to consider:

(i) m proposed to w at some point in the algorithm,

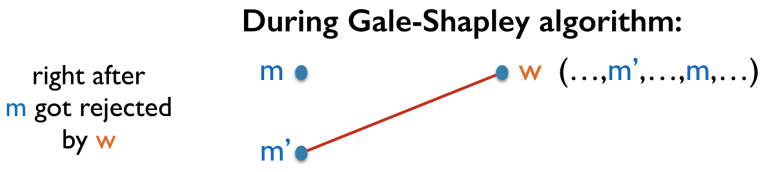
(ii) m never proposed to w.

If (i) happened, then w must have rejected m at some point, which implies w must prefer m' over m (recall w can only go up in her preference list). This implies w does not prefer m over her current partner, and so (m, w) is not unstable. If (ii) happened, then w' must be higher on the preference list of m than w (recall m can only go down in his preference list). This implies m does not prefer w over his current partner, and so (m, w) is not unstable. So in either case, (m, w) is stable, and we are done. □

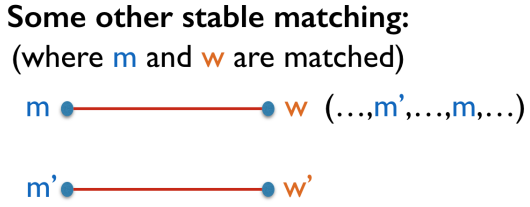
Definition 7.3 (Best and worst valid partners)**.**
Consider an instance of the stable matching problem. We say that $m \in X$ is a *valid partner* of $w \in Y$ (or w is a valid partner of m) if there is some stable matching in which m and w are matched. For $u \in X \cup Y$, we define the *best valid partner* of u, denoted best(u), to be the highest ranked valid partner of u. Similarly, we define the *worst valid partner* of u, denoted worst(u), to be the lowest ranked valid partner of u.

Theorem 7.4 (Gale-Shapley is male optimal)**.**
The Gale-Shapley algorithm always matches a male $m \in X$ with its best valid partner, i.e., it returns $\{(m, \text{best}(m)) : m \in X\}$.

Proof. The proof is by contradiction so assume that at the end of the Gale-Shapley algorithm, there is some man not matched to his best valid partner. This means that in the algorithm, some man gets rejected by a valid partner. Consider the *first time* that this happens in the algorithm. Let m be this man and w be the valid partner that rejects m. Let m' be the man that w is matched to right after rejecting m. Note that w prefers m' over m.



Since w is a valid partner of m, by definition, there is some stable matching in which m and w are matched. Let w' be the match of m' in this stable matching.



We will now show that (m', w) forms an unstable pair in the above stable matching, so in fact, the matching cannot be stable. This is the desired contradiction.

We already know that w prefers m' over m. So we just need to argue that m' prefers w over w'. And this is where we are going to use the assumption that m is the first male in the Gale-Shapley algorithm to be rejected by a valid partner. If m' actually preferred w' over w, than m' would have to be rejected by w' in the algorithm as m' later gets matched to w. This would mean m' was rejected by a valid partner before m was. Since we know this is not the case, we know that m' must prefer w over w'. This concludes the proof. □

Note 7.5 (Interesting consequence of Gale-Shapley)**.**
Note that it is not a priori clear at all that $\{(m, \text{best}(m)) : m \in X\}$ would be a matching, not to mention a stable matching.

Exercise 7.6 (Gale-Shapley is female pessimal)**.**
Show that the Gale-Shapley algorithm always matches a female $w \in Y$ with its worst valid partner, i.e., it returns $\{(\text{worst}(w), w) : w \in Y\}$.

Exercise 7.7 (Is there a unique stable matching?)**.**
Give a polynomial time algorithm that determines if a given instance of the stable matching problem has a unique solution or not.

Exercise 7.8 (Identical preferences)**.**
Suppose we are given an instance of the stable matching problem in which all the men's preferences are identical to each other, and all the women's preferences are identical to each other. Prove or disprove: there is only one stable matching for such an instance.

Exercise 7.9 (Stable roommates problem)**.**
Consider the following variant of the stable matching problem. The input is a set of n people, where n is even. Each person has a preference list that includes every other person. The goal is to find a stable matching. Give an example to show that a stable matching does not always exist.

Quiz

1. True or false: The Gale-Shapley algorithm can output different matchings based on the order of the men proposing.
2. True or false: For every stable matching instance, the male-optimal and female-optimal stable matchings differ in at least one pairing.
3. True or false: Given an instance of the stable matching problem, it is not possible for two males to have the same best valid partner.

Hints to Selected Exercises

Exercise 7.6 (Gale-Shapley is female pessimal):
Use a proof by contradiction and make use of Theorem (Gale-Shapley is male optimal).

Exercise 7.7 (Is there a unique stable matching?):
Use Theorem (Gale-Shapley is male optimal) and Exercise (Gale-Shapley is female pessimal).

Exercise 7.8 (Identical preferences):
There is indeed one stable matching.

Exercise 7.9 (Stable roommates problem):
Consider 4 people with one person being the least preferred for the others.

Chapter 8

Introduction to Graph Theory

PREAMBLE

Chapter structure:

- Section 14.1.1 (Basic Definitions)
  - Definition 8.1 (Undirected graph)
  - Definition 8.6 (Neighborhood of a vertex)
  - Definition 8.8 (d-regular graphs)
  - Theorem 8.9 (Handshake Theorem)
  - Definition 8.11 (Paths and cycles)
  - Definition 8.12 (Connected graph, connected component)
  - Theorem 8.13 (Min number of edges to connect a graph)
  - Definition 8.14 (Tree, leaf, internal node)
  - Definition 8.19 (Directed graph)
  - Definition 8.21 (Neighborhood, out-degree, in-degree, sink, source)
- Section 8.2 (Graph Algorithms)
  - Definition 8.23 (Arbitrary-first search (AFS) algorithm)
  - Definition 8.25 (Breadth-first search (BFS) algorithm)
  - Definition 8.27 (Depth-first search (DFS) algorithm)
  - Definition 8.31 (Minimum spanning tree (MST) problem)
  - Theorem 8.33 (MST cut property)
  - Theorem 8.34 (Jarník-Prim algorithm for MST)
  - Definition 8.38 (Topological order of a directed graph)
  - Definition 8.41 (Topological sorting problem)
  - Lemma 8.42 (Acyclic directed graph has a sink)
  - Theorem 8.45 (Topological sort via DFS)

Chapter goals:

In the study of computational complexity of languages and computational problems, graphs play a very fundamental role. This is because an enormous number of computational problems that arise in computer science can be abstracted away as problems on graphs, which model pairwise relations between objects. This is great for various reasons. For one, this kind of abstraction removes unnecessary distractions about the problem and allows us to focus on its essence. Second, there is a huge literature on graph theory, so we can use this arsenal to better understand the computational complexity of graph problems. Applications of graphs are too many and diverse to list here, but we'll name a few to give you an idea: communication networks, finding shortest routes in various settings, finding matchings between two sets of objects, social network analysis, kidney exchange protocols, linguistics, topology of atoms, and compiler optimization.

Our goal in this chapter is to introduce you to graph theory by providing the basic definitions and some well-known graph algorithms.

8.1 Basic Definitions

Definition 8.1 (Undirected graph)**.**
An *undirected graph*[1] G is a pair (V, E), where

- V is a finite non-empty set called the set of *vertices* (or *nodes*),
- E is a set called the set of *edges*, and every element of E is of the form $\{u, v\}$ for distinct $u, v \in V$.

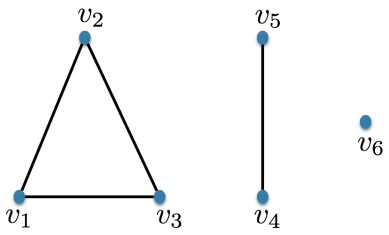
Example 8.2 (A graph with 6 vertices and 4 edges)**.**
Let $G = (V, E)$ where

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$$

and

$$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}, \{v_4, v_5\}\}.$$

We usually draw graphs in a way such that a vertex corresponds to a dot and an edge corresponds to a line connecting two dots. For example, the graph we have defined can be drawn as follows:



Note 8.3 (n and m)**.**
Given a graph $G = (V, E)$, we usually use n to denote the number of vertices $|V|$ and m to denote the number of edges $|E|$.

IMPORTANT 8.4 (Representations of graphs)**.**
There are two common ways to represent a graph. Let $v_1, v_2, \ldots, v_n$ be some arbitrary ordering of the vertices. In the *adjacency matrix* representation, a graph is represented by an $n \times n$ matrix A such that

$$A[i, j] = \begin{cases} 1 & \text{if } \{v_i, v_j\} \in E, \\ 0 & \text{otherwise.} \end{cases}$$

The adjacency matrix representation is not always the best representation of a graph. In particular, it is wasteful if the graph has very few edges. For such graphs, it can be preferable to use the *adjacency list* representation. In the adjacency list representation, you are given an array of size n and the i'th entry of the array contains a pointer to a linked list of vertex i's neighbors.

Exercise 8.5 (Max number of edges in a graph)**.**
In an n-vertex graph, what is the maximum possible value for the number of edges in terms of n?

[1] Often the word "undirected" is omitted.

Definition 8.6 (Neighborhood of a vertex)**.**
Let $G = (V, E)$ be a graph, and $e = \{u, v\} \in E$ be an edge in the graph. In this case, we say that u and v are *neighbors* or *adjacent*. We also say that u and v are *incident* to e. For $v \in V$, we define the *neighborhood* of v, denoted $N(v)$, as the set of all neighbors of v, i.e. $N(v) = \{u : \{v, u\} \in E\}$. The size of the neighborhood, $|N(v)|$, is called the *degree* of v, and is denoted by $\deg(v)$.

Example 8.7 (Example of neighborhood and degree)**.**
Consider Example 8.2 (A graph with 6 vertices and 4 edges). We have $N(v_1) = \{v_2, v_3\}$, $\deg(v_1) = \deg(v_2) = \deg(v_3) = 2$, $\deg(v_4) = \deg(v_5) = 1$, and $\deg(v_6) = 0$.

Definition 8.8 (d-regular graphs)**.**
A graph $G = (V, E)$ is called *d-regular* if every vertex $v \in V$ satisfies $\deg(v) = d$.

Theorem 8.9 (Handshake Theorem)**.**
Let $G = (V, E)$ be a graph. Then

$$\sum_{v \in V} \deg(v) = 2m.$$

Proof. Our goal is to show that the sum of the degrees of all the vertices is equal to twice the number of edges. We will use a *double counting argument* to establish the equality. This means we will identify a set of objects and count it size in two different ways. One way of counting it will give us $\sum_{v \in V} \deg(v)$, and the second way of counting it will give us $2m$. This then immediately implies that $\sum_{v \in V} \deg(v) = 2m$.

We now proceed with the double counting argument. For each vertex $v \in V$, put a "token" on all the edges it is incident to. We want to count the total number of tokens. Every vertex v is incident to $\deg(v)$ edges, so the total number of tokens put is $\sum_{v \in V} \deg(v)$. On the other hand, each edge $\{u, v\}$ in the graph will get two tokens, one from vertex u and one from vertex v. So the total number of tokens put is $2m$. Therefore it must be that $\sum_{v \in V} \deg(v) = 2m$. □

Exercise 8.10 (Application of Handshake Theorem)**.**
Is it possible to have a party with 251 people in which everyone knows exactly 5 other people in the party?

Definition 8.11 (Paths and cycles)**.**
Let $G = (V, E)$ be a graph. A *path* of length k in G is a sequence of distinct vertices

$$v_0, v_1, \ldots, v_k$$

such that $\{v_{i-1}, v_i\} \in E$ for all $i \in \{1, 2, \ldots, k\}$. In this case, we say that the path is from vertex v_0 to vertex v_k.

A *cycle* of length k (also known as a k-cycle) in G is a sequence of vertices

$$v_0, v_1, \ldots, v_{k-1}, v_0$$

such that $v_0, v_1, \ldots, v_{k-1}$ is a path, and $\{v_0, v_{k-1}\} \in E$. In other words, a cycle is just a "closed" path. The starting vertex in the cycle is not important. So for example,

$$v_1, v_2, \ldots, v_{k-1}, v_0, v_1$$

would be considered the same cycle. Also, if we list the vertices in reverse order, we consider it to be the same cycle. For example,

$$v_0, v_{k-1}, v_{k-2} \ldots, v_1, v_0$$

represents the same cycle as before.

A graph that contains no cycles is called *acyclic*.

Definition 8.12 (Connected graph, connected component)**.**
Let $G = (V, E)$ be a graph. We say that two vertices in G are *connected* if there is a path between those two vertices. We say that G is *connected* if every pair of vertices in G is connected.

A subset $S \subseteq V$ is called a *connected component* of G if G restricted to S, i.e. the graph $G' = (S, E' = \{\{u, v\} \in E : u, v \in S\})$, is a connected graph, and S is disconnected from the rest of the graph (i.e. $\{u, v\} \notin E$ when $u \in S$ and $v \notin S$). Note that a connected graph is a graph with only one connected component.

Theorem 8.13 (Min number of edges to connect a graph)**.**
Let $G = (V, E)$ be a connected graph with n vertices and m edges. Then $m \geq n - 1$. Furthermore, $m = n - 1$ if and only if G is acyclic.

Proof. We first prove that a connected graph with n vertices and m edges satisfies $m \geq n-1$. Take G and remove all its edges. This graph consists of isolated vertices and therefore contains n connected components. Let's now imagine a process in which we put back the edges of G one by one. The order in which we do this does not matter. At the end of this process, we must end up with just one connected component since G is connected. When we put back an edge, there are two options. Either

(i) we connect two different connected components by putting an edge between two vertices that are not already connected, or

(ii) we put an edge between two vertices that are already connected, and therefore create a cycle.

Observe that if (i) happens, then the number of connected components goes down by 1. If (ii) happens, the number of connected components remains the same. So every time we put back an edge, the number of connected components in the graph can go down by at most 1. Since we start with n connected components and end with 1 connected component, (i) must happen at least $n-1$ times, and hence $m \geq n-1$. This proves the first part of the theorem. We now prove $m = n - 1 \iff G$ is acyclic.

$m = n - 1 \implies G$ is acyclic: If $m = n - 1$, then (i) must have happened at each step since otherwise, we could not have ended up with one connected component. Note that (i) cannot create a cycle, so in this case, our original graph must be acyclic.

G is acyclic $\implies m = n - 1$: To prove this direction (using the contrapositive), assume $m > n-1$. We know that (i) can happen at most $n-1$ times. So in at least one of the steps, (ii) must happen. This implies G contains a cycle. □

Definition 8.14 (Tree, leaf, internal node)**.**
A graph satisfying two of the following three properties is called a *tree*:

(i) connected,

(ii) $m = n - 1$,

(iii) acyclic.

A vertex of degree 1 in a tree is called a *leaf*. And a vertex of degree more than 1 is called an *internal node*.

Exercise 8.15 (Equivalent definitions of a tree)**.**
Show that if a graph has two of the properties listed in Definition 8.14 (Tree, leaf, internal node), then it automatically has the third as well.

Exercise 8.16 (A tree has at least 2 leaves)**.**
Let T be a tree with at least 2 vertices. Show that T must have at least 2 leaves.

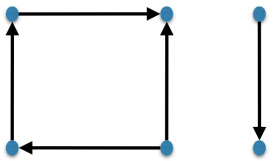
Exercise 8.17 (Max degree is at most number of leaves)**.**
Let T be a tree with L leaves. Let Δ be the largest degree of any vertex in T. Prove that $\Delta \leq L$.

Note 8.18 (Root, parent, child, sibling, etc.)**.**
Given a tree, we can pick an arbitrary node to be the *root* of the tree. In a rooted tree, we use "family tree" terminology: parent, child, sibling, ancestor, descendant, lowest common ancestor, etc. (We assume you are already familiar with these terms.)

Definition 8.19 (Directed graph)**.**
A *directed graph* G is a pair (V, A), where

- V is a non-empty finite set called the set of *vertices* (or *nodes*),
- A is a finite set called the set of *directed edges* (or *arcs*), and every element of A is a tuple (u, v) for $u, v \in V$. If $(u, v) \in A$, we say that there is a directed edge from u to v. Note that $(u, v) \neq (v, u)$ unless $u = v$.

Note 8.20 (Drawing directed graphs)**.**
Below is an example of how we draw a directed graph:



Definition 8.21 (Neighborhood, out-degree, in-degree, sink, source)**.**
Let $G = (V, A)$ be a directed graph. For $u \in V$, we define the neighborhood of u, $N(u)$, as the set $\{v \in V : (u, v) \in A\}$. The *out-degree* of u, denoted $\deg_{\text{out}}(u)$, is $|N(u)|$. The *in-degree* of u, denoted $\deg_{\text{in}}(u)$, is the size of the set $\{v \in V : (v, u) \in A\}$. A vertex with out-degree 0 is called a *sink*. A vertex with in-degree 0 is called a *source*.

Note 8.22 (Paths and cycles in directed graphs)**.**
The notions of *paths* and *cycles* naturally extend to directed graphs. For example, we say that there is a path from u to v if there is a sequence of distinct vertices $u = v_0, v_1, \ldots, v_k = v$ such that $(v_{i-1}, v_i) \in A$ for all $i \in \{1, 2, \ldots, k\}$.

8.2 Graph Algorithms

8.2.1 Graph searching algorithms

Definition 8.23 (Arbitrary-first search (AFS) algorithm)**.**
The *arbitrary-first search* algorithm, denoted AFS, is the following generic algorithm for searching a given graph. Below, "bag" refers to an arbitrary data structure that allows us to add and retrieve objects.

```
G = (V, E): graph. s: vertex in V.
AFS(⟨G, s⟩):
1 Put s into bag.
2 While bag is non-empty:
3   Pick an arbitrary vertex v from bag.
4   If v is unmarked:
5     Mark v.
6     For each neighbor w of v:
7       Put w into bag.
```

Note that when a vertex w is added to the bag, it gets there because it is the neighbor of a vertex v that has been just marked by the algorithm. In this case, we'll say that v is the *parent* of w (and w is the *child* of v). Explicitly keeping track of this parent-child relationship is convenient, so we modify the above algorithm to keep track of this information. Below, a tuple of vertices (v, w) has the meaning that vertex v is the parent of w. The initial vertex s has no parent, so we denote this situation by $(\perp, s)$.

```
G = (V, E): graph. s: vertex in V.
AFS(⟨G, s⟩):
1 Put (⊥, s) into bag.
2 While bag is non-empty:
3   Pick an arbitrary tuple (p, v) from bag.
4   If v is unmarked:
5     Mark v.
6     parent(v) = p.
7     For each neighbor w of v:
8       Put (v, w) into bag.
```

Note 8.24 (Traversing all the vertices in the graph)**.**
Note that AFS(G, s) visits all the vertices in the connected component that s is a part of. If we want to traverse all the vertices in the graph, and the graph has multiple connected components, then we can do:

```
G = (V, E): graph.
AFS2(⟨G⟩):
1 For v not marked as visited:
2   Run AFS(⟨G, v⟩).
```

Definition 8.25 (Breadth-first search (BFS) algorithm)**.**
The *breadth-first search* algorithm, denoted BFS, is AFS where the bag is chosen to be a *queue* data structure.

Note 8.26 (Running time of BFS)**.**
The running time of BFS(G, s) is $O(m)$, where m is the number of edges of the input graph. If we do a BFS for each connected component, the total running time is $O(m + n)$, where n is the number of vertices.[2] (We are assuming the graph is given as an adjacency list.)

Definition 8.27 (Depth-first search (DFS) algorithm)**.**
The *depth-first search* algorithm, denoted DFS, is AFS where the bag is chosen to be a *stack* data structure.

Note 8.28 (Recursive DFS)**.**
There is a natural recursive representation of the DFS algorithm, as follows.

```
G = (V, E): graph. s: vertex in V.
DFS(⟨G, s⟩):
1 Mark s.
2 For each neigbor v of s:
3   If v is unmarked:
4     Run DFS(⟨G, v⟩).
```

Note 8.29 (Running time of DFS)**.**
The running time of DFS(G, s) is $O(m)$, where m is the number of edges of the input graph. If we do a DFS for each connected component, the total running time is $O(m + n)$, where n is the number of vertices. (We are assuming the graph is given as an adjacency list.)

Note 8.30 (Search algorithms on directed graphs)**.**
The search algorithms presented above can be applied to directed graphs as well.

8.2.2 Minimum spanning tree

Definition 8.31 (Minimum spanning tree (MST) problem)**.**
In the *minimum spanning tree problem*, the input is a connected undirected graph $G = (V, E)$ together with a *cost* function $c : E \to \mathbb{R}^+$. The output is a subset of the edges of minimum total cost such that, in the graph restricted to these edges, all the vertices of G are connected.[3] For convenience, we'll assume that the edges have unique edge costs, i.e. $e \neq e' \implies c(e) \neq c(e')$.

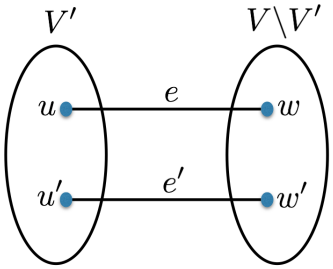
[2] Take a moment to reflect on why this is the case.

[3] Obviously this subset of edges would not contain a cycle since if it did, we could remove any edge on the cycle, preserve the connectivity property, and obtain a cheaper set. Therefore, this set forms a tree.

Note 8.32 (Unique edges costs imply unique MST)**.**
With unique edge costs, the minimum spanning tree is unique.

Theorem 8.33 (MST cut property)**.**
Suppose we are given an instance of the MST problem. For any $V' \subseteq V$, let $e = \{u, w\}$ be the cheapest edge with the property that $u \in V'$ and $w \in V \backslash V'$. Then e must be in the minimum spanning tree.

Proof. Let T be the minimum spanning tree. The proof is by contradiction, so assume that $e = \{u, w\}$ is not in T. Since T spans the whole graph, there must be a path from u to w in T. Let $e' = \{u', w'\}$ be the first edge on this path such that $u' \in V'$ and $w' \in V \backslash V'$. Let $T_{e-e'} = (T \backslash \{e'\}) \cup \{e\}$. If $T_{e-e'}$ is a spanning tree, then we reach a contradiction because $T_{e-e'}$ has lower cost than T (since $c(e) < c(e')$).



$T_{e-e'}$ is a spanning tree: Clearly $T_{e-e'}$ has $n-1$ edges (since T has $n-1$ edges). So if we can show that $T_{e-e'}$ is connected, this would imply that $T_{e-e'}$ is a tree and touches every vertex of the graph, i.e., $T_{e-e'}$ is a spanning tree. Consider any two vertices $s, t \in V$. There is a unique path from s to t in T. If this path does not use the edge $e' = \{u', w'\}$, then the same path exists in $T_{e-e'}$, so s and t are connected in $T_{e-e'}$. If the path does use $e' = \{u', w'\}$, then instead of taking the edge $\{u', w'\}$, we can take the following path: take the path from u' to u, then take the edge $e = \{u, w\}$, then take the path from w to w'. So replacing $\{u', w'\}$ with this path allows us to construct a sequence of vertices starting from s and ending at t, such that each consecutive pair of vertices is an edge. Therefore s and t are connected. □

Theorem 8.34 (Jarník-Prim algorithm for MST)**.**
There is an algorithm that solves the MST problem in polynomial time.

Proof. We first present the algorithm which is due to Jarník and Prim. Given an undirected graph $G = (V, E)$ and a cost function $c : E \to \mathbb{R}^+$:

```
G = (V, E): graph. c : E → ℝ⁺: edge costs.
MST(⟨G, c⟩):
1 V' = {u} (for some arbitrary u ∈ V)
2 E' = ∅.
3 While V' ≠ V:
4     Let {u, v} be the minimum cost edge such that u ∈ V'
      but v ∉ V'.
5     Add {u, v} to E'.
6     Add v to V'.
```

7 Output E'.

By Theorem 8.33 (MST cut property), the algorithm always adds an edge that must be in the MST. The number of iterations is $n-1$, so all the edges of the MST are added to E'. Therefore the algorithm correctly outputs the unique MST.

The running time of the algorithm can be upper bounded by $O(nm)$ because there are $O(n)$ iterations, and the body of the loop can be done in $O(m)$ time. □

Exercise 8.35 (MST with negative costs)**.**
Suppose an instance of the Minimum Spanning Tree problem is allowed to have negative costs for the edges. Explain whether we can use the Jarník-Prim algorithm to compute the minimum spanning tree in this case.

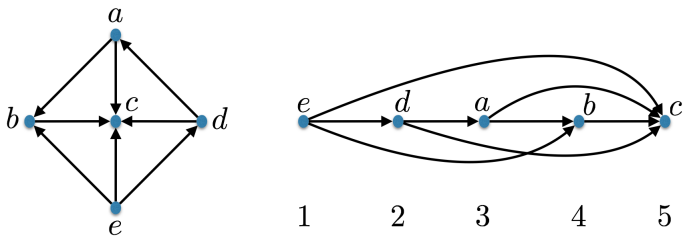
Exercise 8.36 (Maximum spanning tree)**.**
Consider the problem of computing the maximum spanning tree, i.e., a spanning tree that maximizes the sum of the edge costs. Explain whether the Jarník-Prim algorithm solves this problem if we modify it so that at each iteration, the algorithm chooses the edge between V' and $V \backslash V'$ with the maximum cost.

Exercise 8.37 (Kruskal's algorithm)**.**
Consider the following algorithm for the MST problem (which is known as Kruskal's algorithm). Start with MST being the empty set. Go through all the edges of the graph one by one from the cheapest to the most expensive. Add the edge to the MST if it does not create a cycle. Show that this algorithm correctly outputs the MST.

8.2.3 Topological sorting

Definition 8.38 (Topological order of a directed graph)**.**
A *topological order* of an n-vertex directed graph $G = (V, A)$ is a bijection $f : V \to \{1, 2, \ldots, n\}$ such that if $(u, v) \in A$, then $f(u) < f(v)$.

Example 8.39 (Example of topological order)**.**
On the left, we have a directed graph, and on the right, we represent the topological order of the graph.



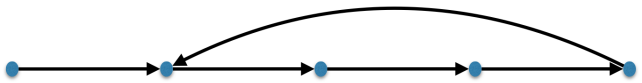
Here, $f(e) = 1$, $f(d) = 2$, $f(a) = 3$, $f(b) = 4$, and $f(c) = 5$.

Exercise 8.40 (Cycle implies no topological order)**.**
Show that if a directed graph has a cycle, then it does not have a topological order.

Definition 8.41 (Topological sorting problem)**.**
In the *topological sorting problem*, the input is a directed acyclic graph, and the output is a topological order of the graph.

Lemma 8.42 (Acyclic directed graph has a sink)**.**
If a directed graph is acyclic, then it has a sink vertex.

Proof. By contrapositive: If a directed graph has no sink vertices, then it means that every vertex has an outgoing edge. Start with any vertex, and follow an outgoing edge to arrive at a new vertex. Repeat this process. At some point, you have to visit a vertex that you have visited before. This forms a cycle.



□

Note 8.43 (Topological sort - naïve algorithm)**.**
The following algorithm solves the topological sorting problem in polynomial time.

$G = (V, A)$: directed acyclic graph.
Top-Sort-Naive($\langle G \rangle$):
1 $p = |V|$.
2 While $p \geq 1$:
3 Find a sink vertex v and remove it from G.
4 $f(v) = p$.
5 $p = p - 1$.
6 Output f.

Exercise 8.44 (Topological sort, correctness of naïve algorithm)**.**
Show the algorithm above correctly solves the topological sorting problem, i.e., show that for $(u, v) \in A$, $f(u) < f(v)$. What is the running time of this algorithm?

Theorem 8.45 (Topological sort via DFS)**.**
There is a $O(n + m)$-time algorithm that solves the topological sorting problem.

Proof. The algorithm is a slight variation of DFS.

$G = (V, A)$: directed acyclic graph.
Top-Sort($\langle G \rangle$):
1 $p = |V|$.
2 For v not marked as visited:
3 Run DFS'($\langle G, v \rangle$).

$G = (V, A)$: directed graph. v: $v \in V$.
DFS'($\langle G, v \rangle$):

1 Mark v as "visited".
2 For each neighbor u of v:
3 If u is not marked visited:
4 Run DFS'($\langle G, u \rangle$).
5 $f(v) = p$.
6 $p = p - 1$.

4 Output f.

The running time is the same as DFS. To show the correctness of the algorithm, all we need to show is that for $(u, v) \in A$, $f(u) < f(v)$. There are two cases to consider.

- *Case 1:* u is visited before v. In this case observe that DFS($\langle G, v \rangle$) will finish before DFS($\langle G, u \rangle$). Therefore $f(v)$ will be assigned a value before $f(u)$, and so $f(u) < f(v)$.
- *Case 2:* v is visited before u. Notice that we cannot visit u from DFS($\langle G, v \rangle$) because that would imply that there is a cycle. Therefore DFS($\langle G, u \rangle$) is called after DFS($\langle G, v \rangle$) is completed. As before, $f(v)$ will be assigned a value before $f(u)$, and so $f(u) < f(v)$.

□

Quiz

1. True or false: For a graph $G = (V, E)$, if for any $u, v \in V$ there exists a unique path from u to v, then G is a tree.
2. True or false: Depth-first-search algorithm runs in $O(n)$ time for a connected graph, where n is the number of vertices of the input graph.
3. True or false: If a graph on n vertices has $n - 1$ edges, then it must be acyclic.
4. True or false: If a graph on n vertices has $n - 1$ edges, then it must be connected.
5. True or false: If a graph on n vertices has $n - 1$ edges, then it must be a tree.
6. True or false: The degree sum of a graph is $\sum_{v \in V} \deg(v)$. Every tree on n vertices has exactly the same degree sum.
7. True or false: In a directed graph a self-loop, i.e. an edge of the form (u, u), is allowed by the definition.
8. True or false: Every directed graph has a topological order.
9. True or false: Suppose a graph has 2 edges with the same cost. Then there are at least 2 MSTs of the graph.
10. True or false: Let G be a 5-regular graph (i.e. a graph in which every vertex has degree exactly 5). It is possible that G has 15251 edges.

Hints to Selected Exercises

Exercise 8.15 (Equivalent definitions of a tree):
Make use of Theorem (Min number of edges to connect a graph) and its proof.

Exercise 8.16 (A tree has at least 2 leaves):
Use the Handshake Theorem.

Exercise 8.17 (Max degree is at most number of leaves):
There are at least 3 different solutions to this problem. One uses the Handshake Theorem. Another uses induction on the number of vertices.

Exercise 8.35 (MST with negative costs):
Yes, we can.

Exercise 8.36 (Maximum spanning tree):
Yes, it does. Consider multiplying the costs by -1.

Exercise 8.37 (Kruskal's algorithm):
The correctness of the algorithm follows from the MST cut property. Show by induction that every time the algorithm decides to add an edge, it adds one that must be in the MST (by the MST cut property).

Chapter 9

Matchings in Graphs

PREAMBLE

Chapter structure:

- Section 9.1 (Maximum Matchings)
  - Definition 9.1 (Matching – maximum, maximal, perfect)
  - Definition 9.5 (Maximum matching problem)
  - Definition 9.6 (Augmenting path)
  - Theorem 9.9 (Characterization for maximum matchings)
  - Definition 9.12 (Bipartite graph)
  - Definition 9.14 (k-colorable graphs)
  - Theorem 9.17 (Characterization of bipartite graphs)
  - Theorem 9.18 (Finding a maximum matching in bipartite graphs)
  - Theorem 9.20 (Hall's Theorem)
  - Corollary 9.21 (Characterization of bipartite graphs with perfect matchings)

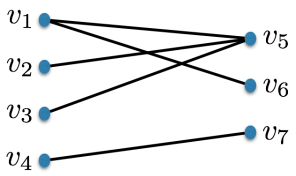
Chapter goals:

In this chapter, we continue our discussion on graphs and turn our attention to finding *matchings* in graphs. Algorithms to find matchings are used a lot in real-world applications, and we discuss some of these applications in lecture. This chapter will expand your toolkit for reasoning about graphs and help you build more intuition about them.

9.1 Maximum Matchings

Definition 9.1 (Matching – maximum, maximal, perfect)**.**
A *matching* in a graph $G = (V, E)$ is a subset of the edges that do not share an endpoint. A *maximum matching* in G is a matching with the maximum number of edges among all possible matchings. A *maximal matching* is a matching with the property that if we add any other edge to the matching, it is no longer a matching.[1] A *perfect matching* is a matching that covers all the vertices of the graph.

Example 9.2 (Examples of matchings)**.**
Consider the following graph.



Note that the empty set and a set with only one edge is always a matching. The set $M = \{\{v_1, v_5\}, \{v_4, v_7\}\}$ is a maximal matching with 2 edges, since we if we tried to add another edge to this set, it would no longer be a matching. On the other hand, this maximal matching is not a maximum matching because there is another matching with 3 edges: $M' = \{\{v_1, v_6\}, \{v_3, v_5\}, \{v_4, v_7\}\}$. This graph does not have a perfect matching. One easy way to see this is that it has an odd number of vertices, and any graph with an odd number of vertices cannot have a perfect matching.

Note 9.3 (Size of a matching)**.**
The size of a matching M refers to the number of edges in the matching, and is denoted by $|M|$. Note that this coincides with the size of the set that M represents.

Exercise 9.4 (Number of perfect matchings in a complete graph)**.**
Let n be even, and let G be the complete graph[2] on n vertices. How many different perfect matchings does G contain?

Definition 9.5 (Maximum matching problem)**.**
In the *maximum matching problem* the input is an undirected graph $G = (V, E)$ and the output is a maximum matching in G.

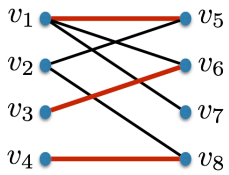
Definition 9.6 (Augmenting path)**.**
Let $G = (V, E)$ be a graph and let $M \subseteq E$ be a matching in G. An *augmenting path* in G with respect to M is a path such that

(i) the path is an *alternating path*, which means that the edges in the path alternate between being in M and not in M
(a single edge which is not in M satisfies this property),

[1] Note that a maximal matching is not necessarily a maximum matching, but a maximum matching is always a maximal matching.
[2] A complete graph is a graph in which every possible edge is present.

(ii) the first and last vertices in the path are not a part of the matching M.

Example 9.7 (Augmenting path example)**.**
Consider the following graph.

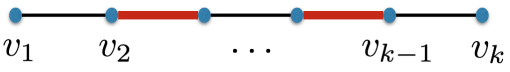


Let M be the matching $\{\{v_1, v_5\}, \{v_3, v_6\}, \{v_4, v_8\}\}$. Then the path (v_2, v_5, v_1, v_7) is an augmenting path with respect to M.

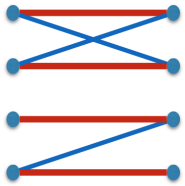
Note 9.8 (Edge cases for augmenting paths)**.**
An augmenting path does not need to contain all the edges in M. It is also possible that it does not contain *any* of the edges of M. A single edge $\{u, v\}$ where u and v are not matched is an augmenting path.

Theorem 9.9 (Characterization for maximum matchings)**.**
Let $G = (V, E)$ be a graph. A matching $M \subseteq E$ is maximum if and only if there is no augmenting path in G with respect to M.

Proof. The statement we want to prove is equivalent to the following. Given a graph $G = (V, E)$, a matching $M \subseteq E$ is not maximum if and only if there is an augmenting path in G with respect to M. There are two directions to prove.
First direction: Suppose there is an augmenting path in G with respect to M. Then we want to show that M is not maximum. Let the augmenting path be $v_1, v_2, \ldots, v_k$:



The highlighted edges represent edges in M. By the definition of an augmenting path, we know that v_1 and v_k are not matched by M. Since v_1 and v_k are not matched and the path is alternating, the number of edges on this path that are in the matching is one less than the number of edges not in the matching. To see that M is not a maximum matching, observe that we can obtain a bigger matching by flipping the matched and unmatched edges on the augmenting path. In other words, if an edge on the path is in the matching, we remove it from the matching, and if an edge on the path is not in the matching, we put it in the matching. This gives us a matching larger than M, so M is not maximum.
Second direction: We now prove the other direction. In particular, we want to show that if M is not a maximum matching, then we can find an augmenting path in G with respect to M. Let M^* denote a maximum matching in G. Since M is not maximum, we know that $|M| < |M^*|$. We define the set S to be the set of edges contained in M^* or M, but not both. That is, $S = (M^* \cup M) \backslash (M^* \cap M)$. If we color the edges in M with blue, and the edges in M^* with red, then S consists of edges that are colored either blue or red, but not both (i.e. no purple edges). Below is an example:



(Horizontal edges correspond to the red edges. The rest is blue.) Our goal is to find an augmenting path with respect to M in S (i.e., with respect to the blue edges), and once we do this, the proof will be complete.

We now proceed to find an augmenting path with respect to M in S. To do so, we make a couple of important observations about S. First, notice that each vertex that is a part of S has degree 1 or 2 because it can be incident to at most one edge in M and at most one edge in M^*. If the degree was more than 2, M and M^* would not be matchings. We make two claims:

(i) Because every vertex has degree 1 or 2, S consists of disjoint paths and cycles.

(ii) The edges in these paths and cycles alternate between blue and red.

The proof of the first claim is omitted and is left as an exercise for the reader. The second claim is true because if the edges were not alternating, i.e., if there were two red or two blue edges in a row, then this would imply the red edges or the blue edges don't form a matching (remember that in a matching no two edges can share an endpoint).

Since M^* is a bigger matching than M, we know that S has more red edges than blue edges. Observe that the cycles in S must have even length, because otherwise the edges cannot alternate between blue and red. Therefore the cycles have an equal number of red and blue edges. This implies that there must be a path in S with more red edges than blue edges. In particular, this path starts and ends with a red edge. This path is an augmenting path with respect to M (i.e., the blue edges), since it is clearly alternating between edges in M and edges not in M, and the endpoints are unmatched with respect to M. So using the assumption that M is not maximum, we were able to find an augmenting path with respect to M. This completes the proof. □

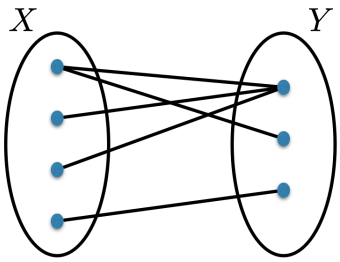
Exercise 9.10 (Graphs with max degree at most 2)**.**
Let $G = (V, E)$ be a graph such that all vertices have degree at most 2. Then prove that G consists of disjoint paths and cycles (where we count an isolated vertex as a path of length 0).

Exercise 9.11 (A tree can have at most one perfect matching)**.**
Show that a tree can have at most one perfect matching.

Definition 9.12 (Bipartite graph)**.**
A graph $G = (V, E)$ is called *bipartite* if there is a partition[3] of V into sets X and Y such that all the edges in E have one endpoint in X and the other in Y. Sometimes the bipartition is given explicitly and the graph is denoted by $G = (X, Y, E)$.

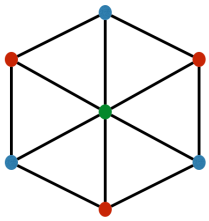
Example 9.13 (Bipartite graph example)**.**
Below is an example of a bipartite graph.

[3]Recall that a *partition* of V into X and Y means that X and Y are disjoint and $X \cup Y = V$.



Definition 9.14 (k-colorable graphs)**.**
Let $G = (V, E)$ be a graph. Let $k \in \mathbb{N}^+$. A *k-coloring* of V is just a map $\chi : V \to C$ where C is a set of cardinality k. (Usually the elements of C are called *colors*. If $k = 3$ then $C = \{\text{red}, \text{green}, \text{blue}\}$ is a popular choice. If k is large, we often just call the "colors" $1, 2, \ldots, k$.) A k-coloring is said to be *legal* for G if every edge in E is *bichromatic*, meaning that its two endpoints have different colors. (I.e., for all $\{u, v\} \in E$ it is required that $\chi(u) \neq \chi(v)$.) Finally, we say that G is *k-colorable* if it has a legal k-coloring.

Example 9.15 (A 3-colorable graph)**.**
The graph below is 3-colorable. We can color the vertex at the center green, and color the outer vertices with blue and red by alternating those two colors.



Note 9.16 (2-colorability is equivalent to bipartiteness)**.**
A graph $G = (V, E)$ is bipartite if and only if it is 2-colorable. The 2-coloring corresponds to partitioning the vertex set V into X and Y such that all the edges have one endpoint in X and the other in Y.

Theorem 9.17 (Characterization of bipartite graphs)**.**
A graph is bipartite if and only if it contains no odd-length cycles.

Proof. There are two directions to prove.
($\Longrightarrow$): For this direction, we want to show that if a graph is bipartite, then it contains no odd-length cycles. We prove the contrapositive. Observe that it is impossible to 2-color an odd-length cycle. So if a graph contains an odd-length cycle, the graph cannot be 2-colored, and therefore cannot be bipartite.
($\Longleftarrow$): For this direction, we want to show that if a graph does not contain an odd-length cycle, then it is bipartite. So suppose the graph contains no cycles of odd length. Without loss of generality, assume the graph is connected (if it is not, we can apply the argument to each connected component separately). For $u, v \in V$, let $\text{dist}(u, v)$ denote the length of the shortest path from u to v (or from v to u). Pick a starting vertex/root s and consider the "BFS tree" rooted at s. In this tree, level 0 corresponds to s, and level i corresponds to all vertices v with $\text{dist}(s, v) = i$. Color odd-indexed levels blue, and color even-indexed levels red.

The proof is done once we show that this is a valid 2-coloring of the graph. To show this, we'll argue that no edge has its endpoints colored the same color. There are two types of edges we need to worry about that could potentially have its endpoints colored the same color. We consider each type below.

First, there could potentially be an edge between two vertices u and v at the same level. Let's assume such an edge exists. Let w be the lowest common ancestor of u and v. Note that $\text{dist}(u, w) = \text{dist}(v, w)$, so the path from w to u, plus the path from w to v, plus the edge $\{u, v\}$, form an odd-length cycle. This is a contradiction.

Second, we need to consider the possibility that there is an edge between a vertex u at level i and a vertex v at level $i + 2k$ for some $k > 0$. However, the existence of such an edge implies that $\text{dist}(s, v) \leq i + 1$, which contradicts the fact that v is at level $i + 2k$. So this type of edge cannot exist either. This completes the proof. □

Theorem 9.18 (Finding a maximum matching in bipartite graphs)**.**
There is a polynomial time algorithm to solve the maximum matching problem in bipartite graphs.

Proof. Let $G = (X, Y, E)$ be the input graph. The high level steps of the algorithm is as follows.

- Let $M = \{\{x, y\}\}$ where $\{x, y\} \in E$ is an arbitrary edge.
- Repeat until there is no augmenting path with respect to M:
  - Find an augmenting path with respect to M.
  - Update M according to the augmenting path (swapping matched and unmatched edges along the path).

Every time we find an augmenting path, we increase the size of our matching by one. When there are no more augmenting paths, we stop and correctly output a maximum matching (the correctness follows from Theorem 9.9 (Characterization for maximum matchings)). The only unclear step of the algorithm is finding an augmenting path with respect to M. And we explain how to do this step below. But before we do that, note that if this step can be done in polynomial time, then the overall running time of the algorithm is polynomial time since the loop repeats $O(n)$ times and the work done in each iteration is polynomial time.

We now show how to find an augmenting path (given $G = (X, Y, E)$ and $M \subseteq E$):

- Direct edges in $E \backslash M$ from X to Y.
- Direct edges in M from Y to X.
- For each unmatched $x \in X$:
  - Run DFS(G, x).
  - If you hit an unmatched $y \in Y$, output the path from x to y.
- Output "no augmenting path found."

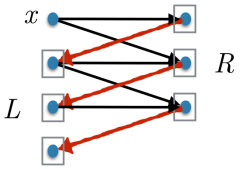
Notice that the goal of the algorithm is to find a directed path from an unmatched $x \in X$ to an unmatched $y \in Y$. The correctness of this part follows from the following observation: There is an augmenting path with respect to M if and only if there is a directed path (in the modified graph) from an unmatched vertex x in X to an unmatched vertex y in Y. (We leave it to the reader to verify this.) The running time is polynomial time since the loop repeats at most $O(n)$ times, and the work done in each iteration is polynomial time. □

Note 9.19 (Finding a maximum matching in non-bipartite graphs)**.**
The high-level algorithm above presented in the proof of Theorem 9.18 (Finding a maximum matching in bipartite graphs) is in fact applicable to general (not necessarily bipartite) graphs. However, the step of finding an augmenting path with respect to a matching turns out to be much more involved, and therefore we do not cover it in this chapter. See https://en.wikipedia.org/wiki/Blossom_algorithm if you would like to learn more.

Theorem 9.20 (Hall's Theorem)**.**
Let $G = (X, Y, E)$ be a bipartite graph. For a subset S of the vertices, let $N(S) = \bigcup_{v \in S} N(v)$. Then G has a matching covering all the vertices in X if and only if for all $S \subseteq X$, we have $|S| \leq |N(S)|$.

Proof. There are two directions to prove.
($\Longrightarrow$): For this direction, we need to show that if G has a matching covering all the vertices in X, then every $S \subseteq X$ satisfies $|S| \leq |N(S)|$. We consider the contrapositive. So suppose there is some $S \subseteq X$ such that $|S| > |N(S)|$. The vertices in S can *only* be matched to vertices in $N(S)$, and since $|S| > |N(S)|$, there cannot be a matching that covers every element in S. And this implies there cannot be a matching covering every element of X.
($\Longleftarrow$): For this direction, we need to show that if every $S \subseteq X$ satisfies $|S| \leq |N(S)|$, then there is a matching that covers all the vertices in X. We will prove the contrapositive. So assume there is no matching that covers all the vertices in X. Our goal is to find some $S \subseteq X$ such that $|S| > |N(S)|$.

In order to identify such a set S, we need to make a couple of definitions. Let M be a maximum matching and let $x \in X$ be an element that it does not cover. We turn G into a directed graph as follows: direct all edges not in M from X to Y, and direct all edges in M from Y to X. We define $L \subseteq X$ to be the set of vertices in X that you can reach by a directed path starting at x (L does not include x). And we define $R \subseteq Y$ to be the set of all vertices in Y that you can reach by a directed path starting at x. Here is an illustration:



We will show that for $S = L \cup \{x\}$, we have $|S| > |N(S)|$. We need two claims to argue this.

Claim 1: $|L| = |R|$.
Proof: Each $\ell \in L$ is matched to some $r \in R$ because the only way we can reach an $\ell \in L$ is through an edge in the matching. Conversely, each $r \in R$ must be matched to some $\ell \in L$ since if this was not true, i.e., if there was an unmatched $r \in R$, that would imply that the path from x to r is an augmenting path, and this would contradict the fact that M is a maximum matching (Theorem 9.9 (Characterization for maximum matchings)). Since every element of L is matched by M to an element of R and vice versa, there is a one-to-one correspondence between L and R, i.e., $|L| = |R|$.

Claim 2: In the original undirected graph, $N(L \cup \{x\}) \subseteq R$.
(In fact, $N(L \cup \{x\}) = R$ but we only need one side of the inclusion.)
Proof: For any $\ell \in L \cup \{x\}$, we want to argue that $N(\ell) \subseteq R$. First consider the case that $\ell = x$. Then all the neighbors of x must be in R since all the edges

incident to x are directed from left to right. So $N(x) \subseteq R$, as desired. Now consider any $\ell \in L$. We want to argue that all the neighbors of ℓ must be in R. To argue about the neighbors of ℓ, we look at all the edges incident to ℓ. In this set of edges incident to ℓ, exactly one edge e is in the matching M. Note that $e \in M$ is directed from Y to X, and must be incident to some $r \in R$ because the only way to reach ℓ is through some $r \in R$ via e. If we now look at all the other edges incident to ℓ, note that they must be directed from X to Y, and the vertices $K \subseteq Y$ that they are incident to must be in R. This is because by definition of L, $\ell \in L$ is reachable from x, which means the vertices in K would also be reachable from x, and therefore would be in R (by the definition of R). This shows that every neighbor of ℓ is in R, and completes the proof of Claim 2.

Combining Claim 1 and Claim 2 above, we have

$$|L \cup \{x\}| > |R| \geq |N(L \cup \{x\})|,$$

i.e., for $S = L \cup \{x\}$, $|S| > |N(S)|$, as desired. □

Corollary 9.21 (Characterization of bipartite graphs with perfect matchings)**.** *Let $G = (X, Y, E)$ be a bipartite graph. Then G has a perfect matching if and only if $|X| = |Y|$ and for any $S \subseteq X$, we have $|S| \leq |N(S)|$.*

Note 9.22 (Hall's Theorem when the two parts have equal size)**.**
Sometimes people call the above corollary Hall's Theorem.

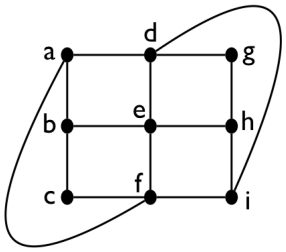
Exercise 9.23 (Practice with perfect matchings)**.**

(a) Let G be a bipartite graph on $2n$ vertices such that every vertex has degree at least n. Prove that G must contain a perfect matching.

(b) Let $G = (X, Y, E)$ be a bipartite graph with $|X| = |Y|$. Show that if G is connected and every vertex has degree at most 2, then G must contain a perfect matching.

Quiz

1. True or false: Given a matching M, there can be at most one augmenting path with respect to M.
2. True of false: A matching M in a non-bipartite graph G is maximum if and only if there is no augmenting path with respect to M.
3. True or false: The graph below is bipartite.



Hints to Selected Exercises

Exercise 9.10 (Graphs with max degree at most 2):
Induct on the number of vertices.

Exercise 9.11 (A tree can have at most one perfect matching):
There are various solutions. One approach is to go by contradiction and assume a tree has two different perfect matchings. Then consider the symmetric difference between those perfect matchings.

Exercise 9.23 (Practice with perfect matchings):
For part (a), you can use Hall's Theorem. For part (b), use Exercise (Graphs with max degree at most 2).

Chapter 10

Boolean Circuits

PREAMBLE

Chapter structure:

- Section 14.1.1 (Basic Definitions)
  - Definition 10.3 (Boolean circuit)
  - Definition 10.8 (Circuit family)
  - Definition 10.9 (A circuit family deciding/computing a decision problem)
  - Definition 10.10 (Circuit size and complexity)
- Section 10.2 (3 Theorems on Circuits)
  - Theorem 10.14 ($O(2^n)$ upper bound on circuit complexity)
  - Proposition 10.15 (Number of Boolean functions)
  - Theorem 10.16 (Shannon's Theorem)
  - Lemma 10.17 (Counting circuits)
  - Theorem 10.20 (Efficient TM implies efficient circuit)
  - Definition 10.21 (Complexity class P)
  - Corollary 10.22 (A language in P has polynomial circuit complexity)

Chapter goals:

In this chapter we introduce a new model of computation called *Boolean circuits*. Boolean circuits have some desirable properties as a computational model. For instance, arguably the definition of a circuit is simpler than a Turing machine. Therefore many people find circuits easier to reason about. Plus, they are closely related to the Turing machine model. In particular, as we will see, if a language can be computed efficiently by a Turing machine, then it can also be computed efficiently by circuits.

In addition to these desirable properties, Boolean circuits are a good mathematical model to study parallel computation (even though we will not do so in this course). Furthermore, real computers are built using digital circuits, and therefore engineers like to study and understand circuits.

Our main motivation to study circuits, and to do it at this point in the course, is that Boolean circuits are related to the famous P vs NP question in various ways. In lecture, we explore one of these connections. In a future chapter on NP and NP-completeness, we will explore another important connection.

Our goal in this chapter is to introduce you to the Boolean circuit model of computation, present the basic definitions regarding what it means for a circuit to compute/decide a language and how we measure the complexity of a circuit. Afterwards, we present 3 Theorems which we hope will give you a big picture view on the Boolean circuit model and how it compares to the Turing machine model.

10.1 Basic Definitions

Note 10.1 (Dividing the set of words by length)**.**
For a finite alphabet Σ, Σ^n denotes all words in Σ^* with length (i.e. number of symbols) exactly n. Let $f : \{0,1\}^* \to \{0,1\}$ be a decision problem. We denote by $f^n : \{0,1\}^n \to \{0,1\}$ the restriction of f to words of length n (i.e. for $x \in \{0,1\}^n$, $f^n(x) \stackrel{\text{def}}{=} f(x)$). So f can be thought of as a collection of functions $(f^0, f^1, f^2, \ldots)$. We'll write $f = (f^0, f^1, f^2, \ldots)$ as a shorthand for this.

Note 10.2 (Unary NOT, binary AND, binary OR)**.**
We denote by $\neg$ the unary NOT operation, by $\wedge$ the binary AND operation, and by $\vee$ the binary OR operation. In particular, we can write the truth tables of these operations as follows:

x	$\neg x$
0	1
1	0

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

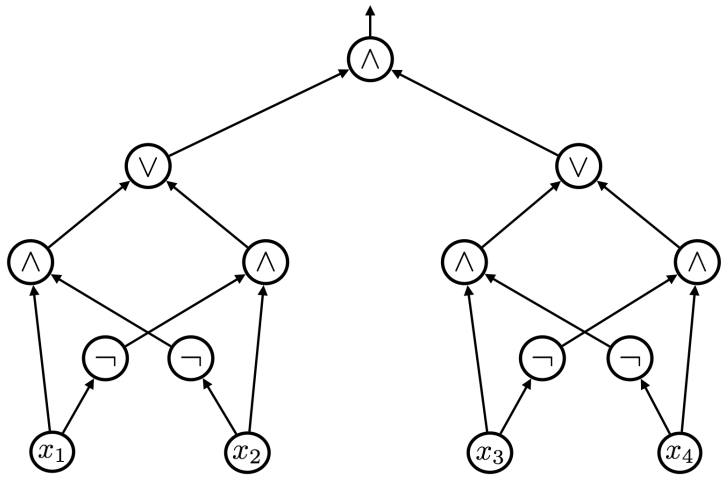
x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

Definition 10.3 (Boolean circuit)**.**
A Boolean circuit with n-input variables ($n \geq 0$) is a directed acyclic graph with the following properties. Each node of the graph is called a *gate* and each directed edge is called a *wire*. There are 5 types of gates that we can choose to include in our circuit: AND gates, OR gates, NOT gates, input gates, and constant gates. There are 2 constant gates, one labeled 0 and one labeled 1. These gates have in-degree/fan-in[1] 0. There are n input gates, one corresponding to each input variable. These gates also have in-degree/fan-in 0. An AND gate corresponds to the binary AND operation $\wedge$ and an OR gate corresponds to the binary OR operation $\vee$. These gates have in-degree/fan-in 2. A NOT gate corresponds to the unary NOT operation $\neg$, and has in-degree/fan-in 1. One of the gates in the circuit is labeled as the *output gate*. Gates can have out-degree more than 1, with the exception of the output gate, which has out-degree 0.

For each $0/1$ assignment to the input variables, the Boolean circuit produces a one-bit output. The output of the circuit is the output of the gate that is labeled as the *output gate*. The output is calculated naturally using the truth tables of the operations corresponding to the gates. The input-output behavior of the circuit defines a function $f : \{0,1\}^n \to \{0,1\}$ and in this case, we say that the circuit *computes* this function.

Example 10.4 (A circuit computing $(x_1 \neq x_2) \wedge (x_3 \neq x_4)$)**.**
Below is an example of how we draw a circuit. In this example, $n = 4$.

[1] The in-degree of a gate is also known as the *fan-in* of the gate.



The output gate is the gate at the very top with an arrow that links to nothing. The circuit outputs 1 if and only if $x_1 \neq x_2$ and $x_3 \neq x_4$.

Note 10.5 (Types of gates a circuit can have)**.**
A Boolean circuit can be defined to have gates other than AND, OR, and NOT. For example, we could define circuits to have XOR (exclusive-or) gates. However, restricting the gate options to AND, OR and NOT is a common and convenient choice. Unless specified otherwise, we will stick to this restriction in these notes.

Exercise 10.6 (Drawing circuits)**.**
Draw a circuit that computes the following functions.

(a) The parity function PAR : $\{0,1\}^2 \to \{0,1\}$ on 2 variables, which is defined as PAR$(x_1, x_2) = 1$ iff $x_1 + x_2$ is odd.

(b) The majority function MAJ : $\{0,1\}^3 \to \{0,1\}$ on 3 variables, which is defined as MAJ$(x_1, x_2, x_3) = 1$ iff $x_1 + x_2 + x_3 \geq 2$.

Exercise 10.7 (NAND is "universal")**.**
Define a NAND gate as NAND$(x, y) = \neg(x \wedge y)$. Show that any circuit with AND, OR and NOT gates can be converted into an equivalent circuit (i.e. a circuit computing the same function) that uses *only* NAND gates (in addition to the input gates and constant gates). The size of this circuit should be at most a constant times the size of the original circuit.

Definition 10.8 (Circuit family)**.**
A *circuit family* C is a collection of circuits, $(C_0, C_1, C_2, \ldots)$, such that each C_n is a circuit that has access to n input gates.

Definition 10.9 (A circuit family deciding/computing a decision problem)**.**
Let $f : \{0,1\}^* \to \{0,1\}$ be a decision problem and let $f^n : \{0,1\}^n \to \{0,1\}$ be the restriction of f to words of length n. We say that a circuit family $C = (C_0, C_1, C_2, \ldots)$ *decides/computes* f if C_n computes f^n for every n.

Definition 10.10 (Circuit size and complexity)**.**
The size of a circuit is defined to be the number of gates in the circuit, excluding the constant gates 0 and 1. The size of a circuit family $C = (C_0, C_1, C_2, \ldots)$ is a function $S : \mathbb{N} \to \mathbb{N}$ such that $S(n)$ equals the size of C_n. The *circuit complexity* of a decision problem $f = (f^0, f^1, f^2, \ldots)$ is the size of the minimal circuit family that decides f. In other words, the circuit complexity of f is defined to be a function $CC_f : \mathbb{N} \to \mathbb{N}$ such that $CC_f(n)$ is the minimum size of a circuit computing f^n. Using the correspondence between decision problems and languages, we can also define the circuit complexity of a language in the same manner.[2]

Note 10.11 (Defining the size of a circuit)**.**
Sometimes the size of a circuit is defined to be the number of non-input, non-constant gates (i.e., it is the number of gates not counting the input gates or the constant gates). In these notes, we do count the input gates when computing the size.

Exercise 10.12 (Circuit complexity of parity)**.**
Let $L \subseteq \{0,1\}^*$ be the set of words which contain an odd number of 1's. Show that the circuit complexity of L is $\Theta(n)$.

10.2 3 Theorems on Circuits

Exercise 10.13 ($O(n2^n)$ upper bound on circuit complexity)**.**
Show that any language L can be computed by a circuit family of size $O(n2^n)$.

Theorem 10.14 ($O(2^n)$ upper bound on circuit complexity)**.**
Any language $L \subseteq \{0,1\}^*$ can be computed by a circuit family of size $O(2^n)$.

Proof. Let

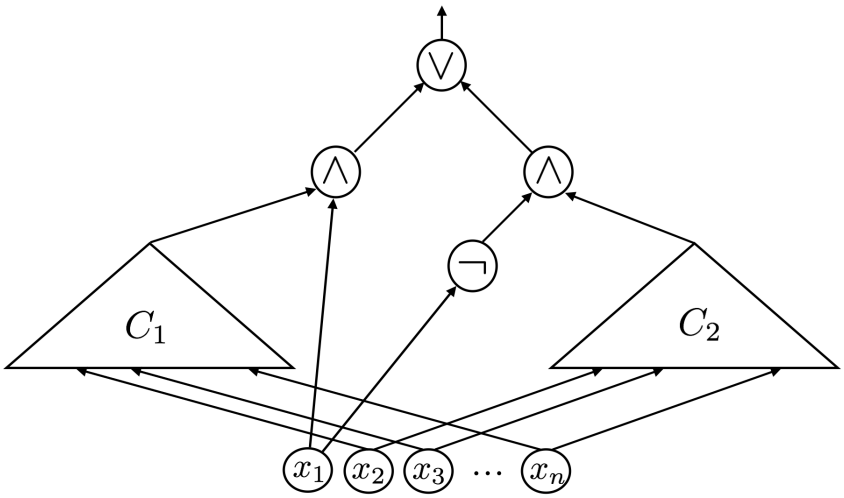
$$S_{\max}(n) = \max_{f:\{0,1\}^n \to \{0,1\}} \text{size of the smallest circuit computing } f,$$

where the maximum is over *all possible* Boolean functions $f : \{0,1\}^n \to \{0,1\}$. Observe that the theorem follows once we show that $S_{\max}(n) = O(2^n)$. Take *any* function $f : \{0,1\}^n \to \{0,1\}$. Notice that we can write

$$f(x_1, x_2, \ldots, x_n) = (x_1 \wedge f(1, x_2, \ldots, x_n)) \vee (\neg x_1 \wedge f(0, x_2, \ldots, x_n)).$$

(This equality can be verified by considering the two cases $x_1 = 0$ and $x_1 = 1$. We leave this part to the reader.) Let C_1 be the smallest size circuit that computes $f(1, x_2, \ldots, x_n)$ and let C_2 be the smallest size circuit that computes $f(0, x_2, \ldots, x_n)$. Note that C_1 and C_2 compute functions on $n-1$ variables. We can then construct a circuit for $f(x_1, x_2, \ldots, x_n)$ by constructing a circuit for $(x_1 \wedge f(1, x_2, \ldots, x_n)) \vee (\neg x_1 \wedge f(0, x_2, \ldots, x_n))$, as shown in the picture below.

[2]Note that circuit complexity corresponds to the intrinsic complexity of the language with respect to the computational model of Boolean circuits. In the case of Boolean circuits, intrinsic complexity (i.e. circuit complexity) is well-defined.



Since the circuits C_1 and C_2 compute functions on $n-1$ variables, their size is bounded by $S_{\max}(n-1)$ each. Then the size of the above circuit is at most $2S_{\max}(n-1)+5$ (the extra 5 gates are for x_1, the NOT gate, the two AND gates, and the output OR gate). Our construction above works for any function $f : \{0,1\}^n \to \{0,1\}$, so we can conclude that $S_{\max}(n) \leq 2S_{\max}(n-1)+5$. Observe that $S_{\max}(0) = 1$. It is then easy to solve the recurrence and verify that $S_{\max}(n) = O(2^n)$ (we omit this part of the proof). □

Proposition 10.15 (Number of Boolean functions)**.**
The set of all functions of the form $f : \{0,1\}^n \to \{0,1\}$ has size 2^{2^n}.

Proof. A function of the form $f : \{0,1\}^n \to \{0,1\}$ has 2^n possible inputs. For each input, we have 2 choices for the output, either 0 or 1. Therefore we have 2^{2^n} different functions. □

Theorem 10.16 (Shannon's Theorem)**.**
There exists a language $L \subseteq \{0,1\}^*$ such that any circuit family computing L must have size at least $2^n/5n$.

Proof. Our goal is to show that for all n, there is a function $f_*^n : \{0,1\}^n \to \{0,1\}$ which cannot be computed by a circuit of size less than $2^n/5n$. If we can do this, then the decision problem $f_* = (f_*^0, f_*^1, f_*^2, \ldots)$ (see Note 10.1 (Dividing the set of words by length)) corresponds to the language such that any circuit family computing it must have size at least $2^n/5n$.

Fix an arbitrary n. To show that there is some function $f_*^n : \{0,1\}^n \to \{0,1\}$ which cannot be computed by a circuit of size less than $2^n/5n$, our strategy will be as follows. We will show that the total number of circuits of size less than $2^n/5n$ is *strictly less than* the total number of functions $f : \{0,1\}^n \to \{0,1\}$. Since one circuit computes one function, this implies that there are not enough circuits of size less than $2^n/5n$ to compute every possible function. So there exists at least one function which cannot be computed by a circuit of size less than $2^n/5n$.

From Proposition 10.15 (Number of Boolean functions) we know that the total number of functions $f : \{0,1\}^n \to \{0,1\}$ is 2^{2^n}. In the next lemma (Lemma 10.17 (Counting circuits)), we show that the number of possible circuits of size at most s is less than or equal to $2^{5s\log s}$. It is an easy exercise (which we leave to the reader) to confirm that for $s \leq 2^n/5n$, $2^{5s\log s} < 2^{2^n}$. In other words, for $s \leq 2^n/5n$, there are more functions than circuits, and the result follows. □

Lemma 10.17 (Counting circuits)**.**
The number of possible circuits of size at most s is less than or equal to $2^{5s \log s}$.

Proof. Let A be the set of circuits of size at most s. We want to show that $|A| \leq 2^{5s \log s}$. Let $B = \{0,1\}^{5s \log s}$. Recall that $|A| \leq |B|$ if and only if there is a surjection from B to A (or equivalently an injection from A to B). Since $|B| = 2^{5s \log s}$, we are done once we show there is a surjection from B to A.

To show that there is a surjection, it suffices to show how to encode a circuit of size at most s with a binary string of length $5s \log s$.[3] The encoding is as follows. Number the gates of the circuit $1, 2, \ldots, s$. Note that it takes $\log_2 s$ bits to write down the number of a gate in binary. We'll assume that the first gate corresponds to the output gate. For each gate of the circuit, write down in binary:

(i) type of the gate (constant, input, OR, AND, NOT),

(ii) from which gates the inputs are coming from.

Once we know (i) and (ii) for every gate, we have all the information to reconstruct the circuit. Note that (i) takes 3 bits to specify, and (ii) takes $2 \log s$ bits.[4] Since we do this for each gate in the circuit, the total number of bits is $s(3 + 2 \log s)$, which can be upper bounded by $5s \log s$. □

Note 10.18 (Almost all functions have exponential circuit complexity)**.**
A minor adjustment to the proof of Theorem 10.16 (Shannon's Theorem) allows us to conclude that *almost all* Boolean functions $f : \{0,1\}^n \to \{0,1\}$ cannot be computed by a circuit with sub-exponential[5] size.

Note 10.19 (Connecting circuit complexity to TM complexity)**.**
The theorem below connects circuit complexity with running-time complexity in the Turing machine computational model. The importance of the theorem is highlighted in the corollary that comes after it. We will also make use of this theorem in a future chapter to prove a very important theorem. We provide a sketch of the proof for the curious reader, but you will not be responsible for this proof.

Theorem 10.20 (Efficient TM implies efficient circuit)**.**
Let $L \subseteq \{0,1\}^*$ be a language which can be decided in $O(T(n))$ time. Then L can be computed by a circuit family of size $O(T(n)^2)$.

Proof Sketch. [6]

Let L be decided by a TM M in $O(T(n))$ time. For simplicity, we will assume that M's tape is infinite in one direction (to the right) as opposed to infinite in both directions.

Fix an arbitrary input length n. We want to design a circuit on n input bits such that for the inputs accepted by M, the circuit will output 1, and for the inputs rejected by M, the circuit will output 0. The size of our circuit will be $O(T(n)^2)$.

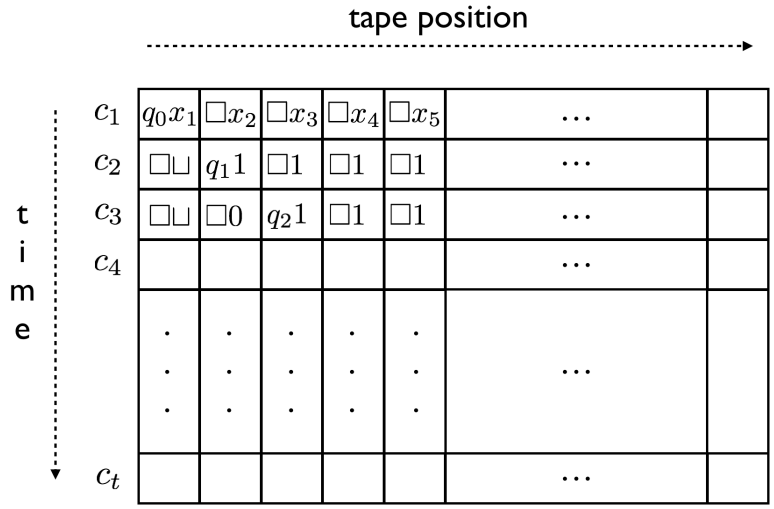
[3] Note the similarity to the CS method for showing a set is countable. Here, instead of showing that a set if countable, we are putting a finite upper bound on the size of a set using the CS method. We do this by showing how to encode the elements of the set with finite length strings with an explicit upper bound on the length.

[4] It is true that some gates take no input, but we will still use $2 \log s$ bits to specify that.

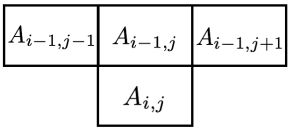
[5] "Sub-exponential" means "smaller than exponential".

[6] The diagrams in this proof are redrawings of the ones in https://lucatrevisan.wordpress.com/2010/04/25/cs254-lecture-3-boolean-circuits/.

Let's denote the input by $x_1, x_2, \ldots, x_n$. We know that when M runs on this input, it goes through configurations (see Definition 3.6 (A TM accepting or rejecting a string)) $c_1, c_2, \ldots, c_t$, where each configuration c_i is of the form uqv for $u, v \in \Gamma^*$, $q \in Q$. Here the number of steps M takes is t so $t = O(T(n))$. Consider a $t \times t$ table where row i corresponds to c_i.

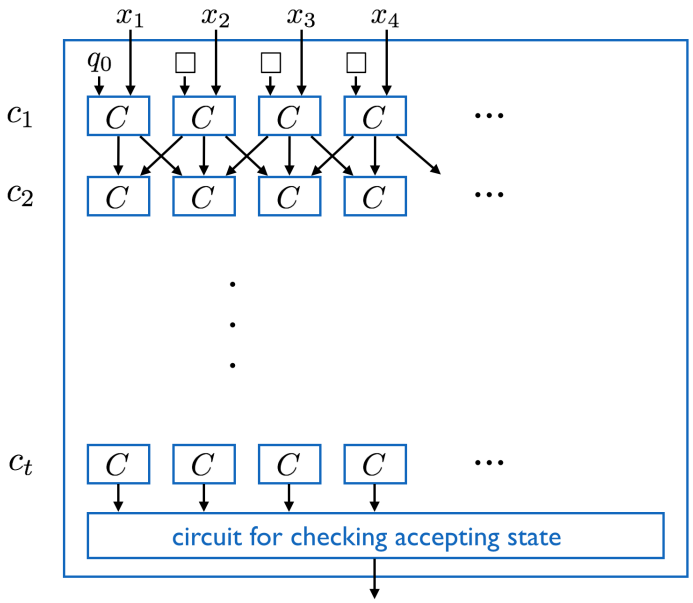


Observe that we need at most t columns because the TM can use at most t cells in t steps. Each cell contains two pieces of information: (i) a state name or NONE (if no state is given), (ii) a symbol from Γ. Let's call this table A. In the table above, NONE is represented with a box $\square$, and the input is assumed to be the all-1 string. Observe that the contents of a cell of the table $A_{i,j}$ are determined by the contents of $A_{i-1,j-1}$, $A_{i-1,j}$ and $A_{i-1,j+1}$.



The transition function of TM M governs this transformation. Assume each cell encodes k bits of information. Note that k is a constant because $|Q|$ and $|\Gamma|$ are constant. So the transition function is of the form $\{0,1\}^{3k} \to \{0,1\}^k$. It determines the contents of a cell based on the contents of the three cells above it and it can be implemented by a circuit of constant size since k is constant.[7] (Note that we can allow our circuit to have multiple output gates, one for each output bit of the function.) Let's call this circuit C. Now we can build a circuit that computes the answer given by M as shown in the picture below.

[7] Recall that any function can be computed by a circuit (Theorem 10.14 ($O(2^n)$ upper bound on circuit complexity)).



The size of the circuit is at most ct^2 for some constant c. $\square$

Definition 10.21 (Complexity class P)**.**
We denote by P the set of all languages that can be decided in polynomial-time, i.e., in time $O(n^k)$ for some constant $k > 0$.

Corollary 10.22 (A language in P has polynomial circuit complexity)**.**
If $L \in$ P, then L can be computed by a circuit family of polynomial size. Equivalently, if L cannot be computed by a circuit family of polynomial size, then $L \notin$ P.

Quiz

1. True or false: If a language can be computed by a circuit family of size $O(n^2)$, then it can be decided in polynomial time by a Turing machine.

2. True or false: A unary language $L \subseteq \{0,1\}^*$ is a language such that no word in L contains a 1. Every unary language can be computed by a circuit family of size $O(n)$.

3. True or false: The circuit complexity of $\{0,1\}^*$ is $O(n)$.

4. Fix $\Sigma = \{0,1\}$. Let $(D_0, D_1, D_2, \ldots)$ be a family of DFAs. We say that a DFA family like this computes a language $L \subseteq \{0,1\}^*$ if for all n, D_n decides $L_n = L \cap \{0,1\}^n$. True or false: there is a DFA family that computes HALTS.

5. True or false: Fix $\Sigma = \{0,1\}$. Then there is a circuit family that computes HALTS $\subseteq \Sigma^*$.

Hints to Selected Exercises

Exercise 10.7 (NAND is "universal"):
Show how to compute OR, AND, and NOT just with NAND gates.

Exercise 10.12 (Circuit complexity of parity):
Note that you need to show both an upper bound and a lower bound. For the upper bound, you can construct a circuit family computing L. Assume for simplicity that n is a power of 2 and try to think of a recursive construction. For the lower bound, focus on the input gates.

Exercise 10.13 ($O(n2^n)$ upper bound on circuit complexity):
See the solution, which is actually more like a hint rather than a complete solution.

Chapter 11

Polynomial-Time Reductions

PREAMBLE

Chapter structure:

- Section 11.1 (Cook and Karp Reductions)
  - Definition 11.1 (k-Coloring problem)
  - Definition 11.2 (Clique problem)
  - Definition 11.3 (Independent set problem)
  - Definition 11.4 (Circuit satisfiability problem)
  - Definition 11.5 (Boolean satisfiability problem)
  - Definition 11.12 (Karp reduction: Polynomial-time many-one reduction)
  - Theorem 11.16 (CLIQUE reduces to IS)
  - Theorem 11.20 (CIRCUIT-SAT reduces to 3COL)
- Section 11.2 (Hardness and Completeness)
  - Definition 11.21 ($\mathcal{C}$-hard, $\mathcal{C}$-complete)

Chapter goals:

In a previous chapter, we have studied reductions and saw how they can be used to both expand the landscape of decidable languages and expand the landscape of undecidable languages. As such, the concept of a reduction is a very powerful tool.

In the context of computational complexity, reductions once again play a very important role. In particular, by tweaking our definition of a reduction a bit so that it is required to be efficiently computable, we can use reductions to expand the landscape of tractable (i.e. efficiently decidable) languages and to expand the landscape of intractable languages. Another cool feature of reductions is that it allows us to show that many seemingly unrelated problems are essentially the same problem in disguise, and this allows us to have a deeper understanding of the mysterious and fascinating world of computational complexity.

Our goal in this chapter is to introduce you to the concept of polynomial-time (efficient) reductions, and give you a few examples to make the concept more concrete and illustrate their power.

In some sense, this chapter marks the introduction to our discussion on the P vs NP problem, even though the definition of NP will only be introduced in the next chapter, and the connection will be made more precise then.

11.1 Cook and Karp Reductions

Definition 11.1 (k-Coloring problem)**.**
In the *k-coloring problem*, the input is an undirected graph $G = (V, E)$, and the output is True if and only if the graph is k-colorable (see Definition 9.14 (k-colorable graphs)). We denote this problem by kCOL. The corresponding language is

$$\{\langle G \rangle : G \text{ is a } k\text{-colorable graph}\}.$$

Definition 11.2 (Clique problem)**.**
Let $G = (V, E)$ be an undirected graph. A subset of the vertices is called a *clique* if there is an edge between any two vertices in the subset. We say that G contains a k-clique if there is a subset of the vertices of size k that forms a clique.

In the *clique problem*, the input is an undirected graph $G = (V, E)$ and a number $k \in \mathbb{N}^+$, and the output is True if and only if the graph contains a k-clique. We denote this problem by CLIQUE. The corresponding language is

$$\{\langle G, k \rangle : G \text{ is a graph, } k \in \mathbb{N}^+, G \text{ contains a } k\text{-clique}\}.$$

Definition 11.3 (Independent set problem)**.**
Let $G = (V, E)$ be an undirected graph. A subset of the vertices is called an *independent set* if there is no edge between any two vertices in the subset. We say that G contains an independent set of size k if there is a subset of the vertices of size k that forms an independent set.

In the *independent set problem*, the input is an undirected graph $G = (V, E)$ and a number $k \in \mathbb{N}^+$, and the output is True if and only if the graph contains an independent set of size k. We denote this problem by IS. The corresponding language is

$$\{\langle G, k \rangle : G \text{ is a graph, } k \in \mathbb{N}^+, G \text{ contains an independent set of size } k\}.$$

Definition 11.4 (Circuit satisfiability problem)**.**
We say that a circuit is *satisfiable* if there is $0/1$ assignment to the input gates that makes the circuit output 1. In the *circuit satisfiability problem*, the input is a Boolean circuit, and the output is True if and only if the circuit is satisfiable. We denote this problem by CIRCUIT-SAT. The corresponding language is

$$\{\langle C \rangle : C \text{ is a Boolean circuit that is satisfiable}\}.$$

Definition 11.5 (Boolean satisfiability problem)**.**
Let $x_1, \ldots, x_n$ be Boolean variables, i.e., variables that can be assigned True or False. A *literal* refers to a Boolean variable or its negation. A *clause* is an "OR" of literals. For example, $x_1 \vee \neg x_3 \vee x_4$ is a clause. A Boolean formula in *conjunctive normal form* (CNF) is an "AND" of clauses. For example,

$$(x_1 \vee \neg x_3 \vee x_4) \wedge (x_2 \vee x_2 \vee x_4) \wedge (x_1 \vee \neg x_1 \vee \neg x_5)$$

is a CNF formula. We say that a Boolean formula is *satisfiable* if there is a $0/1$ assignment to the Boolean variables that makes the formula evaluate to 1.

In the CNF *satisfiability problem*, the input is a CNF formula, and the output is True if and only if the formula is satisfiable. We denote this problem by SAT. The corresponding language is

$$\{\langle\varphi\rangle : \varphi \text{ is a satisfiable CNF formula}\}.$$

In a variation of SAT, we restrict the input formula such that every clause has exactly 3 literals (we call such a formula a 3CNF formula). This variation of the problem is denoted by 3SAT.

Note 11.6 (Names of decision problems and languages)**.**
The name of a decision problem above refers both to the decision problem and the corresponding language.

Note 11.7 (Inputs of decision problems)**.**
Recall that in all the decision problems above, the input is an arbitrary word in Σ^*. If the input does not correspond to a valid encoding of an object expected as input (e.g. a graph in the case of kCOL), then those inputs are rejected (i.e., they are not in the corresponding language).

Note 11.8 (Exponential-time algorithms for the decision problems above)**.**
All the problems defined above are decidable and have exponential-time algorithms solving them.

Note 11.9 (Cook reduction: Polynomial-time (Turing) reduction)**.**
Fix some alphabet Σ. Let A and B be two languages. We say that A *polynomial-time reduces* to B, written $A \leq^P B$, if there is a polynomial-time decider for A that uses a decider for B as a black-box subroutine.[1] Polynomial-time reductions are also known as *Cook reductions*, named after Stephen Cook.

Note 11.10 (Polynomial-time reductions and P)**.**
Observe that if $A \leq^P B$ and $B \in \mathsf{P}$, then $A \in \mathsf{P}$. Equivalently, taking the contrapositive, if $A \leq^P B$ and $A \notin \mathsf{P}$, then $B \notin \mathsf{P}$. So when $A \leq^P B$, we think of B as being at least as hard as A with respect to polynomial-time decidability.

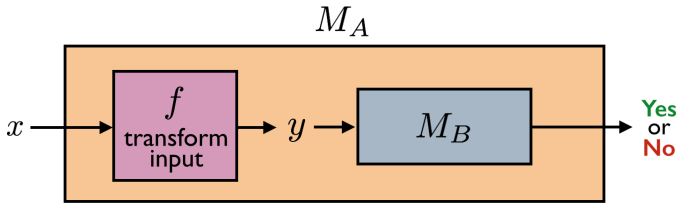
Note 11.11 (Transitivity of Cook reductions)**.**
Note that if $A \leq^P B$ and $B \leq^P C$, then $A \leq^P C$.

Definition 11.12 (Karp reduction: Polynomial-time many-one reduction)**.**
Let A and B be two languages. Suppose that there is a polynomial-time computable function (also called a polynomial-time transformation) $f : \Sigma^* \to \Sigma^*$ such that $x \in A$ if and only if $f(x) \in B$. Then we say that there is a *polynomial-time many-one reduction* (or a *Karp reduction*, named after Richard Karp) from A to B, and denote it by $A \leq_m^P B$.

[1] Technically, the black-box decider for B is called an oracle, and every call to the oracle is assumed to take 1 step. In these notes, we omit the formal definition of these reductions that require introducing oracle Turing machines. This semi-informal treatment is sufficient for our purposes.

IMPORTANT 11.13 (Many-one reductions vs Turing reductions)**.**
If there is a many-one reduction from language A to language B, then one can construct a regular (Turing) reduction from A to B. We explain this below.

To establish a Turing reduction from A to B, we need to show how we can come up with a decider M_A for A given that we have a decider M_B for B. Now suppose we have a many-one reduction from A to B. This means we have a computable function f as in the definition of a many-one reduction. This f then allows us to build M_A as follows. Given any input x, first feed x into f, and then feed the output $y = f(x)$ into M_B. The output of M_A is the output of M_B. We illustrate this construction with the following picture.



Take a moment to verify that this reduction from A to B is indeed correct given the definition of f.

Even though a many-one reduction can be viewed as a regular (Turing) reduction, not all reductions are many-one reductions.

Exercise 11.14 (Transitivity of Karp reductions)**.**
Show that if $A \leq_m^P B$ and $B \leq_m^P C$, then $A \leq_m^P C$.

IMPORTANT 11.15 (Steps to establish a Karp reduction)**.**
To show that there is a Karp reduction from A to B, you need to do the following things.

1. Present a computable function $f : \Sigma^* \to \Sigma^*$.
2. Show that $x \in A \implies f(x) \in B$.
3. Show that $x \notin A \implies f(x) \notin B$ (it is usually easier to argue the contrapositive).
4. Show that f can be computed in polynomial time.

Theorem 11.16 (CLIQUE reduces to IS)**.**
CLIQUE $\leq_m^P$ IS.

Proof. Following the previous important note, we start by presenting a computable function $f : \Sigma^* \to \Sigma^*$.

$G = (V, E)$: graph. k: positive integer.
$f(\langle G, k\rangle)$:
1 $E' = \{\{u, v\} : \{u, v\} \notin E\}$.
2 Output $\langle G' = (V, E'), k\rangle$.

(In a Karp reduction from A to B, when we define $f : \Sigma^* \to \Sigma^*$, it is standard to define it so that invalid instances of A are mapped to invalid instances of B. We omit saying this explicitly when presenting the reduction, but you should be aware that this is implicitly there in the definition of f. In the above definition of f, for example, any string x that does not correspond to a valid instance of CLIQUE (i.e., not a valid encoding of a graph G together with a positive integer k) is mapped to an invalid instance of IS (e.g. they can be mapped to ϵ, which we can assume to not be a valid instance of IS.))

To show that f works as desired, we first make a definition. Given a graph $G = (V, E)$, the *complement* of G is the graph $G' = (V, E')$ where $E' = \{\{u, v\} : \{u, v\} \notin E\}$. In other words, we construct G' by removing all the edges of G and adding all the edges that were not present in G.

We now argue that $x \in$ CLIQUE if and only if $f(x) \in$ IS. First, assume $x \in$ CLIQUE. Then x corresponds to a valid encoding $\langle G = (V, E), k\rangle$ of a graph and an integer. Furthermore, G contains a clique $S \subseteq V$ of size k. In the complement graph, this S is an independent set ($\{u, v\} \in E$ for all distinct $u, v \in S$ implies $\{u, v\} \notin E'$ for all distinct $u, v \in S$). Therefore $\langle G' = (V, E'), k\rangle \in$ IS. Conversely, if $\langle G' = (V, E'), k\rangle \in$ IS, then G' contains an independent set $S \subseteq V$ of size k. This set S is a clique in the complement of G', which is G. So the pre-image of $\langle G' = (V, E'), k\rangle$ under f, which is $\langle G = (V, E), k\rangle$, is in CLIQUE.

Finally, we argue that the function f can be computed in polynomial time. This is easy to see since the construction of E' (and therefore G') can be done in polynomial time as there are polynomially many possible edges. □

Exercise 11.17 (IS reduces to CLIQUE)**.**
How can you modify the above reduction to show that IS $\leq_m^P$ CLIQUE?

Exercise 11.18 (Hamiltonian path reductions)**.**
Let $G = (V, E)$ be a graph. A *Hamiltonian path* in G is a path that visits every vertex of the graph exactly once. The HAMPATH problem is the following: given a graph $G = (V, E)$, output True if it contains a Hamiltonian path, and output False otherwise.

(a) Let $L = \{\langle G, k\rangle : G$ is a graph, $k \in \mathbb{N}$, G has a path of length $k\}$. Show that HAMPATH $\leq_m^P L$.

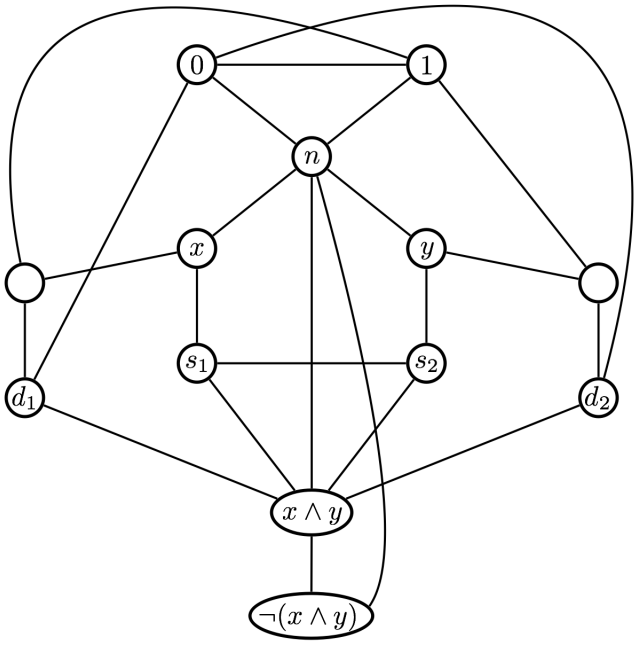
(b) Let $K = \{\langle G, k\rangle : G$ is a graph, $k \in \mathbb{N}$, G has a spanning tree with $\leq k$ leaves$\}$. Show that HAMPATH $\leq_m^P K$.

Note 11.19 (Reductions among unrelated problems)**.**
The theorem below illustrates how reductions can establish an intimate relationship between seemingly unrelated problems.

Theorem 11.20 (CIRCUIT-SAT reduces to 3COL)**.**
CIRCUIT-SAT $\leq_m^P$ 3COL.

Proof. To prove the theorem, we will present a Karp reduction from CIRCUIT-SAT to 3COL. In particular, given a valid CIRCUIT-SAT instance C, we will construct a 3COL instance G such that C is a satisfiable Boolean circuit if and only if G is 3-colorable. Furthermore, the construction will be done in polynomial time.

First, using Exercise 10.7 (NAND is "universal"), we know that any Boolean circuit with AND, OR, and NOT gates can be converted into an equivalent circuit that only has NAND gates (in addition to the input gates and constant gates). This transformation can easily be done in polynomial time. So without loss of generality, we assume that our circuit C is a circuit with NAND gates, input gates and constant gates. We construct G by converting each NAND gate into the following graph.



The vertices labeled with x and y correspond to the inputs of the NAND gate. The vertex labeled with $\neg(x \wedge y)$ corresponds to the output of the gate. We construct such a graph for each NAND gate of the circuit, however, we make sure that if, say, gate g_1 is an input to gate g_2, then the vertex corresponding to the output of g_1 coincides with (is the same as) the vertex corresponding to one of the inputs of g_2. Furthermore, the vertices labeled with 0, 1 and n are the same for each gate. In other words, in the whole graph, there is only one vertex labeled with 0, one vertex labeled with 1, and one vertex labeled with n. Lastly, we put an edge between the vertex corresponding to the output vertex of the output gate and the vertex labeled with 0. This completes the construction of the graph G. Before we prove that the reduction is correct, we make some preliminary observations.

Let's call the 3 colors we use to color the graph 0, 1 and n (we think of n as "none"). Any valid coloring of G must assign different colors to 3 vertices that form a triangle (e.g. vertices labeled with 0, 1 and n). If G is 3-colorable, we can assume without loss generality that the vertex labeled 0 is colored with the color 0, the vertex labeled 1 is colored with color 1, and the vertex labeled n is colored with the color n. This is without loss of generality because if there is a valid coloring of G, any permutation of the colors corresponds to a valid coloring as well. Therefore we can permute the colors so that the labels of those vertices coincide with the colors they are colored with.

Notice that since the vertices corresponding to the inputs of a gate (i.e. the x and y vertices) are connected to vertex n, they will be assigned the colors 0 or 1. Let's consider two cases:

- If x and y are assigned the same color (i.e. either they are both 0 or they are both 1), the vertex labeled with $x \wedge y$ will have to be colored with that same color. That is, the vertex labeled with $x \wedge y$ must get the color

corresponding to the evaluation of $x \wedge y$. To see this, just notice that the vertices labeled s_1 and s_2 must be colored with the two colors that x and y are not colored with. This forces the vertex $x \wedge y$ to be colored with the same color as x and y.

- If x and y are assigned different colors (i.e. one is colored with 0 and the other with 1), the vertex labeled with $x \wedge y$ will have to be colored with 0. That is, as in the first case, the vertex labeled with $x \wedge y$ must get the color corresponding to the evaluation of $x \wedge y$. To see this, just notice that one of the vertices labeled d_1 or d_2 must be colored with 1. This forces the vertex $x \wedge y$ to be colored with 0 since it is already connected to vertex n.

In either case, the color of the vertex $x \wedge y$ must correspond to the evaluation of $x \wedge y$. It is then easy to see that the color of the vertex $\neg(x \wedge y)$ must correspond to the evaluation of $\neg(x \wedge y)$.

We are now ready to argue that circuit C is satisfiable if and only if G is 3-colorable. Let's first assume that the circuit we have is satisfiable. We want to show that the graph G we constructed is 3-colorable. Since the circuit is satisfiable, there is a $0/1$ assignment to the input variables that makes the circuit evaluate to 1. We claim that we can use this $0/1$ assignment to validly color the vertices of G. We start by coloring each vertex that corresponds to an input variable: In the satisfying truth assignment, if an input variable is set to 0, we color the corresponding vertex with the color 0, and if an input variable is set to 1, we color the corresponding vertex with the color 1. As we have argued earlier, a vertex that corresponds to the output of a gate (the vertex at the very bottom of the picture above) is forced to be colored with the color that corresponds to the value that the gate outputs. It is easy to see that the other vertices, i.e., the ones labeled s_1, s_2, d_1, d_2 and the unlabeled vertices can be assigned valid colors. Once we color the vertices in this manner, the vertices corresponding to the inputs and output of a gate will be consistently colored with the values that it takes as input and the value it outputs. Recall that in the construction of G, we connected the output vertex of the output gate with the vertex labeled with 0, which forces it to be assigned the color 1. We know this will indeed happen since the $0/1$ assignment we started with makes the circuit output 1. This shows that we can obtain a valid 3-coloring of the graph G.

The other direction is very similar. Assume that the constructed graph G has a valid 3-coloring. As we have argued before, we can assume without loss of generality that the vertices labeled 0, 1, and n are assigned the colors 0, 1, and n respectively. We know that the vertices corresponding to the inputs of a gate must be assigned the colors 0 or 1 (since they are connected to the vertex labeled n). Again, as argued before, given the colors of the input vertices of a gate, the output vertex of the gate is forced to be colored with the value that the gate would output in the circuit. The fact that we can 3-color the graph means that the output vertex of the output gate is colored with 1 (since it is connected to vertex 0 and vertex n by construction). This implies that the colors of the vertices corresponding to the input variables form a $0/1$ assignment that makes the circuit output a 1, i.e. the circuit is satisfiable.

To finish the proof, we must argue that the construction of graph G, given circuit C, can be done in polynomial time. This is easy to see since for each gate of the circuit, we create at most a constant number of vertices and a constant number of edges. So if the circuit has s gates, the construction can be done in $O(s)$ steps. □

11.2 Hardness and Completeness

Definition 11.21 ($\mathcal{C}$-hard, $\mathcal{C}$-complete)**.**
Let $\mathcal{C}$ be a set of languages containing P.

- We say that L is $\mathcal{C}$-hard (with respect to Cook reductions) if for all languages $K \in \mathcal{C}$, $K \leq^P L$.
  (With respect to polynomial time decidability, a $\mathcal{C}$-hard language is at least as "hard" as any language in $\mathcal{C}$.)
- We say that L is $\mathcal{C}$-complete if L is $\mathcal{C}$-hard and $L \in \mathcal{C}$.
  (A $\mathcal{C}$-complete language represents the "hardest" language in $\mathcal{C}$ with respect to polynomial time decidability.)

Note 11.22 ($\mathcal{C}$-completeness and P)**.**
Suppose L is $\mathcal{C}$-complete. Then observe that $L \in \mathsf{P} \iff \mathcal{C} = \mathsf{P}$.

Note 11.23 ($\mathcal{C}$-hardness with respect to Cook and Karp reductions)**.**
Above we have defined $\mathcal{C}$-hardness using Cook reductions. In literature, however, they are often defined using Karp reductions, which actually leads to a different notion of $\mathcal{C}$-hardness. There are good reasons to use this restricted form of reductions. More advanced courses may explore some of these reasons.

Quiz

1. True or false: $\Sigma^* \leq_m^P \emptyset$.
2. True or false: For languages A and B, $A \leq_m^P B$ if and only if $B \leq_m^P A$.
3. True or false: The language

   $$\text{251CLIQUE} = \{\langle G \rangle : G \text{ is a graph containing a clique of size 251}\}$$

   is in P.
4. True or false: Let $L, K \subseteq \Sigma^*$ be two languages. Suppose there is a polynomial-time computable function $f : \Sigma^* \to \Sigma^*$ such that $x \in L$ iff $f(x) \notin K$. Then L Cook-reduces to K.
5. True or false: There is a Cook reduction from CIRCUIT-SAT to HALTS.

Hints to Selected Exercises

Exercise 11.17 (IS reduces to CLIQUE):
Can you use the same reduction as in the proof of Theorem (CLIQUE reduces to IS).

Exercise 11.18 (Hamiltonian path reductions):
Part (a): A graph has a Hamiltonian path if and only if it has a path of length $n - 1$. Part (b): A Hamiltonian path forms a spanning tree with 2 leaves.

Chapter 12

Non-Deterministic Polynomial Time

PREAMBLE

Chapter structure:

- Section 12.1 (Non-Deterministic Polynomial Time NP)
  - Definition 12.1 (Non-deterministic polynomial time, complexity class NP)
  - Proposition 12.2 (3COL is in NP)
  - Proposition 12.4 (CIRCUIT-SAT is in NP)
  - Proposition 12.8 (P is contained in NP)
  - Definition 12.9 (Complexity class EXP)
- Section 12.2 (NP-complete problems)
  - Theorem 12.12 (Cook-Levin Theorem)
  - Theorem 12.14 (3COL is NP-complete)
  - Theorem 12.15 (3SAT is NP-complete)
  - Theorem 12.16 (CLIQUE is NP-complete)
  - Theorem 12.17 (IS is NP-complete)
- Section 12.3 (Proof of Cook-Levin Theorem)

Chapter goals:

In this chapter, we introduce the famous complexity class NP, which stands for "non-deterministic polynomial time". Recall that P denotes the set of all languages that can be decided in polynomial time. The class NP contains many natural and well-studied languages that we would love to decide in polynomial time. In particular, if we could decide the languages in NP efficiently, this would lead to amazing applications. For instance, in mathematics, proofs to theorems with reasonable length proofs would be found automatically by computers. In artificial intelligence, many machine learning tasks we struggle with would be easy to solve (like vision recognition, speech recognition, language translation and comprehension, etc). Many optimization tasks would become efficiently solvable, which would affect the economy in a major way. Another main impact would happen in privacy and security. We would say "bye" to public-key cryptography which is being used heavily on the internet today. (We will learn about public-key cryptography in a later chapter.) These are just a few examples; there are many more.

Our goal in this chapter is to present the formal definition of NP, and discuss how it relates to P. We will also discuss the notion of NP-completeness (which is intimately related to the question of whether NP equals P) and give several examples of NP-complete languages, along with proofs that they are indeed NP-complete.

12.1 Non-Deterministic Polynomial Time NP

Definition 12.1 (Non-deterministic polynomial time, complexity class NP)**.**
Fix some alphabet Σ. We say that a language L can be decided in *non-deterministic polynomial time* if there exists

(i) a polynomial-time decider TM V that takes two strings as input, and

(ii) a constant $k > 0$,

such that for all $x \in \Sigma^*$:

- if $x \in L$, then there exists $u \in \Sigma^*$ with $|u| \leq |x|^k$ such that $V(x, u)$ accepts,
- if $x \notin L$, then for all $u \in \Sigma^*$, $V(x, u)$ rejects.

If $x \in L$, a string u that makes $V(x, u)$ accept is called a *proof* (or *certificate*) of x being in L. The TM V is called a *verifier*.

We denote by NP the set of all languages which can be decided in non-deterministic polynomial time.

Proposition 12.2 (3COL is in NP)**.**
3COL $\in$ NP.

Proof. To show 3COL is in NP, we need to show that there is a polynomial-time verifier TM A with the properties stated in Definition 12.1 (Non-deterministic polynomial time, complexity class NP) (we are using A to denote the verifier and not V because we will use V to denote the vertex set of a graph). Recall that an instance of the 3COL problem is an undirected graph G. The description of A is as follows.

> $G = (V, E)$: graph.
> $A(\langle G \rangle, u)$:
> 1 If u is not a valid encoding of a 3-coloring of V, reject.
> 2 If there is $\{v, w\} \in E$ where v and w have the same color, reject.
> 3 Else, accept.

We now show that A satisfies the two conditions stated in Definition 12.1 (Non-deterministic polynomial time, complexity class NP). If x is in the language, that means x is a valid encoding of a graph $G = (V, E)$ and this graph is 3-colorable. When u is a valid 3-coloring, $|u| = O(|V|)$. And for this x and u, the verifier accepts. On the other hand, if x is not in the language, then either (i) x is not a valid encoding of a graph or (ii) it is a valid encoding of a graph which is not 3-colorable. In case (i), the verifier rejects (which is done implicitly since the input is not of the correct type). In case (ii), any u that does not correspond to a 3-coloring of the vertices makes the verifier reject. Furthermore, any u that does correspond to a 3-coloring of the vertices must be such that there is an edge whose endpoints are colored with the same color. Therefore, in this case, the verifier again rejects, as desired.

Now we show that the machine is polynomial-time. To check whether u is a valid encoding of a 3-coloring of the vertices takes polynomial time since you just need to check that you are given $|V|$ colors, each being one of 3 colors. To check that it is indeed a valid 3-coloring is polynomial time as well since you just need to go through every edge once.

This completes the proof that 3COL is in NP. □

Note 12.3 (Steps to show a languages is in NP)**.**
Showing that a language L is in NP involves the following steps:

1. Present a TM V (that takes two inputs x and u).
2. Argue that V has polynomial running time.
3. Argue that V works correctly, which involves arguing the following for some constant $k > 0$:
   (a) for all $x \in L$, there exists $u \in \Sigma^*$ with $|u| \leq |x|^k$ such that $V(x, u)$ accepts;
   (b) for all $x \notin L$ and for all $u \in \Sigma^*$, $V(x, u)$ rejects.

Proposition 12.4 (CIRCUIT-SAT is in NP)**.**
CIRCUIT-SAT $\in$ NP.

Proof. To show CIRCUIT-SAT is in NP, we need to show that there is a polynomial-time verifier V with the properties stated in Definition 12.1 (Non-deterministic polynomial time, complexity class NP). We start by presenting V.

C: Boolean circuit.
$V(\langle C \rangle, u)$:
1 If u does not correspond to a valid $0/1$ assignment to input gates, reject.
2 Compute the output of the circuit $C(u)$.
3 If the output is 0, reject.
4 Else, accept.

We first show that the verifier V satisfies the two conditions stated in Definition 12.1 (Non-deterministic polynomial time, complexity class NP). If x is in the language, that means that x corresponds to a valid encoding of a circuit and there is some $0/1$-assignment to the input gates that makes the circuit output 1. When u is such a $0/1$-assignment, then $|u| = O(n)$ (where n is the length of x), and the verifier accepts the input (x, u). On the other hand, if x is not in the language, then either (i) x is not a valid encoding of a circuit or (ii) it is a valid encoding of a circuit which is not satisfiable. In case (i), the verifier rejects (which is done implicitly since the input is not of the correct type). In case (ii), any u that does not correspond to a $0/1$-assignment to the input gates makes the verifier reject. Furthermore, any u that does correspond to a $0/1$-assignment to the input gates must be such that, with this assignment, the circuit evaluates to 0. Therefore, in this case, the verifier again rejects, as desired.

Now we show the verifier is polynomial-time. To check whether u is a valid $0/1$-assignment to the input gates takes polynomial time since you just need to check that you are given t bits, where t is the number of input gates. The output of the circuit can be computed in polynomial time since it takes constant number of steps to compute each gate.

This completes the proof of CIRCUIT-SAT is in NP. □

Exercise 12.5 (CLIQUE is in NP)**.**
Show that CLIQUE $\in$ NP.

Exercise 12.6 (IS is in NP)**.**
Show that IS $\in$ NP.

Exercise 12.7 (3SAT is in NP)**.**
Show that 3SAT $\in$ NP.

Proposition 12.8 (P is contained in NP)**.**
P $\subseteq$ NP.

Proof. Given a language $L \in$ P, we want to show that $L \in$ NP. Since L is in P, we know that there is a polynomial-time decider M that decides L. To show that $L \in$ NP, we need to describe a polynomial-time verifier V that has the properties described in Definition 12.1 (Non-deterministic polynomial time, complexity class NP). The description of V is as follows.

```
V(x, u):
1 Run M(x).
2 If it accepts, accept.
3 Else, reject.
```

First, note that since M is a polynomial time decider, the line "Run $M(x)$" takes polynomial time, and so V is polynomial-time. We now check that V satisfies the two conditions stated in Definition 12.1 (Non-deterministic polynomial time, complexity class NP). If $x \in L$, then $M(x)$ accepts, so for any u, $V(x, u)$ accepts. For example, $V(x, \epsilon)$ accepts, and clearly $|\epsilon| = 0 \leq |x|$. If $x \notin L$, then $M(x)$ rejects, so no matter what u is, $V(x, u)$ rejects, as desired. This shows that $L \in$ NP. □

Definition 12.9 (Complexity class EXP)**.**
We denote by EXP the set of all languages that can be decided in at most exponential-time, i.e., in time $O(2^{n^C})$ for some constant $C > 0$.

Exercise 12.10 (NP is contained in EXP)**.**
Show that NP $\subseteq$ EXP.

12.2 NP-complete problems

Note 12.11 (NP-hardness and NP-completeness)**.**
Recall Definition 11.21 ($\mathcal{C}$-hard, $\mathcal{C}$-complete). We use this definition in this section with $\mathcal{C}$ being NP.

Theorem 12.12 (Cook-Levin Theorem)**.**
CIRCUIT-SAT *is* NP-complete.

IMPORTANT 12.13 (Showing a language is NP-hard)**.**
To show that a language L is NP-hard, by the transitivity of polynomial-time reductions, it suffices to show that $K \leq_m^P L$ for some language K which is known to be NP-hard. In particular, using Theorem 12.12 (Cook-Levin Theorem), it suffices to show that CIRCUIT-SAT $\leq_m^P L$.

Theorem 12.14 (3COL is NP-complete)**.**
3COL *is* NP-complete.

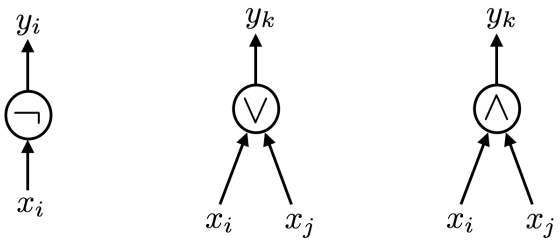
Proof. We have already done all the work to prove that 3COL is NP-complete. First of all, in Proposition 12.2 (3COL is in NP), we have shown that 3COL $\in$ NP. To show that 3COL is NP-hard, by the transitivity of reductions, it suffices to show that CIRCUIT-SAT $\leq_m^P$ 3COL, which we have done in Theorem 11.20 (CIRCUIT-SAT reduces to 3COL). □

Theorem 12.15 (3SAT is NP-complete)**.**
3SAT *is* NP-complete.

Proof Sketch. We sketch the main ideas in the proof. To show that 3SAT is NP-complete, we have to show that it is in NP and it is NP-hard. We leave the proof of membership in NP as an exercise.

To show that 3SAT is NP-hard, by the transitivity of reductions, it suffices to show that CIRCUIT-SAT $\leq_m^P$ 3SAT. Given an instance of CIRCUIT-SAT, i.e. a Boolean circuit C, we will construct an instance of 3SAT, i.e. a Boolean CNF formula φ in which every clause has exactly 3 literals. The reduction will be polynomial-time and will be such that C is a Yes instance of CIRCUIT-SAT (i.e. C is satisfiable) if and only if φ is a Yes instance of 3SAT (i.e. φ is satisfiable).

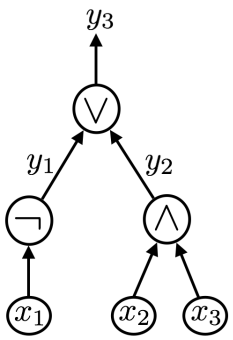
The construction is as follows. A circuit C has three types of gates (excluding the input-gates): NOT, OR, AND.



We will convert each such gate of the circuit C into a 3SAT formula. It is easy to verify that

$$\begin{aligned} y_i = \neg x_i &\iff (x_i \vee y_i) \wedge (\neg x_i \vee \neg y_i), \\ y_k = x_i \vee x_j &\iff (y_k \vee \neg x_i) \wedge (y_k \vee \neg x_j) \wedge (\neg y_k \vee x_i \vee x_j), \\ y_k = x_i \wedge x_j &\iff (\neg y_k \vee x_i) \wedge (\neg y_k \vee x_j) \wedge (y_k \vee \neg x_i \vee \neg x_j). \end{aligned}$$

So the behavior of each gate can be represented using a Boolean formula. As an example, consider the circuit below.



In this case, we would let

$$\begin{aligned}
\text{Clause}_1 &= (x_1 \vee y_1) \wedge (\neg x_1 \vee \neg y_1) \\
\text{Clause}_2 &= (\neg y_2 \vee x_2) \wedge (\neg y_2 \vee x_3) \wedge (y_2 \vee \neg x_2 \vee \neg x_3) \\
\text{Clause}_3 &= (y_3 \vee \neg y_1) \wedge (y_3 \vee \neg y_2) \wedge (\neg y_3 \vee y_1 \vee y_2),
\end{aligned}$$

and the Boolean formula equivalent to the circuit would be

$$\varphi = \text{Clause}_1 \wedge \text{Clause}_2 \wedge \text{Clause}_3 \wedge y_3.$$

This is not quite a 3SAT formula since each clause does not necessarily have exactly 3 literals. However, each clause has at most 3 literals, and every clause in the formula can be converted into a clause with exactly 3 literals by duplicating a literal in the clause if necessary.

This completes the description of how we construct a 3SAT formula from a Boolean circuit. We leave it as an exercise to the reader to verify that C is satisfiable if and only if φ is satisfiable, and that the reduction can be carried out in polynomial time. □

Theorem 12.16 (CLIQUE is NP-complete)**.**
CLIQUE *is* NP-complete.

Proof. To show that CLIQUE is NP-complete, we have to show that it is in NP and it is NP-hard. Exercise 12.5 (CLIQUE is in NP) asks you to show that CLIQUE is in NP, so we will show that CLIQUE is NP-hard by presenting a reduction from 3SAT to CLIQUE.

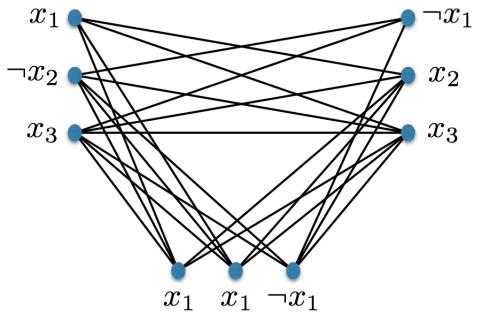
Our reduction will be a Karp reduction. Given an instance of 3SAT, i.e. a Boolean formula φ, we will construct an instance of CLIQUE, $\langle G, k \rangle$ where G is a graph and k is a number, such that φ is a Yes instance of 3SAT (i.e. φ is satisfiable) if and only if $\langle G, k \rangle$ is a Yes instance of CLIQUE (i.e. G contains a k-clique). Furthermore, this construction will be done in polynomial time.

Let

$$\varphi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_m \vee b_m \vee c_m),$$

where each a_i, b_i and c_i is a literal, be an arbitrary 3SAT formula. Notice that φ is satisfiable if and only if there is a truth assignment to the variables so that each clause has at least one literal set to True. From this formula, we build a graph G as follows. For each clause, we create 3 vertices corresponding to the literals of that clause. So in total the graph has $3m$ vertices. We now specify which vertices are connected to each other with an edge. We do *not* put an edge between two vertices if they correspond to the same clause. We do *not* put an edge between x_i and $\neg x_i$ for any i. Every other pair of vertices is connected with an edge. This completes the construction of the graph. We still need to specify k. We set $k = m$.

As an example, if $\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_1 \vee \neg x_1)$, then the corresponding graph is as follows:



We now prove that φ is satisfiable if and only if G has a clique of size m. If φ is satisfiable, then there is an assignment to the variables such that in each clause, there is at least one literal set to True. We claim that the vertices that correspond to these literals form a clique of size m in G. It is clear that the number of such vertices is m. To see that they form a clique, notice that the only way two of these vertices do not share an edge is if they correspond to x_i and $\neg x_i$ for some i. But a satisfying assignment cannot assign True to both x_i and $\neg x_i$.

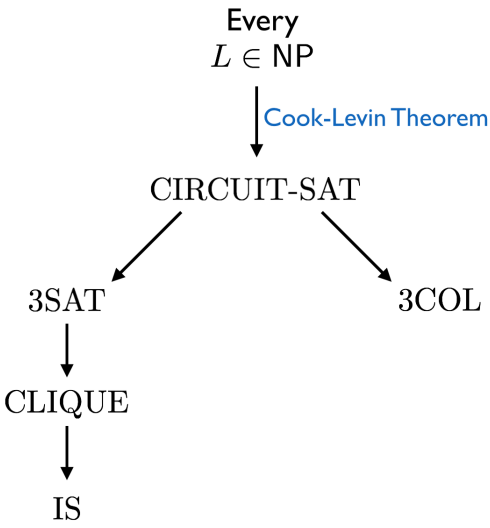
For the other direction, suppose that the constructed graph G has a clique K of size m. Since there are no edges between two literals if they are in the same clause, there must be exactly one vertex from each clause in K. We claim that we can set the literals corresponding to these vertices to True and therefore show that φ is satisfiable. To see this, notice that the only way we could not simultaneously set these literals to True is if two of them correspond to x_i and $\neg x_i$ for some i. But there is no edge between such literals, so they cannot both be in the same clique.

This completes the correctness of the reduction. We still have to argue that it can be done in polynomial time. This is rather straightforward. Creating the vertex set of G is clearly polynomial-time since there is just one vertex for each literal of the Boolean formula. Similarly, the edges can be easily added in polynomial time as there are at most $O(m^2)$ many of them. □

Theorem 12.17 (IS is NP-complete)**.**
IS *is* NP-complete.

Proof. To show that IS is NP-complete, we have to show that it is in NP and it is NP-hard. Exercise 12.6 (IS is in NP) asks you to show that IS is in NP, so we show that IS is NP-hard. By Theorem 12.16 (CLIQUE is NP-complete), we know that CLIQUE is NP-hard, and by Theorem 11.16 (CLIQUE reduces to IS) we know that CLIQUE $\leq_m^P$ IS. By the transitivity of reductions, we conclude that IS is also NP-hard. □

Note 12.18 (Overview of reductions)**.**
The collection of reductions that we have shown can be represented as follows:



Exercise 12.19 (MSAT is NP-complete)**.**
The MSAT problem is defined very similarly to SAT (for the definition of SAT, see Definition 11.5 (Boolean satisfiability problem)). In SAT, we output True if and only if the input CNF formula has at least one satisfying truth assignment to the variables. In MSAT, we want to output True if and only if the input CNF formula has at least two distinct satisfying assignments.

Show that MSAT is NP-complete.

Exercise 12.20 (BIN is NP-complete)**.**
In the PARTITION problem, we are given n non-negative integers $a_1, a_2, \ldots, a_n$, and we want to output True if and only if there is a way to partition the integers into two parts so that the sum of the two parts are equal. In other words, we want to output True if and only if there is set $S \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in S} a_i = \sum_{j \in \{1,\ldots,n\} \setminus S} a_j$.

In the BIN problem we are given a set of n items with non-negative sizes $s_1, s_2, \ldots, s_n \in \mathbb{N}$ (not necessarily distinct), a capacity $C \geq 0$ (not necessarily an integer), and an integer $k \in \mathbb{N}$. We want to output True if and only if the items can be placed into at most k bins such that the total size of items in each bin does not exceed the capacity C.

Show that BIN is NP-complete assuming PARTITION is NP-hard.

12.3 Proof of Cook-Levin Theorem

Proof of Cook-Levin Theorem. Using Theorem (Efficient TM implies efficient circuit), we can prove the famous Cook-Levin Theorem which states that CIRCUIT-SAT is NP-complete. We present the proof below.

To prove this theorem, we have to show that CIRCUIT-SAT is in NP and that it is NP-hard. We have already shown that CIRCUIT-SAT is in NP in Proposition 12.4 (CIRCUIT-SAT is in NP). To show that is is NP-hard, we will show that for any $L \in$ NP, $L \leq_m^P$ CIRCUIT-SAT.

Let L be an arbitrary language in NP. We will define a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ that maps $x \in \{0,1\}^*$ to a circuit $\langle C_x \rangle$ such that

$$x \in L \iff C_x \text{ is satisfiable.}$$

Since L is in NP, there is a polynomial-time verifier TM V and constants c, k such that:

$$x \in L \iff \exists u, |u| = c|x|^k, V(x,u) = 1.$$

(Here we insist that the proof is of length exactly $c|x|^k$, which is fine to do and does not change the definition of NP.)

Note that V is just a regular decider TM. Using Theorem (Efficient TM implies efficient circuit), we know that there exists a polynomial-size circuit family that simulates V. Let $C = C(x,u)$ be the circuit in this family with $|x| + c|x|^k$ input gates (this is a circuit with input gates corresponding to the x-variables as well as the u-variables). For $x \in \{0,1\}^*$, let C_x be the circuit C where the input gates x_i are replaced with constant gates corresponding to the actual values of the x_i's. So C_x is a circuit where the input gates correspond to the u-variables only. Our function $f : \{0,1\}^* \to \{0,1\}^*$ thus maps x to $\langle C_x \rangle$. Observe that

$$\begin{aligned} x \in L &\iff \exists u, |u| = c|x|^k, V(x,u) = 1 \\ &\iff C_x \text{ is satisfiable.} \end{aligned}$$

This shows $x \in L$ if and only if $\langle C_x \rangle \in$ CIRCUIT-SAT, as desired.

It is not hard to argue that $\langle C_x \rangle$ can be constructed in polynomial time (which we leave as an exercise to the reader). □

Quiz

1. True or false: Every language in NP is decidable.
2. True or false: Let Σ be an alphabet. Then $\Sigma^* \in \mathsf{NP}$.
3. True or false: $\{0^k 1^k : k \in \mathbb{N}\} \in \mathsf{NP}$.
4. True or false: If there is a polynomial-time algorithm for 3COL, then $\mathsf{P} = \mathsf{NP}$.
5. True or false: For languages A and B, if $A \leq_m^P B$ and $B \notin \mathsf{NP}$, then $A \notin \mathsf{NP}$.
6. True or false: HALTS is NP-complete.

Chapter 13

Approximation Algorithms

PREAMBLE

Chapter structure:

- Section 14.1.1 (Basic Definitions)
  - Definition 13.1 (Optimization problem)
  - Definition 13.2 (Optimization version of the Vertex-cover problem)
  - Definition 13.5 (Approximation algorithm)
- Section 13.2 (Examples of Approximation Algorithms)
  - Lemma 13.7 (Vertex cover vs matching)
  - Theorem 13.8 (Gavril's Algorithm)
  - Definition 13.10 (Max-cut problem)
  - Theorem 13.11 ($(1/2)$-approximation algorithm for MAX-CUT)
  - Definition 13.12 (Traveling salesperson problem (TSP))
  - Theorem 13.13 (2-approximation algorithm for Metric-TSP)
  - Definition 13.14 (Max-coverage problem)

Chapter goals:

Given that many problems that we would like to solve efficiently are NP-hard, we do not hope to find a polynomial time algorithm that exactly solves the problem for all inputs. This doesn't mean that we should give up. Rather, it means that we should look for efficient solutions that we might consider "good enough", by relaxing some of the requirements. For instance, we could relax the condition that the algorithm outputs a correct answer for all inputs. Or perhaps, we would be happy if the algorithm gave us an approximately good solution.

In this chapter, we study *approximation algorithms*, which give us ways to deal with NP-hard problems. Approximation algorithms do not necessarily give us exact solutions, but give us approximately good solutions with guarantees on how close the output of the algorithm is to a desired solution. To make all of this precise, we first start with the definition of an *optimization problem*. Afterwards, we give the formal definition of an approximation algorithm, and present 3 examples. Our goal is for you to develop a basic level of comfort with thinking about and analyzing approximation algorithms.

13.1 Basic Definitions

Definition 13.1 (Optimization problem)**.**
A *minimization optimization problem* is a function $f : \Sigma^* \times \Sigma^* \to \mathbb{R}^{\geq 0} \cup \{\text{no}\}$. If $f(x, y) = \alpha \in \mathbb{R}^{\geq 0}$, we say that y is a *solution* to x with value α. If $f(x, y) = \text{no}$, then y is not a solution to x. We let $\text{OPT}_f(x)$ denote the minimum $f(x, y)$ among all solutions y to x.[1] We drop the subscript f, and just write $\text{OPT}(x)$, when f is clear from the context.

In a *maximization optimization problem*, $\text{OPT}_f(x)$ is defined using a maximum rather than a minimum.

We say that an optimization problem f is computable if there is an algorithm such that given as input $x \in \Sigma^*$, it produces as output a solution y to x such that $f(x, y) = \text{OPT}(x)$. We often describe an optimization problem by describing the input and a corresponding output (i.e. a solution y such that $f(x, y) = \text{OPT}(x)$).

Definition 13.2 (Optimization version of the Vertex-cover problem)**.**
Given an undirected graph $G = (V, E)$, a *vertex cover* in G is a set $S \subseteq V$ such that for all edges in E, at least one of its endpoints is in S.[2]

The VERTEX-COVER problem is the following. Given as input an undirected graph G together with an integer k, output True if and only if there is a vertex cover in G of size at most k. The corresponding language is

$$\{\langle G, k\rangle : G \text{ is a graph that has a vertex cover of size at most } k\}.$$

In the optimization version of VERTEX-COVER, we are given as input an undirected graph G and the output is a vertex cover of minimum size. We refer to this problem as MIN-VC.

Using the notation in Definition 13.1 (Optimization problem), the corresponding function f is defined as follows. Let $x = \langle G\rangle$ for some graph G. If y represents a vertex cover in G, then $f(x, y)$ is defined to be the size of the set that y represents. Otherwise $f(x, y) = \text{no}$.

Note 13.3 (Examples of optimization problems)**.**
Each decision problem that we have defined in the beginning of Chapter 11 (Polynomial-Time Reductions) has a natural optimization version.

Note 13.4 (NP-hardness for optimization problems)**.**
The complexity class NP is a set of decision problems (or languages). Similarly, the set of NP-hard problems is a set of decision problems. Given an optimization problem f, suppose it is the case that if f can be computed in polynomial time, then every decision problem in NP can be decided in polynomial time. In this case, we will abuse the definition of NP-hard and say that f is NP-hard.

Definition 13.5 (Approximation algorithm)**.**

- Let f be a minimization optimization problem and let $\alpha > 1$ be some parameter. We say that an algorithm A is an α-approximation algorithm for f if for all instances x, $f(x, A(x)) \leq \alpha \cdot \text{OPT}(x)$.

[1] There are a few technicalities. We will assume that f is such that every x has at least one solution y, and that the minimum always exists.

[2] We previously called such a set a *popular set*.

- Let f be a maximization optimization problem and let $0 < \beta < 1$ be some parameter. We say that an algorithm A is a β-approximation algorithm for f if for all instances x, $f(x, A(x)) \geq \beta \cdot \text{OPT}(x)$.

IMPORTANT 13.6 (Analyzing approximation algorithms)**.**
When showing that a certain minimization problem has an α-approximation algorithm, you need to first present an algorithm A, and then argue that for any input x, the value of the output produced by the algorithm is within a factor α of the optimum:

$$f(x, A(x)) \leq \alpha \cdot \text{OPT}(x).$$

When doing this, it is usually hard to know exactly what the optimum value would be. So a good strategy is to find a convenient *lower bound* on the optimum, and then argue that the output of the algorithm is within a factor α of this lower bound. In other words, if $\text{LB}(x)$ denotes the lower bound (so $\text{LB}(x) \leq \text{OPT}(x)$), we want to argue that

$$f(x, A(x)) \leq \alpha \cdot \text{LB}(x).$$

For example, for the MIN-VC problem, we will use Lemma 13.7 (Vertex cover vs matching) below to say that the optimum (the size of the minimum size vertex cover) is lower bounded by the size of a matching in the graph.

The same principle applies to maximization problems as well. For maximization problems, we want to find a convenient *upper bound* on the optimum.

13.2 Examples of Approximation Algorithms

Lemma 13.7 (Vertex cover vs matching)**.**
Given a graph $G = (V, E)$, let $M \subseteq E$ be a matching in G, and let $S \subset V$ be a vertex cover in G. Then, $|S| \geq |M|$.

Proof. Observe that in a vertex cover, one vertex cannot be incident to more than one edge of a matching. Therefore, a vertex cover must have at least $|M|$ vertices in order to touch every edge of M. (Recall that the size of a matching, $|M|$, is the number of edges in the matching.) □

Theorem 13.8 (Gavril's Algorithm)**.**
There is a polynomial-time 2-approximation algorithm for the optimization problem MIN-VC.

Proof. We start by presenting the algorithm, which greedily chooses a *maximal* matching M in the graph, and then outputs all the vertices that are incident to an edge in M.

$G = (V, E)$: undirected graph.
$A(\langle G \rangle)$:
1 Let $M = \emptyset$.
2 For each edge $e \in E$ do:
3 If $M \cup \{e\}$ is a matching, let $M = M \cup \{e\}$.
4 Let $S =$ set of all the vertices incident to an edge in M.
5 Output S.

We need to argue that the algorithm runs in polynomial time and that it is a 2-approximation algorithm. It is easy to see that the running-time is polynomial. We have a loop that repeats $|E|$ times, and in each iteration, we do at most $O(|E|)$ steps. So the total cost of the loop is $O(|E|^2)$. The construction of S takes $O(|V|)$ steps, so in total, the algorithm runs in polynomial time.

Now we argue that the algorithm is a 2-approximation algorithm. To do this, we need to argue that

(i) S is indeed a valid vertex-cover,

(ii) if S^* is a vertex cover of minimum size, then $|S| \leq 2|S^*|$.

For (i), notice that the M constructed by the algorithm is a maximal matching, i.e., there is no edge $e \in E$ such that $M \cup \{e\}$ is a matching. This implies that the set S is indeed a valid vertex-cover, i.e., it touches every edge in the graph. For (ii), a convenient lower bound on $|S^*|$ is given by Lemma 13.7 (Vertex cover vs matching): for any matching M, $|S^*| \geq |M|$. Observe that $|S| = 2|M|$. Putting the two together, we get $|S| \leq 2|S^*|$ as desired. □

Exercise 13.9 (Optimality of the analysis of Gavril's Algorithm)**.**
Describe an infinite family of graphs for which the above algorithm returns a vertex cover which has twice the size of a minimum vertex cover.

Definition 13.10 (Max-cut problem)**.**
Let $G = (V, E)$ be a graph. Given a coloring of the vertices with 2 colors, we say that an edge $e = \{u, v\}$ is *cut* if u and v are colored differently. In the *max-cut problem*, the input is a graph G, and the output is a coloring of the vertices with 2 colors that maximizes the number of cut edges. We denote this problem by MAX-CUT.

Theorem 13.11 ($(1/2)$-approximation algorithm for MAX-CUT)**.**
There is a polynomial-time $\frac{1}{2}$-approximation algorithm for the optimization problem MAX-CUT.

Proof. Here is the algorithm:

$G = (V, E)$: undirected graph.
$A(\langle G \rangle)$:
1 Color every vertex with the same color. Let $c = 0$. (c stores the number of cut edges.)
2 Repeat:
3 If there is a vertex such that changing its color increases the number of cut edges, change the color of that vertex. Update c.
4 Else, output the current coloring of the vertices.

We first argue that the algorithm runs in polynomial time. Note that the maximum number of cut edges possible is $|E|$. Therefore the loop repeats at most $|E|$ times. In each iteration, the number of steps we need to take is at most $O(|V|^2)$ since we can just go through every vertex once, and for each one of them, we can check all the edges incident to it. So in total, the number of steps is polynomial in the input length.

We now show that the algorithm is a $\frac{1}{2}$-approximation algorithm. It is clear that the algorithm returns a valid coloring of the vertices. Therefore, if c is the number of cut edges returned by the algorithm, all we need to show is that $c \geq \frac{1}{2}\text{OPT}(\langle G \rangle)$. We will use the trivial upper bound of m (the total number of edges) on $\text{OPT}(\langle G \rangle)$, i.e. $\text{OPT}(\langle G \rangle) \leq m$. So our goal will be to show $c \geq \frac{1}{2}m$.

Observe that in the coloring that the algorithm returns, for each $v \in V$, at least $\deg(v)/2$ edges incident to v are cut edges. To see this, notice that if there was a vertex such that this was not true, then we could change the color of the vertex to obtain a solution that has strictly more cut edges, so our algorithm would have changed the color of this vertex. From Theorem 8.9 (Handshake Theorem), we know that when we count the number of edges of a graph by adding up the degrees of all the vertices, we count every edge exactly twice, i.e. $2m = \sum_v \deg(v)$. In a similar way we can count the number of cut edges, which implies $2c \geq \sum_v \deg(v)/2$. The RHS of this inequality is equal to m, so we have $c \geq \frac{1}{2}m$, as desired. □

Definition 13.12 (Traveling salesperson problem (TSP))**.**
In the *Traveling salesperson problem*, the input is a connected graph $G = (V, E)$ together with edge costs $c : E \to \mathbb{N}$. The output is a Hamiltonian cycle that minimizes the total cost of the edges in the cycle, if one exists.

A popular variation of this problem is called *Metric-TSP*. In this version of the problem, instead of outputting a Hamiltonian cycle of minimum cost, we output a "tour" that starts and ends at the same vertex and visits every vertex of the graph at least once (so the tour is allowed to visit a vertex more than once). In other words, the output is a list of vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}, v_{i_1}$ such that the vertices are not necessarily unique, all the vertices of the graph appear in the list, any two consecutive vertices in the list form an edge, and the total cost of the edges is minimized.

Theorem 13.13 (2-approximation algorithm for Metric-TSP)**.**
There is a polynomial-time 2*-approximation algorithm for* Metric-TSP*.*

Proof. The algorithm first computes a minimum spanning tree, and then does a depth-first search on the tree starting from an arbitrary vertex. More precisely:

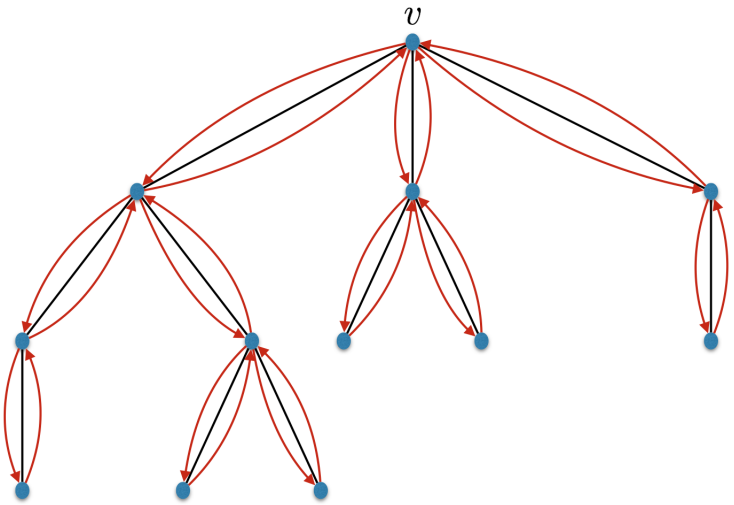
$G = (V, E)$: connected graph. $c : E \to \mathbb{N}$: function.
$A(\langle G, c \rangle)$:
1. Compute a MST T of G.
2. Let v be an arbitrary vertex in V.
3. Let L be an empty *list*.
4. Run DFS($\langle T, v \rangle$).

> $G = (V, E)$: graph. v: $v \in V$.
> DFS($\langle G, v \rangle$):
> 1. Mark v as "visited".
> 2. Add v to L.
> 3. For each $u \in N(v)$:
> 4. If u is not marked "visited", then run DFS($\langle G, u \rangle$).
> 5. Add v to L.

5. Output L.

This is clearly a polynomial-time algorithm since computing a minimum spanning tree (Theorem 8.34 (Jarník-Prim algorithm for MST)) and doing a depth-first search both take polynomial time.

To see that the algorithm outputs a valid tour, note that it visits every vertex (since T is a spanning tree), and it starts and ends at the same vertex v.



Let $c(L)$ denote the total cost of the tour that the algorithm outputs. Let L^* be a optimal solution (so $c(L^*) = \text{OPT}(\langle G, c\rangle)$). Our goal is to show that $c(L) \leq 2c(L^*)$. The graph induced by L^* is a connected graph on all of the vertices. Let T^* be a spanning tree within this induced graph. It is clear that $c(L^*) \geq c(T^*)$ and this will be the convenient lower bound we use on the optimum. In other words, we'll show $c(L) \leq 2c(T^*)$. Clearly $c(L) = 2c(T)$ since the tour uses every edge of T exactly twice. Furthermore, since T is a *minimum* spanning tree, $c(T) \leq c(T^*)$. Putting these together, we have $c(L) \leq 2c(T^*)$, as desired. □

Definition 13.14 (Max-coverage problem)**.**
In the *max-coverage problem*, the input is a set X, a collection of (possibly intersecting) subsets $S_1, S_2, \ldots, S_m \subseteq X$ (we assume the union of all the sets is X), and a number $k \in \{0, 1, \ldots, m\}$. The output is a set $T \subseteq \{1, 2, \ldots, m\}$ of size k that maximizes $|\cup_{i \in T} S_i|$ (the elements in this intersection are called *covered elements*). We denote this problem by MAX-COVERAGE.

Exercise 13.15 (Approximation algorithm for MAX-COVERAGE)**.**
In this exercise, you will prove that there is a polynomial-time $\left(1 - \frac{1}{e}\right)$-approximation algorithm for the MAX-COVERAGE problem. The algorithm you should consider is the following greedy algorithm:

$S_1, S_2, \ldots, S_m$: sets. k: integer in $\{0, 1, \ldots, m\}$.
$A(\langle S_1, \ldots, S_m, k\rangle)$:
1 $T = \emptyset$.
2 $U = \emptyset$. (keeping track of elements covered)
3 Repeat k times:
4 Pick j such that $j \notin T$ and $|S_j - U|$ is maximized.
5 Add j to T.
6 Update U to $U \cup S_j$.

7 Output T.

(a) Show that the algorithm runs in polynomial time.

(b) Let T^* denote the optimum solution, and let $U^* = \cup_{j \in T^*} S_j$. Note that the value of the optimum solution is $|U^*|$. Define U_i to be the set U in the above algorithm after i iterations of the loop. Let $r_i = |U^*| - |U_i|$. Prove that $r_i \leq \left(1 - \frac{1}{k}\right)^i |U^*|$.

(c) Using the inequality[3] $1 - 1/k \leq e^{-1/k}$, conclude that the algorithm is a $\left(1 - \frac{1}{e}\right)$-approximation algorithm for the MAX-COVERAGE problem.

Exercise 13.16 (Approximation algorithm for MIN-SET-COVER)**.**
In the *set-cover problem*, the input is a set X together with a collection of (possibly intersecting) subsets $S_1, S_2, \ldots, S_m \subseteq X$ (we assume the union of all the sets is X). The output is a minimum size set $T \subseteq \{1, 2, \ldots, m\}$ such that $\cup_{i \in T} S_i = X$. We denote this problem by MIN-SET-COVER. Give a polynomial-time $(\ln |X|)$-approximation algorithm for this problem.

[3] This can be derived from the Taylor expansion of e^x.

Quiz

1. True or false: Suppose A is an α-approximation algorithm for the MAX-CLIQUE problem for some $\alpha < 1$. Then it must be the case that for all input graphs $G = (V, E)$, the size of the clique returned by A is at least $\alpha \cdot |V|$.

2. True or false: The approximation algorithm for Metric-TSP that we have seen is not a $(2 - \epsilon)$-approximation algorithm for any constant $\epsilon > 0$.

3. True or false: Let A_1 be a 2-approximation algorithm for MIN-VC, and let A_2 be a 4-approximation algorithm for MIN-VC. Define a new approximation algorithm A_3 that runs A_1 and A_2 on the given MIN-VC instance, and outputs the smaller among the two vertex covers they find. Then A_3 is a 2-approximation algorithm.

4. True or we don't know: Let A be a polynomial-time algorithm for MIN-VC such that the output of A is within a factor of 1.9 of the optimum for all but 251 possible inputs. Then we cannot say that A is a 1.9-approximation algorithm for MIN-VC. But we can conclude that there is definitely a polynomial-time 1.9-approximation algorithm for MIN-VC.

Hints to Selected Exercises

Exercise 13.16 (Approximation algorithm for MIN-SET-COVER):
The algorithm is the same as the one given in the previous exercise. Instead of repeating for a fixed number of times, repeat until you cover every element.

Chapter 14

Probability Theory

PREAMBLE

Chapter structure:

- Section 14.1 (Probability I: The Basics)
  - Definition 14.1 (Finite probability space, sample space, probability distribution)
  - Definition 14.6 (Uniform distribution)
  - Definition 14.7 (Event)
  - Definition 14.10 (Disjoint events)
  - Definition 14.12 (Conditional probability)
  - Proposition 14.15 (Chain rule)
  - Proposition 14.17 (Law of total probability)
  - Proposition 14.20 (Bayes' rule)
  - Definition 14.22 (Independent events)
- Section 14.2 (Probability II: Random Variables)
  - Definition 14.25 (Random variable)
  - Definition 14.27 (Common events through a random variable)
  - Definition 14.30 (Probability mass function (PMF))
  - Definition 14.33 (Expected value of a random variable)
  - Proposition 14.36 (Linearity of expectation)
  - Corollary 14.37 (Linearity of expectation 2)
  - Definition 14.40 (Indicator random variable)
  - Proposition 14.41 (Expectation of an indicator random variable)
  - Definition 14.44 (Conditional expectation)
  - Proposition 14.45 (Law of total expectation)
  - Definition 14.48 (Independent random variables)
  - Theorem 14.50 (Markov's inequality)
  - Definition 14.52 (Bernoulli random variable)
  - Definition 14.54 (Binomial random variable)
  - Definition 14.58 (Geometric random variable)

Chapter goals:

Randomness is an essential concept and tool in modeling and analyzing nature. Therefore, it should not be surprising that it also plays a foundational role in computer science. For many problems, solutions that make use of randomness are the simplest, most efficient and most elegant solutions. And in many settings, one can prove that randomness is absolutely required to achieve a solution. (We will mention some concrete examples in lecture.)

The right language and mathematical model to analyze/study randomization is probability theory. The goal of this chapter is to introduce the basic definitions and theorems in this field.

14.1 Probability I: The Basics

14.1.1 Basic Definitions

Definition 14.1 (Finite probability space, sample space, probability distribution)**.**
A *finite probability space* is a tuple $(\Omega, \mathbf{Pr})$, where

- Ω is a non-empty finite set called the *sample space*;
- $\mathbf{Pr} : \Omega \to [0,1]$ is a function, called the *probability distribution*, with the property that $\sum_{\ell \in \Omega} \mathbf{Pr}[\ell] = 1$.

The elements of Ω are called *outcomes* or *samples*. If $\mathbf{Pr}[\ell] = p$, then we say that *the probability of outcome ℓ is p.*

Note 14.2 (Modeling randomness)**.**
The abstract definition above of a finite probability space helps us to mathematically model and reason about situations involving randomness and uncertainties (these situations are often called "random experiments" or just "experiments"). For example, consider the experiment of flipping a single coin. We model this as follows. We let $\Omega = \{\text{Heads}, \text{Tails}\}$ and we define function $\mathbf{Pr}$ such that $\mathbf{Pr}[\text{Heads}] = 1/2$ and $\mathbf{Pr}[\text{Tails}] = 1/2$. This corresponds to our intuitive understanding that the probability of seeing the outcome Heads is $1/2$ and the probability of seeing the outcome Tails is also $1/2$.

Note 14.3 (Restriction to finite sample spaces)**.**
In this course, we'll usually restrict ourselves to finite sample spaces. In cases where we need an infinite Ω, the above definition will generalize naturally.

Exercise 14.4 (Probability space modeling)**.**
How would you model a roll of a single 6-sided die using Definition 14.1 (Finite probability space, sample space, probability distribution)? How about a roll of two dice? How about a roll of a die and a coin toss together?

Note 14.5 (Modeling through randomized code)**.**
Sometimes, the modeling of a real-world random experiment as a probability space can be non-trivial or tricky. It helps a lot to have a step in between where you first go from the real-world experiment to computer code/algorithm (that makes calls to random number generators), and then you define your probability space based on the computer code. In this course, we allow our programs to have access to the functions Bernoulli(p) and RandInt(n). The function Bernoulli(p) takes a number $0 \leq p \leq 1$ as input and returns 1 with probability p and 0 with probability $1-p$. The function RandInt(n) takes a positive integer n as input and returns a random integer from 1 to n (i.e., every number from 1 to n has probability $1/n$). Here is a very simple example of going from a real-world experiment to computer code. The experiment is as follows. You flip a fair coin. If it's heads, you roll a 3-sided die. If it is tails, you roll a 4-sided die. This experiment can be represented as:

```
flip = Bernoulli(1/2)
if flip = 0:
    die = RandInt(3)
else:
    die = RandInt(4)
```

If we were to ask "What is the probability that you roll a 3 or higher?", this would correspond to asking what is the probability that after the above code is executed, the variable named `die` stores a value that is 3 or higher. This simple example does not illustrate the usefulness of having a computer code representation of the random experiment, but one can appreciate its value with more sophisticated examples and we do encourage you to think of random experiments as computer code:

real-world experiment $\longrightarrow$ computer code $\longrightarrow$ probability space $(\Omega, \mathbf{Pr})$.

Definition 14.6 (Uniform distribution)**.**
If a probability distribution $\mathbf{Pr} : \Omega \to [0,1]$ is such that $\mathbf{Pr}[\ell] = 1/|\Omega|$ for all $\ell \in \Omega$, then we call it a *uniform distribution*.

Definition 14.7 (Event)**.**
Let $(\Omega, \mathbf{Pr})$ be a probability space. Any subset of outcomes $E \subseteq \Omega$ is called an *event*. We abuse notation and write $\mathbf{Pr}[E]$ to denote $\sum_{\ell \in E} \mathbf{Pr}[\ell]$. Using this notation, $\mathbf{Pr}[\emptyset] = 0$ and $\mathbf{Pr}[\Omega] = 1$. We use the notation $\overline{E}$ to denote the event $\Omega \backslash E$.

Exercise 14.8 (Practice with events)**.**

(a) Suppose we roll two 6-sided dice. How many different events are there? Write down the event corresponding to "we roll a double" and determine its probability.

(b) Suppose we roll a 3-sided die and see the number d. We then roll a d-sided die. How many different events are there? Write down the event corresponding to "the second roll is a 2" and determine its probability.

Exercise 14.9 (Basic facts about probability)**.**
Let A and B be two events. Prove the following.

- If $A \subseteq B$, then $\mathbf{Pr}[A] \leq \mathbf{Pr}[B]$.
- $\mathbf{Pr}[\overline{A}] = 1 - \mathbf{Pr}[A]$.
- $\mathbf{Pr}[A \cup B] = \mathbf{Pr}[A] + \mathbf{Pr}[B] - \mathbf{Pr}[A \cap B]$.

Definition 14.10 (Disjoint events)**.**
We say that two events A and B are *disjoint* if $A \cap B = \emptyset$.

Exercise 14.11 (Union bound)**.**
Let $A_1, A_2, \ldots, A_n$ be events. Then

$$\mathbf{Pr}[A_1 \cup A_2 \cup \cdots \cup A_n] \leq \mathbf{Pr}[A_1] + \mathbf{Pr}[A_2] + \cdots + \mathbf{Pr}[A_n].$$

We get equality if and only if the A_i's are pairwise disjoint.

Definition 14.12 (Conditional probability)**.**
Let B be an event with $\mathbf{Pr}[B] \neq 0$. The *conditional probability of outcome* $\ell \in \Omega$ *given* B, denoted $\mathbf{Pr}[\ell \mid B]$, is defined as

$$\mathbf{Pr}[\ell \mid B] = \begin{cases} 0 & \text{if } \ell \notin B \\ \frac{\mathbf{Pr}[\ell]}{\mathbf{Pr}[B]} & \text{if } \ell \in B \end{cases}$$

For an event A, the *conditional probability of* A *given* B, denoted $\mathbf{Pr}[A \mid B]$, is defined as

$$\mathbf{Pr}[A \mid B] = \frac{\mathbf{Pr}[A \cap B]}{\mathbf{Pr}[B]}. \tag{14.1}$$

Note 14.13 (Intuitive understanding of conditional probability)**.**
Although it may not be immediately obvious, the above definition of conditional probability does correspond to our intuitive understanding of what conditional probability should represent. If we are told that event B has already happened, then we know that the probability of any outcome outside of B should be 0. Therefore, we can view the conditioning on event B as a transformation of our probability space where we revise the probabilities (i.e., we revise the probability function $\mathbf{Pr}[\cdot]$). In particular, the original probability space $(\Omega, \mathbf{Pr})$ gets transformed to $(\Omega, \mathbf{Pr}_B)$, where $\mathbf{Pr}_B$ is such that for any $\ell \notin B$, we have $\mathbf{Pr}_B[\ell] = 0$, and for any $\ell \in B$, we have $\mathbf{Pr}_B[\ell] = \mathbf{Pr}[\ell]/\mathbf{Pr}[B]$. The $1/\mathbf{Pr}[B]$ factor here is a necessary *normalization* factor that ensures the probabilities of all the outcomes sum to 1 (which is required by Definition 14.1 (Finite probability space, sample space, probability distribution)). Indeed

$$\begin{aligned} \textstyle\sum_{\ell \in \Omega} \mathbf{Pr}_B[\ell] &= \textstyle\sum_{\ell \notin B} \mathbf{Pr}_B[\ell] + \sum_{\ell \in B} \mathbf{Pr}_B[\ell] \\ &= 0 + \textstyle\sum_{\ell \in B} \mathbf{Pr}[\ell]/\mathbf{Pr}[B] \\ &= \frac{1}{\mathbf{Pr}[B]} \textstyle\sum_{\ell \in B} \mathbf{Pr}[\ell] \\ &= 1. \end{aligned}$$

If we are interested in the event "A given B" (denoted by $A \mid B$) in the probability space $(\Omega, \mathbf{Pr})$, then we are interested in the event A in the probability space $(\Omega, \mathbf{Pr}_B)$. That is, $\mathbf{Pr}[A \mid B] = \mathbf{Pr}_B[A]$. Therefore,

$$\mathbf{Pr}[A \mid B] = \mathbf{Pr}_B[A] = \mathbf{Pr}_B[A \cap B] = \frac{\mathbf{Pr}[A \cap B]}{\mathbf{Pr}[B]},$$

where the last equality holds by the definition of $\mathbf{Pr}_B[\cdot]$. We have thus recovered the equality in Definition 14.12 (Conditional probability).

Conditioning on event B can also be viewed as redefining the sample space Ω to be B, and then renormalizing the probabilities so that $\mathbf{Pr}[\Omega] = \mathbf{Pr}[B] = 1$.

Exercise 14.14 (Conditional probability practice)**.**
Suppose we roll a 3-sided die and see the number d. We then roll a d-sided die. We are interested in the probability that the first roll was a 1 given that the second roll was a 1. First express this probability using the notation of conditional probability and then determine what the probability is.

14.1.2 Three Useful Rules

Proposition 14.15 (Chain rule)**.**
Let $n \geq 2$ and let $A_1, A_2, \ldots, A_n$ be events. Then

$$\mathbf{Pr}[A_1 \cap \cdots \cap A_n] = \mathbf{Pr}[A_1] \cdot \mathbf{Pr}[A_2 \mid A_1] \cdot \mathbf{Pr}[A_3 \mid A_1 \cap A_2] \cdots \mathbf{Pr}[A_n \mid A_1 \cap A_2 \cap \cdots \cap A_{n-1}].$$

Proof. We prove the proposition by induction on n. The base case with two events follows directly from the definition of conditional probability. Let $A = A_n$ and $B = A_1 \cap \ldots \cap A_{n-1}$. Then

$$\begin{aligned}\mathbf{Pr}[A_1 \cap \cdots \cap A_n] &= \mathbf{Pr}[A \cap B] \\ &= \mathbf{Pr}[B] \cdot \mathbf{Pr}[A \mid B] \\ &= \mathbf{Pr}[A_1 \cap \cdots \cap A_{n-1}] \cdot \mathbf{Pr}[A_n \mid A_1 \cap \cdots \cap A_{n-1}],\end{aligned}$$

where we used the definition of conditional probability for the second equality. Applying the induction hypothesis to $\mathbf{Pr}[A_1 \cap \cdots \cap A_{n-1}]$ gives the desired result. □

Exercise 14.16 (Practice with chain rule)**.**
Suppose there are 100 students in 15-251 and 5 of the students are Trump supporters. We pick 3 students from class uniformly at random. Calculate the probability that none of them are Trump supporters using Proposition 14.15 (Chain rule).

Proposition 14.17 (Law of total probability)**.**
Let $A_1, A_2, \ldots, A_n, B$ be events such that the A_i's form a partition of the sample space Ω. Then

$$\mathbf{Pr}[B] = \mathbf{Pr}[B \cap A_1] + \mathbf{Pr}[B \cap A_2] + \cdots + \mathbf{Pr}[B \cap A_n].$$

Equivalently,

$$\mathbf{Pr}[B] = \mathbf{Pr}[A_1] \cdot \mathbf{Pr}[B \mid A_1] + \mathbf{Pr}[A_2] \cdot \mathbf{Pr}[B \mid A_2] + \cdots + \mathbf{Pr}[A_n] \cdot \mathbf{Pr}[B \mid A_n].$$

Exercise 14.18 (Proof of the law of total probability)**.**
Prove the above proposition.

Exercise 14.19 (Practice with the law of total probability)**.**
There are 2 bins. Bin 1 contains 6 red balls and 4 blue balls. Bin 2 contains 3 red balls and 7 blue balls. We choose a bin uniformly at random, and then choose one of the balls in that bin uniformly at random. Calculate the probability that the chosen ball is red using Proposition 14.17 (Law of total probability).

Proposition 14.20 (Bayes' rule)**.**
Let A and B be events. Then,

$$\mathbf{Pr}[A \mid B] = \frac{\mathbf{Pr}[A] \cdot \mathbf{Pr}[B \mid A]}{\mathbf{Pr}[B]}.$$

Proof. Since by definition $\mathbf{Pr}[B \mid A] = \mathbf{Pr}[A \cap B]/\mathbf{Pr}[A]$, the RHS of the equality above simplifies to $\mathbf{Pr}[A \cap B]/\mathbf{Pr}[B]$. This, by definition, is $\mathbf{Pr}[A \mid B]$. □

Exercise 14.21 (Practice with Bayes' rule)**.**
CAPTCHAs are tests designed to be hard for computers to solve but easy for people to solve. Suppose it is estimated that $3/4$ of all attempts to solve a CAPTCHA are from humans and the remainder are from computers. If a human has a $9/10$ chance of successfully solving a CAPTCHA and a computer has a $1/5$ chance, what is the probability that the entity attempting a CAPTCHA was a human, given that the CAPTCHA was successfully solved?

14.1.3 Independence

Definition 14.22 (Independent events)**.**

- Let A and B be two events. We say that A and B are *independent* if $\mathbf{Pr}[A \cap B] = \mathbf{Pr}[A] \cdot \mathbf{Pr}[B]$. Note that if $\mathbf{Pr}[B] \neq 0$, then this is equivalent to $\mathbf{Pr}[A \mid B] = \mathbf{Pr}[A]$. If $\mathbf{Pr}[A] \neq 0$, it is also equivalent to $\mathbf{Pr}[B \mid A] = \mathbf{Pr}[B]$.
- Let $A_1, A_2, \ldots, A_n$ be events with non-zero probabilities. We say that $A_1, \ldots, A_n$ are *independent* if for any subset $S \subseteq \{1, 2, \ldots, n\}$,

$$\mathbf{Pr}\left[\bigcap_{i \in S} A_i\right] = \prod_{i \in S} \mathbf{Pr}[A_i].$$

Note 14.23 (Defining independence through computer code)**.**
Above we have given the definition of *independent events* as presented in 100% of the textbooks on probability theory. Yet, there is something deeply unsatisfying about this definition. In many situations people want to compute a probability of the form $\mathbf{Pr}[A \cap B]$, and if possible (if they are independent), would like to use the equality $\mathbf{Pr}[A \cap B] = \mathbf{Pr}[A]\,\mathbf{Pr}[B]$ to simplify the calculation. In order to do this, they will informally argue that events A and B are independent in the intuitive sense of the word. For example, they argue that if B happens, then this doesn't affect the probability of A happening (this argument is not done by calculation, but by informal argument). Then using this, they justify using the equality $\mathbf{Pr}[A \cap B] = \mathbf{Pr}[A]\,\mathbf{Pr}[B]$ in their calculations. So really, secretly, people are not using Definition 14.22 (Independent events) but some other non-formal intuitive definition of independence, and then concluding what the formal definition says, which is $\mathbf{Pr}[A \cap B] = \mathbf{Pr}[A]\,\mathbf{Pr}[B]$.

To be a bit more explicit, recall that the approach to answering probability related questions is to go from a real-world experiment we want to analyze to a formal probability space model:

$$\text{real-world experiment} \longrightarrow \text{probability space } (\Omega, \mathbf{Pr}).$$

People often argue the independence of events A and B on the left-hand-side in order to use $\mathbf{Pr}[A \cap B] = \mathbf{Pr}[A]\,\mathbf{Pr}[B]$ on the right-hand-side. The left-hand-side, however, is not really a formal setting and may have ambiguities. And why does our intuitive notion of independence allow us to conclude $\mathbf{Pr}[A \cap$

$B] = \mathbf{Pr}[A]\,\mathbf{Pr}[B]$? In these situations, it helps to add the "computer code" step in between:

real-world experiment $\longrightarrow$ computer code $\longrightarrow$ probability space $(\Omega, \mathbf{Pr})$.

Computer code has no ambiguities and we can give a formal definition of independence using it. Suppose you have a randomized code modeling the real-world experiment, and suppose that you can divide the code into two separate parts. Suppose A is an event that depends only on the first part of the code, and B is an event that depends only on the second part of the code. If you can prove that the two parts of the code cannot affect each other, then we say that A and B are independent. When A and B are independent in this sense, then one can verify that indeed the equality $\mathbf{Pr}[A \cap B] = \mathbf{Pr}[A]\,\mathbf{Pr}[B]$ holds.

Exercise 14.24 (Pair-wise independent but not three-wise)**.**
Give an example of a probability space with 3 events A_1, A_2 and A_3 such that each pair of events A_i and A_j are independent, however A_1, A_2, A_3 together are dependent.

14.2 Probability II: Random Variables

14.2.1 Basics of random variables

Definition 14.25 (Random variable)**.**
A *random variable* is a function $\boldsymbol{X} : \Omega \to \mathbb{R}$.

Note 14.26 (Random variable intuition)**.**
Note that a random variable is just a labeling of the elements in Ω with some real numbers. One can think of this as a transformation of the original sample space into one that contains only numbers. And this is often a desirable transformation. For example, this transformation allows us to take a *weighted average* of the elements in Ω, where the weights correspond to the probabilities of the elements (if the distribution is uniform, the weighted average is just the regular average). This is called the *expectation* of the random variable and is formally defined in Definition 14.33 (Expected value of a random variable). Without this transformation into real numbers, the concept of an "expected value" would not be possible to define.

Definition 14.27 (Common events through a random variable)**.**
Let $\boldsymbol{X}$ be a random variable and $x \in \mathbb{R}$ be some real value. We use

$$\begin{aligned}
\boldsymbol{X} = x &\quad \text{to denote the event } \{\ell \in \Omega : \boldsymbol{X}(\ell) = x\},\\
\boldsymbol{X} \leq x &\quad \text{to denote the event } \{\ell \in \Omega : \boldsymbol{X}(\ell) \leq x\},\\
\boldsymbol{X} \geq x &\quad \text{to denote the event } \{\ell \in \Omega : \boldsymbol{X}(\ell) \geq x\},\\
\boldsymbol{X} < x &\quad \text{to denote the event } \{\ell \in \Omega : \boldsymbol{X}(\ell) < x\},\\
\boldsymbol{X} > x &\quad \text{to denote the event } \{\ell \in \Omega : \boldsymbol{X}(\ell) > x\}.
\end{aligned}$$

For example, $\mathbf{Pr}[\boldsymbol{X} = x]$ denotes $\mathbf{Pr}[\{\ell \in \Omega : \boldsymbol{X}(\ell) = x\}]$. More generally, for $S \subseteq \mathbb{R}$, we use

$$\boldsymbol{X} \in S \quad \text{to denote the event } \{\ell \in \Omega : \boldsymbol{X}(\ell) \in S\}.$$

Exercise 14.28 (Practice with random variables)**.**
Suppose we roll two 6-sided dice. Let $\boldsymbol{X}$ be the random variable that denotes the sum of the numbers we see. Explicitly write down the input-output pairs for the function $\boldsymbol{X}$. Calculate $\mathbf{Pr}[\boldsymbol{X} \geq 7]$.

Note 14.29 (Forgetting the original sample space)**.**
Given some probability space $(\Omega, \mathbf{Pr})$ and a random variable $\boldsymbol{X} : \Omega \to \mathbb{R}$, we often forget about the original sample space and consider the sample space to be the range of $\boldsymbol{X}$, $\text{range}(\boldsymbol{X}) = \{\boldsymbol{X}(\ell) : \ell \in \Omega\}$.

Definition 14.30 (Probability mass function (PMF))**.**
Let $\boldsymbol{X} : \Omega \to \mathbb{R}$ be a random variable. The *probability mass function* (PMF) of $\boldsymbol{X}$ is a function $p_{\boldsymbol{X}} : \mathbb{R} \to [0, 1]$ such that for any $x \in \mathbb{R}$, $p_{\boldsymbol{X}}(x) = \mathbf{Pr}[\boldsymbol{X} = x]$.

Exercise 14.31 (Facts about probability mass function)**.**
Verify the following:

- $\sum_{x \in \text{range}(\boldsymbol{X})} p_{\boldsymbol{X}}(x) = 1$,
- for $S \subseteq \mathbb{R}$, $\mathbf{Pr}[\boldsymbol{X} \in S] = \sum_{x \in S} p_{\boldsymbol{X}}(x)$.

Note 14.32 (Defining a random variable through PMF)**.**
Related to the previous remark, we sometimes "define" a random variable by just specifying its probability mass function. In particular we make no mention of the underlying sample space.

Definition 14.33 (Expected value of a random variable)**.**
Let $\boldsymbol{X}$ be a random variable. The *expected value* of $\boldsymbol{X}$, denoted $\mathbf{E}[\boldsymbol{X}]$, is defined as follows:

$$\mathbf{E}[\boldsymbol{X}] = \sum_{\ell \in \Omega} \mathbf{Pr}[\ell] \cdot \boldsymbol{X}(\ell).$$

Equivalently,

$$\mathbf{E}[\boldsymbol{X}] = \sum_{x \in \text{range}(\boldsymbol{X})} \mathbf{Pr}[\boldsymbol{X} = x] \cdot x,$$

where $\text{range}(\boldsymbol{X}) = \{\boldsymbol{X}(\ell) : \ell \in \Omega\}$.

Exercise 14.34 (Equivalence of expected value definitions)**.**
Prove that the above two expressions for $\mathbf{E}[\boldsymbol{X}]$ are equivalent.

Exercise 14.35 (Practice with expected value)**.**
Suppose we roll two 6-sided dice. Let $\boldsymbol{X}$ be the random variable that denotes the sum of the numbers we see. Calculate $\mathbf{E}[\boldsymbol{X}]$.

Proposition 14.36 (Linearity of expectation)**.**
Let $\boldsymbol{X}$ and $\boldsymbol{Y}$ be two random variables, and let $c_1, c_2 \in \mathbb{R}$ be some constants. Then $\mathbf{E}[c_1\boldsymbol{X} + c_2\boldsymbol{Y}] = c_1\,\mathbf{E}[\boldsymbol{X}] + c_2\,\mathbf{E}[\boldsymbol{Y}]$.

Proof. Define the random variable $\boldsymbol{Z}$ as $\boldsymbol{Z} = c_1\boldsymbol{X} + c_2\boldsymbol{Y}$. Then using the definition of expected value, we have

$$\begin{aligned}
\mathbf{E}[c_1\boldsymbol{X} + c_2\boldsymbol{Y}] &= \mathbf{E}[\boldsymbol{Z}] \\
&= \sum_{\ell\in\Omega} \mathbf{Pr}[\ell] \cdot \boldsymbol{Z}(\ell) \\
&= \sum_{\ell\in\Omega} \mathbf{Pr}[\ell] \cdot (c_1\boldsymbol{X}(\ell) + c_2\boldsymbol{Y}(\ell)) \\
&= \sum_{\ell\in\Omega} \mathbf{Pr}[\ell] \cdot c_1\boldsymbol{X}(\ell) + \mathbf{Pr}[\ell] \cdot c_2\boldsymbol{Y}(\ell) \\
&= \left(\sum_{\ell\in\Omega} \mathbf{Pr}[\ell] \cdot c_1\boldsymbol{X}(\ell)\right) + \left(\sum_{\ell\in\Omega} \mathbf{Pr}[\ell] \cdot c_2\boldsymbol{Y}(\ell)\right) \\
&= c_1\left(\sum_{\ell\in\Omega} \mathbf{Pr}[\ell] \cdot \boldsymbol{X}(\ell)\right) + c_2\left(\sum_{\ell\in\Omega} \mathbf{Pr}[\ell] \cdot \boldsymbol{Y}(\ell)\right) \\
&= c_1\,\mathbf{E}[\boldsymbol{X}] + c_2\,\mathbf{E}[\boldsymbol{Y}],
\end{aligned}$$

as desired. □

Corollary 14.37 (Linearity of expectation 2)**.**
Let $\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_n$ be random variables, and $c_1, c_2, \ldots, c_n \in \mathbb{R}$ be some constants. Then

$$\mathbf{E}[c_1\boldsymbol{X}_1 + c_2\boldsymbol{X}_2 + \cdots + c_n\boldsymbol{X}_n] = c_1\,\mathbf{E}[\boldsymbol{X}_1] + c_2\,\mathbf{E}[\boldsymbol{X}_2] + \cdots + c_n\,\mathbf{E}[\boldsymbol{X}_n].$$

In particular, when all the c_i's are 1, we get

$$\mathbf{E}[\boldsymbol{X}_1 + \boldsymbol{X}_2 + \cdots + \boldsymbol{X}_n] = \mathbf{E}[\boldsymbol{X}_1] + \mathbf{E}[\boldsymbol{X}_2] + \cdots + \mathbf{E}[\boldsymbol{X}_n].$$

Exercise 14.38 (Practice with linearity of expectation)**.**
Suppose we roll three 10-sided dice. Let $\boldsymbol{X}$ be the sum of the three values we see. Calculate $\mathbf{E}[\boldsymbol{X}]$.

Exercise 14.39 (Expectation of product of random variables)**.**
Let $\boldsymbol{X}$ and $\boldsymbol{Y}$ be random variables. Is it always true that $\mathbf{E}[\boldsymbol{X}\boldsymbol{Y}] = \mathbf{E}[\boldsymbol{X}]\,\mathbf{E}[\boldsymbol{Y}]$?

Definition 14.40 (Indicator random variable)**.**
Let $E \subseteq \Omega$ be some event. The *indicator random variable* with respect to E is denoted by $\boldsymbol{I}_E$ and is defined as

$$\boldsymbol{I}_E(\ell) = \begin{cases} 1 & \text{if } \ell \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Proposition 14.41 (Expectation of an indicator random variable)**.**
Let E be an event. Then $\mathbf{E}[\boldsymbol{I}_E] = \mathbf{Pr}[E]$.

Proof. By the definition of expected value,

$$\begin{aligned}\mathbf{E}[\boldsymbol{I}_E] &= \mathbf{Pr}[\boldsymbol{I}_E = 1] \cdot 1 + \mathbf{Pr}[\boldsymbol{I}_E = 0] \cdot 0 \\ &= \mathbf{Pr}[\boldsymbol{I}_E = 1] \\ &= \mathbf{Pr}[\{\ell \in \Omega : \boldsymbol{I}_E(\ell) = 1\}] \\ &= \mathbf{Pr}[\{\ell \in \Omega : \ell \in E\}] \\ &= \mathbf{Pr}[E].\end{aligned}$$

□

IMPORTANT 14.42 (Combining linearity of expectation and indicators)**.** Suppose that you are interested in computing $\mathbf{E}[\boldsymbol{X}]$ for some random variable $\boldsymbol{X}$. If you can write $\boldsymbol{X}$ as a sum of indicator random variables, i.e., if $\boldsymbol{X} = \sum_j \boldsymbol{I}_{E_j}$ where $\boldsymbol{I}_{E_j}$ are indicator random variables, then by linearity of expectation,

$$\mathbf{E}[\boldsymbol{X}] = \mathbf{E}\left[\sum_j \boldsymbol{I}_{E_j}\right] = \sum_j \mathbf{E}[\boldsymbol{I}_{E_j}].$$

Furthermore, by Proposition 14.41 (Expectation of an indicator random variable), we know $\mathbf{E}[\boldsymbol{I}_{E_j}] = \mathbf{Pr}[E_j]$. Therefore $\mathbf{E}[\boldsymbol{X}] = \sum_j \mathbf{Pr}[E_j]$. This often provides an extremely convenient way of computing $\mathbf{E}[\boldsymbol{X}]$. **This combination of indicator random variables together with linearity expectation is one of the most useful tricks in probability theory.**

Exercise 14.43 (Practice with linearity of expectation and indicators)**.**

(a) There are n balls and n bins. For each ball, you pick one of the bins uniformly at random and drop the ball in that bin. What is the expected number of balls in bin 1? What is the expected number of empty bins?

(b) Suppose you randomly color the vertices of the complete graph on n vertices one of k colors. What is the expected number of paths of length c (where we assume $c \geq 3$) such that no two adjacent vertices on the path have the same color?

Definition 14.44 (Conditional expectation)**.**
Let $\boldsymbol{X}$ be a random variable and E be an event. The *conditional expectation* of $\boldsymbol{X}$ given the event E, denoted by $\mathbf{E}[\boldsymbol{X} \mid E]$, is defined as

$$\mathbf{E}[\boldsymbol{X} \mid E] = \sum_{x \in \text{range}(\boldsymbol{X})} x \cdot \mathbf{Pr}[\boldsymbol{X} = x \mid E].$$

Proposition 14.45 (Law of total expectation)**.**
Let $\boldsymbol{X}$ be a random variable and $A_1, A_2, \ldots, A_n$ be events that partition the sample space Ω. Then

$$\mathbf{E}[\boldsymbol{X}] = \mathbf{E}[\boldsymbol{X} \mid A_1] \cdot \mathbf{Pr}[A_1] + \mathbf{E}[\boldsymbol{X} \mid A_2] \cdot \mathbf{Pr}[A_2] + \cdots + \mathbf{E}[\boldsymbol{X} \mid A_n] \cdot \mathbf{Pr}[A_n].$$

Exercise 14.46 (Proof of law of total expectation)**.**
Prove the above proposition.

Exercise 14.47 (Practice with law of total expectation)**.**
We first roll a 4-sided die. If we see the value d, we then roll a d-sided die. Let $\boldsymbol{X}$ be the sum of the two values we see. Calculate $\mathbf{E}[\boldsymbol{X}]$.

Definition 14.48 (Independent random variables)**.**
Two random variables $\boldsymbol{X}$ and $\boldsymbol{Y}$ are *independent* if for all $x, y \in \mathbb{R}$, the events $\boldsymbol{X} = x$ and $\boldsymbol{Y} = y$ are independent. The definition generalizes to more than two random variables analogous to Definition 14.22 (Independent events).

Exercise 14.49 (Expectation of product of independent random variables)**.**
Show that if $\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_n$ are independent random variables, then

$$\mathbf{E}[\boldsymbol{X}_1 \boldsymbol{X}_2 \cdots \boldsymbol{X}_n] = \mathbf{E}[\boldsymbol{X}_1] \cdot \mathbf{E}[\boldsymbol{X}_2] \cdots \mathbf{E}[\boldsymbol{X}_n].$$

14.2.2 The most fundamental inequality in probability theory

Theorem 14.50 (Markov's inequality)**.**
Let $\boldsymbol{X}$ be a non-negative random variable with non-zero expectation. Then for any $c > 0$,

$$\mathbf{Pr}[\boldsymbol{X} \geq c\,\mathbf{E}[\boldsymbol{X}]] \leq \frac{1}{c}.$$

Proof. Let $\boldsymbol{I}$ be the indicator random variable for the event $[\boldsymbol{X} \geq c\,\mathbf{E}[\boldsymbol{X}]]$. Then $\mathbf{E}[\boldsymbol{I}] = \mathbf{Pr}[\boldsymbol{X} \geq c\,\mathbf{E}[\boldsymbol{X}]]$, which corresponds to the LHS of the inequality above. We claim that

$$\boldsymbol{I} \leq \frac{\boldsymbol{X}}{c\,\mathbf{E}[\boldsymbol{X}]}.$$

First, observe that the result follows from this claim: if we take the expectations of both sides of the inequality, we get $\mathbf{E}[\boldsymbol{I}] \leq \mathbf{E}[\frac{\boldsymbol{X}}{c\,\mathbf{E}[\boldsymbol{X}]}] = \frac{\mathbf{E}[\boldsymbol{X}]}{c\,\mathbf{E}[\boldsymbol{X}]} = \frac{1}{c}$, as desired.

We now prove the above claim. The indicator random variable $\boldsymbol{I}$ can be equal to either 0 or 1. If it is 0, then the inequality holds trivially because by assumption $\mathbf{X}$ is non-negative (so $\mathbf{E}[\boldsymbol{X}]$ is also non-negative) and $c > 0$. If on the other hand $\boldsymbol{I} = 1$, then the above inequality becomes equivalent to $1 \leq \frac{\boldsymbol{X}}{c\,\mathbf{E}[\boldsymbol{X}]} \Leftrightarrow \boldsymbol{X} \geq c\,\mathbf{E}[\boldsymbol{X}]$, which must hold by the definition of $\boldsymbol{I}$.[1] □

Exercise 14.51 (Practice with Markov's inequality)**.**
During the Fall 2017 semester, the 15-251 TAs decide to strike because they are not happy with the lack of free food in grading sessions. Without the TA support, the performance of the students in the class drop dramatically. The class average on the first midterm exam is 15%. Using Markov's Inequality, give an upper bound on the fraction of the class that got an A (i.e., at least a 90%) in the exam.

[1] The notation $\boldsymbol{I} = 1$ may seem strange at first. After all, $\boldsymbol{I}$ is a function, and 1 is an integer. The meaning of this equality, however, should be clear from the context. When we are considering the case $\boldsymbol{I} = 1$, we are considering all $\ell \in \Omega$ such that $\boldsymbol{I}(\ell) = 1$. We have a similar situation for $\boldsymbol{I} = 0$. When we want to prove an inequality involving random variables, we must show that the inequality holds for all inputs $\ell \in \Omega$.

14.2.3 Three popular random variables

Definition 14.52 (Bernoulli random variable)**.**
Let $0 < p < 1$ be some parameter. If $\boldsymbol{X}$ is a random variable with probability mass function $p_{\boldsymbol{X}}(1) = p$ and $p_{\boldsymbol{X}}(0) = 1 - p$, then we say that $\boldsymbol{X}$ has a *Bernoulli distribution with parameter* p (we also say that $\boldsymbol{X}$ is a Bernoulli random variable). We write $\boldsymbol{X} \sim \text{Bernoulli}(p)$ to denote this. The parameter p is often called the *success* probability.

Note 14.53 (Expectation of Bernoulli random variable)**.**
Note that $\mathbf{E}[\boldsymbol{X}] = p$.

Definition 14.54 (Binomial random variable)**.**
Let $\boldsymbol{X} = \boldsymbol{X}_1 + \boldsymbol{X}_2 + \cdots + \boldsymbol{X}_n$, where the $\boldsymbol{X}_i$'s are independent and for all i, $\boldsymbol{X}_i \sim \text{Bernoulli}(p)$. Then we say that $\boldsymbol{X}$ has a *binomial distribution with parameters* n *and* p (we also say that $\boldsymbol{X}$ is a binomial random variable). We write $\boldsymbol{X} \sim \text{Bin}(n, p)$ to denote this.

Note 14.55 (Bernoulli is a special case of Binomial)**.**
Note that a Bernoulli random variable is a special kind of a binomial random variable where $n = 1$.

Exercise 14.56 (Expectation of a Binomial random variable)**.**
Let $\boldsymbol{X}$ be a random variable with $\boldsymbol{X} \sim \text{Bin}(n, p)$. Determine $\mathbf{E}[\boldsymbol{X}]$ (use linearity of expectation). Also determine $\boldsymbol{X}$'s probability mass function.

Exercise 14.57 (Practice with Binomial random variable)**.**
We toss a coin 5 times. What is the probability that we see at least 4 heads?

Definition 14.58 (Geometric random variable)**.**
Let $\boldsymbol{X}$ be a random variable with probability mass function $p_{\boldsymbol{X}}$ such that for $n \in \{1, 2, \ldots\}$, $p_{\boldsymbol{X}}(n) = (1-p)^{n-1}p$. Then we say that $\boldsymbol{X}$ has a *geometric distribution with parameter* p (we also say that $\boldsymbol{X}$ is a geometric random variable). We write $\boldsymbol{X} \sim \text{Geometric}(p)$ to denote this.

Exercise 14.59 (PMF of a geometric random variable)**.**
Let $\boldsymbol{X}$ be a geometric random variable. Verify that $\sum_{n=1}^{\infty} p_{\boldsymbol{X}}(n) = 1$.

Exercise 14.60 (Practice with geometric random variable)**.**
Suppose we repeatedly flip a coin until we see a heads for the first time. Determine the probability that we flip the coin n times. Determine the expected number of coin flips.

Exercise 14.61 (Expectation of a geometric random variable)**.**
Let $\boldsymbol{X}$ be a random variable with $\boldsymbol{X} \sim \text{Geometric}(p)$. Determine $\mathbf{E}[\boldsymbol{X}]$.

IMPORTANT 14.62 (Some general tips)**.**
Here are some general tips on probability calculations (this is not meant to be an exhaustive list).

- If you are trying to upper bound $\mathbf{Pr}[A]$, you can try to find B with $A \subseteq B$, and then bound $\mathbf{Pr}[B]$. Note that if an event A implies an event B, then this means $A \subseteq B$. Similarly, if you are trying to lower bound $\mathbf{Pr}[A]$, you can try to find B with $B \subseteq A$, and then bound $\mathbf{Pr}[B]$.
- If you are trying to upper bound $\mathbf{Pr}[A]$, you can try to lower bound $\mathbf{Pr}[\overline{A}]$ since $\mathbf{Pr}[A] = 1 - \mathbf{Pr}[\overline{A}]$. Similarly, if you are trying to lower bound $\mathbf{Pr}[A]$, you can try to upper bound $\mathbf{Pr}[\overline{A}]$.
- In some situations, law of total probability can be very useful in calculating (or bounding) $\mathbf{Pr}[A]$.
- If you need to calculate $\mathbf{Pr}[A_1 \cap \cdots \cap A_n]$, try the chain rule. If the events are independent, then this probability is equal to the product $\mathbf{Pr}[A_1] \cdots \mathbf{Pr}[A_n]$. Note that the event "for all $i \in \{1, \ldots, n\}$, A_i" is the same as $A_1 \cap \cdots \cap A_n$.
- If you need to upper bound $\mathbf{Pr}[A_1 \cup \cdots \cup A_n]$, you can try to use the union bound. Note that the event "there exists an $i \in \{1, \ldots, n\}$ such that A_i" is the same as $A_1 \cup \cdots \cup A_n$.
- When trying to calculate $\mathbf{E}[\boldsymbol{X}]$, try:
  (i) directly using the definition of expectation;
  (ii) writing $\boldsymbol{X}$ as a sum of indicator random variables, and then using linearity of expectation;
  (iii) using law of total expectation.

Quiz

1. True or false: If two events A and B are independent, then their complements $\overline{A}$ and $\overline{B}$ are also independent. (The complement of an event A is $\overline{A} = \Omega \backslash A$.)

2. True or false: If events A and B are disjoint, then they are independent.

3. True or false: Assume that every time a baby is born, there is $1/2$ chance that the baby is a boy. A couple has two children. At least one of the children is a boy. The probability that both children are boys is $1/2$.

4. True or false: For any non-negative random variable $\boldsymbol{X}$, $\mathbf{E}[\boldsymbol{X}^2] \leq \mathbf{E}[\boldsymbol{X}]^2$.

5. True or false: Let $\boldsymbol{X}$ be a random variable. If $\mathbf{E}[\boldsymbol{X}] = \mu$, then $\mathbf{Pr}[\boldsymbol{X} = \mu] > 0$.

6. True or false: For any event A and random variables $\boldsymbol{X}$ and $\boldsymbol{Y}$, $\mathbf{E}[\boldsymbol{X} + \boldsymbol{Y} \mid A] = \mathbf{E}[\boldsymbol{X} \mid A] + \mathbf{E}[\boldsymbol{Y} \mid A]$.

7. True or false: For any random variable $\boldsymbol{X}$, $\mathbf{E}[1/\boldsymbol{X}] = 1/\mathbf{E}[\boldsymbol{X}]$.

8. True or false: For any random variable $\boldsymbol{X}$, $\mathbf{Pr}[\boldsymbol{X} \geq \mathbf{E}[\boldsymbol{X}]] > 0$.

9. True or false: For any random variable $\boldsymbol{X}$, $\mathbf{E}[-\boldsymbol{X}^3] = -\mathbf{E}[\boldsymbol{X}^3]$.

Chapter 15

Randomized Algorithms

PREAMBLE

Chapter structure:

- Section 15.1 (Monte Carlo and Las Vegas Algorithms)
  - Definition 15.2 (Monte Carlo algorithm)
  - Definition 15.3 (Las Vegas algorithm)
- Section 15.2 (Monte Carlo Algorithm for the Minimum Cut Problem)
  - Definition 15.7 (Minimum cut problem)
  - Definition 15.8 (Multi-graph)
  - Definition 15.9 (Contraction of two vertices in a graph)
  - Theorem 15.10 (Contraction algorithm for min cut)

Chapter goals:

One of the primary applications of randomness to computer science is *randomized algorithms*. A randomized algorithm is an algorithm that has access to a randomness source like a random number generator, and a randomized algorithm is allowed to err with a very small probability of error. There are problems that we know how to solve efficiently using a randomized algorithms, however, we do not know how to solve those problems efficiently with a deterministic algorithm (i.e. an algorithm that does not make use of randomness). In fact, one of the most important open problems in computer science asks whether every efficient randomized algorithm has a deterministic counterpart solving the same problem.

In this chapter we introduce the definitions of two types of randomized algorithms: Monte Carlo algorithms and Las Vegas algorithms. The rest of the chapter is devoted to a case study: Monte Carlo algorithm for the minimum cut problem. The algorithm illustrates how a simple randomized algorithm can be powerful enough to solve a non-trivial problem. Furthermore, the analysis is quite elegant and uses some of the concepts we have learned in the last chapter. Finally, the algorithm allows us to present a very important and powerful technique: boosting the success probability of randomized algorithms by repeated trials.

15.1 Monte Carlo and Las Vegas Algorithms

Note 15.1 (Randomized algorithm)**.**
Informally, we'll say that an algorithm is *randomized* if it has access to a randomness source. In this course, we'll assume that a randomized algorithm is allowed to call RandInt(m), which returns a uniformly random element of $\{1, 2, \ldots, m\}$, and Bernoulli(p), which returns 1 with probability p and returns 0 with probability $1 - p$. We assume that both RandInt and Bernoulli take $O(1)$ time to execute. The notion of a randomized algorithm can be formally defined using *probabilistic Turing machines*, but we will not do so here.

Definition 15.2 (Monte Carlo algorithm)**.**
Let $f : \Sigma^* \to \Sigma^*$ be a computational problem. Let $0 \leq \epsilon < 1$ be some parameter and $T : \mathbb{N} \to \mathbb{N}$ be some function. Suppose A is a randomized algorithm such that

- for all $x \in \Sigma^*$, $\mathbf{Pr}[A(x) \neq f(x)] \leq \epsilon$;
- for all $x \in \Sigma^*$, $\mathbf{Pr}[\text{number of steps } A(x) \text{ takes is at most } T(|x|)] = 1$.

(Note that the probabilities are over the random choices made by A.) Then we say that A is a $T(n)$-time *Monte Carlo algorithm* that computes f with ϵ probability of error.

Definition 15.3 (Las Vegas algorithm)**.**
Let $f : \Sigma^* \to \Sigma^*$ be a computational problem. Let $T : \mathbb{N} \to \mathbb{N}$ be some function. Suppose A is a randomized algorithm such that

- for all $x \in \Sigma^*$, $\mathbf{Pr}[A(x) = f(x)] = 1$, where the probability is over the random choices made by A;
- for all $x \in \Sigma^*$, $\mathbf{E}[\text{number of steps } A(x) \text{ takes}] \leq T(|x|)$.

Then we say that A is a $T(n)$-time *Las Vegas algorithm* that computes f.

Note 15.4 (Randomized algorithms for optimization problems)**.**
One can also define the notions of Monte Carlo algorithms and Las Vegas algorithms that compute optimization problems (Definition 13.1 (Optimization problem)).

Exercise 15.5 (Las Vegas to Monte Carlo)**.**
Suppose you are given a Las Vegas algorithm A that solves $f : \Sigma^* \to \Sigma^*$ in expected time $T(n)$. Show that for any constant $\epsilon > 0$, there is a Monte Carlo algorithm that solves f in time $O(T(n))$ and error probability ϵ.

Exercise 15.6 (Monte Carlo to Las Vegas)**.**
Suppose you are given a Monte Carlo algorithm A that runs in worst-case $T_1(n)$ time and solves $f : \Sigma^* \to \Sigma^*$ with success probability at least p (i.e., for every input, the algorithm gives the correct answer with probability at least p and takes at most $T_1(n)$ steps). Suppose it is possible to check in $T_2(n)$ time whether the output produced by A is correct or not. Show how to convert A into a Las Vegas algorithm that runs in expected time $O((T_1(n) + T_2(n))/p)$.

15.2 Monte Carlo Algorithm for the Minimum Cut Problem

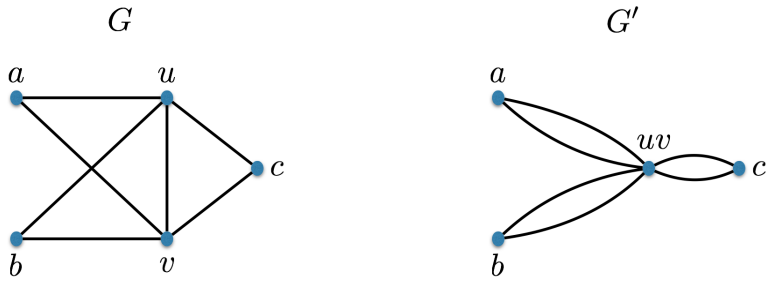
Definition 15.7 (Minimum cut problem)**.**
In the minimum cut problem, the input is a connected undirected graph G, and the output is a 2-coloring of the vertices such that the number of cut edges is minimized. (See Definition 13.10 (Max-cut problem) for the definition of a *cut edge*.) Equivalently, we want to output a non-empty subset $S \subsetneq V$ such that the number of edges between S and $V \backslash S$ is minimized. Such a set S is called a *cut* and the size of the cut is the number of edges between S and $V \backslash S$ (note that the size of the cut is not the number of vertices). We denote this problem by MIN-CUT.

Definition 15.8 (Multi-graph)**.**
A *multi-graph* $G = (V, E)$ is an undirected graph in which E is allowed to be a multi-set. In other words, a multi-graph can have multiple edges between two vertices.[1]

Definition 15.9 (Contraction of two vertices in a graph)**.**
Let $G = (V, E)$ be a multi-graph and let $u, v \in V$ be two vertices in the graph. *Contraction* of u and v produces a new multi-graph $G' = (V', E')$. Informally, in G', we collapse/contract the vertices u and v into one vertex and preserve the edges between these two vertices and the other vertices in the graph. Formally, we remove the vertices u and v, and create a new vertex called uv, i.e. $V' = V \backslash \{u, v\} \cup \{uv\}$. The multi-set of edges E' is defined as follows:

- for each $\{u, w\} \in E$ with $w \neq v$, we add $\{uv, w\}$ to E';
- for each $\{v, w\} \in E$ with $w \neq u$, we add $\{uv, w\}$ to E';
- for each $\{w, w'\} \in E$ with $w, w' \notin \{u, v\}$, we add $\{w, w'\}$ to E'.

Below is an example:



Theorem 15.10 (Contraction algorithm for min cut)**.**
There is a polynomial-time Monte-Carlo algorithm that solves the MIN-CUT *problem with error probability at most* $1/e^n$*, where* n *is the number of vertices in the input graph.*

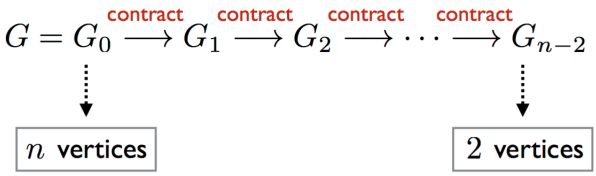
Proof. The algorithm has two phases. The description of the *first phase* is as follows.

[1] Note that this definition does not allow for self-loops.

G: connected undirected graph.
$A(\langle G \rangle)$:

1 Repeat until two vertices remain:
2 Select an edge $\{u, v\}$ uniformly at random.
3 Contract u and v to obtain a new graph.
4 Two vertices remain, which corresponds to a partition of V into V_1 and V_2. Output V_1.

Let G_i denote the graph we have after i iterations of the algorithm. So $G_0 = G$, G_1 is the graph after we contract one of the edges, and so on. Note that the algorithm has $n-2$ iterations because in each iteration the number of vertices goes down by exactly one and we stop when 2 vertices remain.



This makes it clear that the algorithm runs in polynomial time: we have $n-2$ iterations, and in each iteration we can contract an edge, which can be done in polynomial time. Our goal now is to show that the success probability of the first phase, i.e., the probability that the above algorithm outputs a minimum cut, is at least

$$\frac{2}{n(n-1)} \geq \frac{1}{n^2}.$$

In the second phase, we'll boost the success probability to the desired $1 - 1/e^n$. We make two observations.

Observation 1: For any i, a cut in G_i of size k corresponds to a cut in $G = G_0$ of size k. (We leave the proof of this as an exercise.)

Observation 2: For any i and any vertex v in G_i, the size of the minimum cut (in G) is at most $\deg_{G_i}(v)$. This is because a single vertex v forms a cut by itself (i.e. $S = \{v\}$ is a cut), and the size of this cut is $\deg(v)$. By Observation 1, the original graph G has a corresponding cut with the same size. Since the minimum cut has the minimum possible size among all cuts in G, its size cannot be larger than $\deg(v)$.

We are now ready to analyze the success probability of the first phase. Let $F \subseteq E$ correspond to an optimum solution, i.e., a minimum size set of cut edges. We will show

$$\mathbf{Pr}[\text{algorithm finds } F] \geq \frac{2}{n(n-1)}.$$

Observe that if the algorithm picks an edge in F to contract, its output cannot correspond to F. If the algorithm never contracts an edge in F, then its output corresponds to F.[2] In other words, the algorithm's output corresponds to F if and only if it never contracts an edge of F. Let E_i be the event that at iteration i of the algorithm, an edge in F is contracted. As noted above, there are $n-2$ iterations in total. Therefore

$$\mathbf{Pr}[\text{algorithm finds } F] = \mathbf{Pr}[\overline{E}_1 \cap \overline{E}_2 \cap \ldots \cap \overline{E}_{n-2}].$$

[2] We are not giving a detailed argument for this, but please do verify this for yourself.

Using Proposition 14.15 (Chain rule), we have

$$\begin{aligned}&\mathbf{Pr}[\overline{E}_1 \cap \overline{E}_2 \cap \ldots \cap \overline{E}_{n-2}] = \\ &\mathbf{Pr}[\overline{E}_1] \cdot \mathbf{Pr}[\overline{E}_2 \mid \overline{E}_1] \cdot \mathbf{Pr}[\overline{E}_3 \mid \overline{E}_1 \cap \overline{E}_2] \cdots \mathbf{Pr}[\overline{E}_{n-2} \mid \overline{E}_1 \cap \overline{E}_2 \cap \ldots \cap \overline{E}_{n-3}].\end{aligned} \tag{15.1}$$

To lower bound the success probability of the algorithm, we'll find a lower bound for each term of the RHS of the above equation. We start with $\mathbf{Pr}[\overline{E}_1]$. It is easy to see that $\mathbf{Pr}[E_1] = |F|/m$. However, it will be more convenient to have a bound on $\mathbf{Pr}[E_1]$ in terms of $|F|$ and n rather than m. By Observation 2 above, we know

$$\forall v \in V, \quad |F| \leq \deg(v).$$

Using this, we have

$$2m = \sum_{v \in V} \deg(v) \geq |F| \cdot n, \tag{15.2}$$

or equivalently, $|F| \leq 2m/n$. Therefore,

$$\mathbf{Pr}[E_1] = \frac{|F|}{m} \leq \frac{2}{n},$$

or equivalently, $\mathbf{Pr}[\overline{E}_1] \geq 1 - 2/n$. At this point, going back to Equality ((15.1)) above, we can write

$$\begin{aligned}&\mathbf{Pr}[\text{algorithm finds } F] \geq \\ &\left(1 - \frac{2}{n}\right) \cdot \mathbf{Pr}[\overline{E}_2 \mid \overline{E}_1] \cdot \mathbf{Pr}[\overline{E}_3 \mid \overline{E}_1 \cap \overline{E}_2] \cdots \mathbf{Pr}[\overline{E}_{n-2} \mid \overline{E}_1 \cap \overline{E}_2 \cap \ldots \cap \overline{E}_{n-3}].\end{aligned}$$

We move onto the second term $\mathbf{Pr}[\overline{E}_2 \mid \overline{E}_1]$. Let ℓ_1 be the number of edges remaining after the first iteration of the algorithm. Then

$$\mathbf{Pr}[\overline{E}_2 \mid \overline{E}_1] = 1 - \mathbf{Pr}[E_2 \mid \overline{E}_1] = 1 - \frac{|F|}{\ell_1}.$$

As before, using Observation 2, for any v in G_1, $|F| \leq \deg_{G_1}(v)$. Therefore, the analog of Inequality ((15.2)) above for the graph G_1 yields $2\ell_1 \geq |F|(n-1)$. Using this inequality,

$$\mathbf{Pr}[\overline{E}_2 \mid \overline{E}_1] = 1 - \frac{|F|}{\ell_1} \geq 1 - \frac{2|F|}{|F|(n-1)} = 1 - \frac{2}{n-1}.$$

Thus

$$\begin{aligned}&\mathbf{Pr}[\text{algorithm finds } F] \geq \\ &\left(1 - \frac{2}{n}\right) \cdot \left(1 - \frac{2}{n-1}\right) \cdot \mathbf{Pr}[\overline{E}_3 \mid \overline{E}_1 \cap \overline{E}_2] \cdots \mathbf{Pr}[\overline{E}_{n-2} \mid \overline{E}_1 \cap \overline{E}_2 \cap \ldots \cap \overline{E}_{n-3}].\end{aligned}$$

Applying the same reasoning for the rest of the terms in the product above, we get

$$\begin{aligned}&\mathbf{Pr}[\text{algorithm finds } F] \geq \\ &\left(1 - \frac{2}{n}\right) \cdot \left(1 - \frac{2}{n-1}\right) \cdot \left(1 - \frac{2}{n-2}\right) \cdots \left(1 - \frac{2}{n-(n-3)}\right) = \\ &\left(\frac{n-2}{n}\right) \cdot \left(\frac{n-3}{n-1}\right) \cdot \left(\frac{n-4}{n-2}\right) \cdots \left(\frac{2}{4}\right) \cdot \left(\frac{1}{3}\right).\end{aligned}$$

After cancellations between the numerators and denominators of the fractions, the first two denominators and the last two numerators survive, and the above simplifies to $2/n(n-1)$. So we have reached our goal for the first phase and have shown that

$$\mathbf{Pr}[\text{algorithm finds } F] \geq \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}} \geq \frac{1}{n^2}.$$

This implies

$$\mathbf{Pr}[\text{algorithm finds a min-cut}] \geq \frac{1}{n^2}.$$

In the second phase of the algorithm, we boost the success probability by repeating the first phase t times using completely new and independent random choices. Among the t cuts we find, we return the minimum-sized one. As t grows, the success probability increases. Our analysis will show that $t = n^3$ is sufficient for the bound we want. Let A_i be the event that our algorithm does *not* find a min-cut at repetition i. Note that the A_i's are independent since our algorithm uses fresh random bits for each repetition. Also, each A_i has the same probability, i.e. $\mathbf{Pr}[A_i] = \mathbf{Pr}[A_j]$ for all i and j. Therefore

$$\begin{aligned}\mathbf{Pr}[\text{our algorithm fails to find a min-cut}] &= \mathbf{Pr}[A_1 \cap \cdots \cap A_t] \\ &= \mathbf{Pr}[A_1] \cdots \mathbf{Pr}[A_t] \\ &= \mathbf{Pr}[A_1]^t.\end{aligned}$$

From the analysis of the first phase, we know that

$$\mathbf{Pr}[A_1] \leq 1 - \frac{1}{n^2}.$$

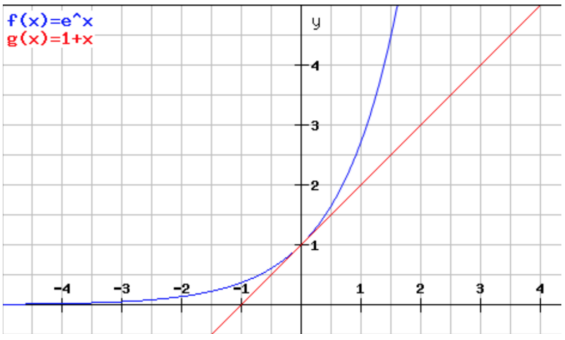
So

$$\mathbf{Pr}[\text{our algorithm fails to find a min-cut}] \leq \left(1 - \frac{1}{n^2}\right)^t.$$

To upper bound this, we'll use an extremely useful inequality:

$$\forall x \in \mathbb{R}, \quad 1 + x \leq e^x.$$

We will not prove this inequality, but we provide a plot of the two functions below.



Notice that the inequality is close to being tight for values of x close to 0. Letting $x = -1/n^2$, we see that

$$\mathbf{Pr}[\text{our algorithm fails to find a min-cut}] \leq (1+x)^t \leq e^{xt} = e^{-t/n^2}.$$

For $t = n^3$, this probability is upper bounded by $1/e^n$, as desired. □

Exercise 15.11 (Boosting for one-sided error)**.**
This question asks you to boost the success probability of a Monte Carlo algorithm computing a decision problem with *one-sided error*.

Let $f : \Sigma^* \to \{0,1\}$ be a decision problem, and let A be a Monte Carlo algorithm for f such that if x is a YES instance, then A always gives the correct answer, and if x is a NO instance, then A gives the correct answer with probability at least $1/2$. Suppose A runs in worst-case $O(T(n))$ time. Design a new Monte Carlo algorithm A' for f that runs in $O(nT(n))$ time and has error probability at most $1/2^n$.

Exercise 15.12 (Boosting for two-sided error)**.**
This question asks you to boost the success probability of a Monte Carlo algorithm computing a decision problem with *two-sided error*.

Let $f : \Sigma^* \to \{0,1\}$ be a decision problem, and let A be a Monte Carlo algorithm for f with error probability $1/4$, i.e., for all $x \in \Sigma^*$, $\mathbf{Pr}[A(x) \neq f(x)] \leq 1/4$. We want to boost the success probability to $1 - 1/2^n$, and our strategy will be as follows. Given x, run $A(x)$ $6n$ times (where $n = |x|$), and output the more common output bit among the $6n$ output bits (breaking ties arbitrarily). Show that the probability of outputting the wrong answer is at most $1/2^n$.

Exercise 15.13 (Maximum number of minimum cuts)**.**
Using the analysis of the randomized minimum cut algorithm, show that a graph can have at most $n(n-1)/2$ distinct minimum cuts.

Exercise 15.14 (Contracting two random vertices)**.**
Suppose we modify the min-cut algorithm seen in class so that rather than picking an edge uniformly at random, we pick 2 vertices uniformly at random and contract them into a single vertex. True or False: The success probability of the algorithm (excluding the part that boosts the success probability) is $1/n^k$ for some constant k, where n is the number of vertices. Justify your answer.

Hints to Selected Exercises

Exercise 15.5 (Las Vegas to Monte Carlo):
Run $A(x)$ for a certain number of steps (which you should choose carefully), and return its answer if it terminates within that number of steps. In your analysis, use Markov's inequality.

Exercise 15.6 (Monte Carlo to Las Vegas):
Repeatedly run $A(x)$ until it gives you a correct answer. You will need to know the expectation of a Geometric random variable.

Exercise 15.11 (Boosting for one-sided error):
Run $A(x)$ $|x|$ many times.

Exercise 15.12 (Boosting for two-sided error):
Write down the expression for the error probability (which will be a big sum involving binomial coefficients). Then feel free to be crude with how you upper bound this expression, i.e., don't try to be tight with your inequalities.

Exercise 15.14 (Contracting two random vertices):
False. Consider two disjoint cliques joined by a single edge.