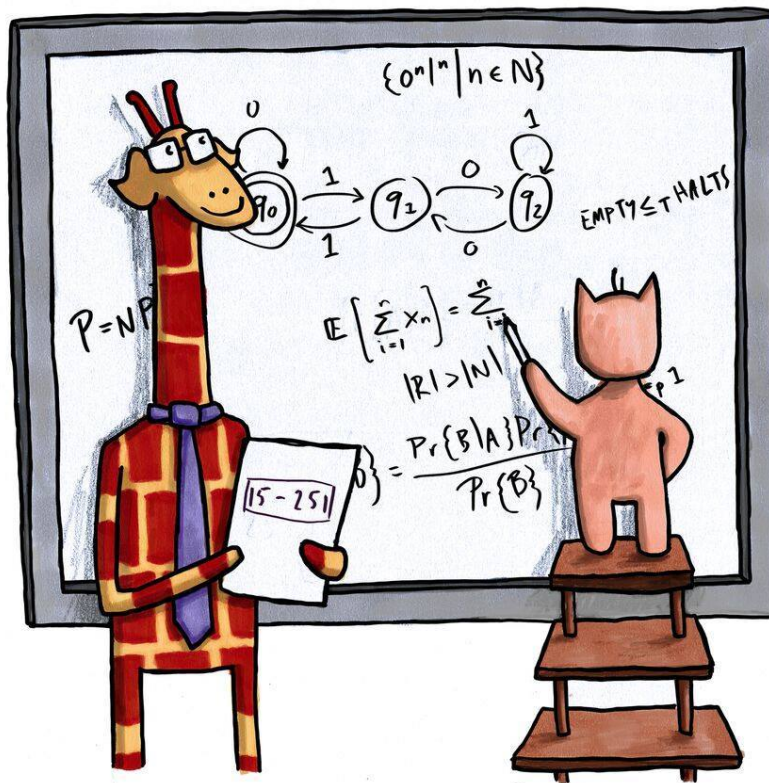# CMU 15-251,   Fall 2017
Great Ideas in Theoretical Computer Science

Course Notes: Condensed 2

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

November 17, 2017

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 Ariel Procaccia for the Fall 2017 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

Contents

v

Chapter 1

Strings and Encodings

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*.

Definition 1.2 (String/word, empty string).
Given an alphabet Σ, a *string* (or *word*) over Σ is a (possibly infinite) sequence of symbols, written as $a_1 a_2 a_3 \ldots$, where each $a_i \in \Sigma$. The string with no symbols is called the *empty string* and is denoted by ϵ.

Definition 1.3 (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.

Definition 1.4 (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,

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

Definition 1.5 (Reversal of a string).
For a string $w = a_1 a_2 \ldots a_n$, the *reversal* of w, denoted w^R, is the string $w^R = a_n a_{n-1} \ldots a_1$.

Definition 1.6 (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.

Definition 1.7 (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.

Definition 1.8 (Substring).
We say that a string u is a *substring* of string w if $w = xuy$ for some strings x and y.

1.2 Languages

Definition 1.9 (Language).
Any (possibly infinite) subset $L \subseteq \Sigma^*$ is called a *language* over the alphabet Σ.

Definition 1.10 (Reversal of a language).
Given a language $L \subseteq \Sigma^*$, we define its *reversal*, denoted L^R, as the language

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

Definition 1.11 (Concatenation of languages).
Given two languages $L_1, L_2 \subseteq \Sigma^*$, we define their *concatenation*, denoted $L_1 L_2$ or $L_1 \cdot L_2$, as the language

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

Definition 1.12 (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,

$$L^n = \{u_1 u_2 \cdots u_n \in \Sigma^* : u_i \in L \text{ for all } i \in \{1, 2, \ldots, n\}\}.$$

Definition 1.13 (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,

$$L^* = \{u_1 u_2 \cdots u_n \in \Sigma^* : n \in \mathbb{N}, u_i \in L \text{ for all } i \in \{1, 2, \ldots, n\}\}.$$

1.3 Encodings

Definition 1.14 (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 $\text{Enc} : A \rightarrow \Sigma^*$. 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]

---

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

1.4 Computational Problems and Decision Problems

Definition 1.15 (Computational problem)**.**
Let Σ be an alphabet. Any function $f : \Sigma^* \to \Sigma^*$ is called a *computational problem* over the alphabet Σ.

Definition 1.16 (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 $\{\text{Reject}, \text{Accept}\}$.

Chapter 2

Deterministic Finite Automata

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*).

Definition 2.2 (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.

Definition 2.3 (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.

Definition 2.4 (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 $\delta : Q \times \Sigma^* \to Q$.

Definition 2.5 (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]

Definition 2.6 (Regular language).
A language $L \subseteq \Sigma^*$ is called *regular* if there is a deterministic finite automaton M such that $L = L(M)$.

2.2   Irregular Languages

Theorem 2.7 ($0^n 1^n$ 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 $L = \{0^n 1^n : n \in \mathbb{N}\}$ 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. $\square$

Theorem 2.8 (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

---

[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.

[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.

a^{2^j} end up in the same state in M. Therefore $\mathsf{a}^{2^i}w$ and $\mathsf{a}^{2^j}w$, *for any* string $w \in \{\mathsf{a}\}^*$, 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 $\mathsf{a}^{2^i}w$ ends up in an accepting state but $\mathsf{a}^{2^j}w$ ends up in a rejecting state (i.e. they end up in different states).

Consider the string $w = \mathsf{a}^{2^i}$. Then $\mathsf{a}^{2^i}w = \mathsf{a}^{2^i}\mathsf{a}^{2^i} = \mathsf{a}^{2^{i+1}}$, and therefore must end up in an accepting state. On the other hand, $\mathsf{a}^{2^j}w = \mathsf{a}^{2^j}\mathsf{a}^{2^i} = \mathsf{a}^{2^j + 2^i}$. 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 $\mathsf{a}^{2^j + 2^i}$ leads to a reject state in M as claimed. $\quad\square$

2.3 Closure Properties of Regular Languages

Theorem 2.9 (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. $\quad\square$

Corollary 2.10 (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 2.9 (Regular languages are closed under union)) and closed under complementation (Exercise (Are regular languages closed under complementation?)). The result then follows since $A \cap B = \overline{\overline{A} \cup \overline{B}}$. $\qquad\square$

Theorem 2.11 (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 $L_1 L_2$ 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. To show $L_1 L_2$ is regular, we'll construct a DFA $M'' = (Q'', \Sigma, \delta'', q_0'', F'')$ that recognizes $L_1 L_2$. 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 $w \in L_1 L_2$ 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 $w = w_1 w_2 \ldots w_n \in \Sigma^*$, a *thread* with respect to w is a sequence of states

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

where $r_0, r_1, \ldots, r_i$ is an accepting computation path of M with respect to $w_1 w_2 \ldots w_i$, [3] and $q_0', s_{i+1}, s_{i+2}, \ldots, s_n$ is a computation path (not necessarily accepting) of M' with respect to $w_{i+1} w_{i+2} \ldots w_n$. A thread like this corresponds to simulating M on $w_1 w_2 \ldots w_i$ (at which point we require that an accepting state of M is reached), and then simulating M' on $w_{i+1} w_{i+2} \ldots w_n$. For each way of writing w as uv where $u \in L_1$, there is a corresponding thread for it. Note that $w \in L_1 L_2$ 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 $n = 10$):

$$s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}$$
$$s_5', s_6', s_7', s_8', s_9', s_{10}'$$

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).

---

[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 2.2 (Computation path for a DFA)).

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,

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

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'.)

---

[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 $\mathcal{P}(Q)$.

This completes the definition of M''.

To see that M'' indeed recognizes the language $L_1 L_2$, i.e. $L(M'') = L_1 L_2$, 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 $w \in L_1 L_2$ if and only if there is a thread with respect to w that ends in an accepting state of M'. $\square$

Chapter 3

Turing Machines

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 <u>not</u> 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 $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\text{L}, \text{R}\}$
  (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*).

Definition 3.2 (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*) $\alpha_0, \alpha_1, \ldots, \alpha_T$ 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 $t = 0, 1, 2, \ldots, T - 1$;

---

[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.

(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*.

Definition 3.3 (Decider Turing machine).
A Turing machine is called a *decider* if it halts on all inputs.

Definition 3.4 (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 3.5 (Decidable language).
A language L is called *decidable* (or *computable*) if $L = L(M)$ for some decider Turing machine M.

Definition 3.6 (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.

3.2 Decidable Languages

Definition 3.7 (Languages related to encodings of DFAs).
Fix some alphabet Σ. We define the following languages:

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

Theorem 3.8 (ACCEPTS$_{\text{DFA}}$ and SELF-ACCEPTS$_{\text{DFA}}$ are decidable).
The languages ACCEPTS$_{\text{DFA}}$ *and* SELF-ACCEPTS$_{\text{DFA}}$ *are decidable.*

Proof. Our goal is to show that ACCEPTS$_{\text{DFA}}$ and 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 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 ACCEPTS$_{\text{DFA}}$.

For 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 SELF-ACCEPTS$_{\text{DFA}}$. ☐

Theorem 3.9 (EMPTY$_{\text{DFA}}$ is decidable)**.**
The language EMPTY$_{\text{DFA}}$ *is decidable.*

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

A decider for 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,[3] then 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 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 3.10 (EQ$_{\text{DFA}}$ is decidable)**.**
The language EQ$_{\text{DFA}}$ *is decidable.*

Proof. Our goal is to show that EQ$_{\text{DFA}}$ is decidable. We will do so by constructing a decider for EQ$_{\text{DFA}}$.

Our argument is going to use the fact that EMPTY$_{\text{DFA}}$ is decidable (Theorem 3.9 (EMPTY$_{\text{DFA}}$ is decidable)). In particular, the decider we present for EQ$_{\text{DFA}}$ will use the decider for EMPTY$_{\text{DFA}}$ as a subroutine. Let M denote a decider TM for EMPTY$_{\text{DFA}}$.

---

[3]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.

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*[4]

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

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 (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;[5]

- 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 EQ_{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. □

---

[4]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 $(A \cap \overline{B}) \cup (\overline{A} \cap B)$.

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

Chapter 4

Countable and Uncountable Sets

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 $B \hookrightarrow A$.

Definition 4.3 (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]

Definition 4.4 (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. $|A| > |\mathbb{N}|$.

Theorem 4.5 (Characterization of countably infinite sets).
A set A is countably infinite if and only if $|A| = |\mathbb{N}|$.

---

[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 $|A| < |B|$.

4.2 Countable Sets

Proposition 4.6 ($\mathbb{Z} \times \mathbb{Z}$ 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. $\square$

Proposition 4.7 ($\mathbb{Q}$ is countable).
The set of rational numbers $\mathbb{Q}$ is countable.

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. $|\mathbb{Q}| \leq |\mathbb{N}|$. $\square$

Proposition 4.8 (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.

---

[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.

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. $\qquad\square$

Proposition 4.9 (The set of Turing machines is countable)**.**
The set of all Turing machines $\{M : M$ is a TM$\}$ is countable.

Proof. Let $T = \{M : M$ 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 4.8 (Σ^* is countable)), i.e., $|\Sigma^*| \leq |\mathbb{N}|$, the result follows. $\qquad\square$

Proposition 4.10 (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. $\qquad\square$

4.3 Uncountable Sets

Theorem 4.11 (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. $\qquad\square$

Corollary 4.12 ($\mathcal{P}(\mathbb{N})$ is uncountable)**.**
The set $\mathcal{P}(\mathbb{N})$ is uncountable.

Corollary 4.13 (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 4.8 (Σ^* is countable)). So by Theorem 4.5 (Characterization of countably infinite sets), we know $|\Sigma^*| = |\mathbb{N}|$. Theorem 4.11 (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. $\qquad\square$

Definition 4.14 (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 4.15 ($\{0,1\}^\infty$ 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 4.12 ($\mathcal{P}(\mathbb{N})$ 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.

$f(1)$	0	0	0	0	0	$\cdots$
$f(2)$	1	1	1	1	1	$\cdots$
$f(3)$	0	1	0	1	0	$\cdots$
$f(4)$	1	0	1	0	1	$\cdots$
$f(5)$	0	0	1	1	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. $\qquad\square$

Chapter 5

Undecidable Languages

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 <u>not</u> 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 4.13 (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 4.9 (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. $\square$

5.2 Examples of Undecidable Languages

Definition 5.2 (Halting problem).
The *halting problem* is defined as the decision problem corresponding to the language HALTS $= \{\langle M, x \rangle : M$ is a TM which halts on input $x\}$.

Theorem 5.3 (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. $\square$

Definition 5.4 (Languages related to encodings of TMs).
We define the following languages:

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

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

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

Theorem 5.5 (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 5.3 (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 5.6 (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 5.5 (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_ACCEPTS(⟨M, x⟩):
 1  Construct the following string, which we call ⟨M'⟩.
 2  "M'(y):
 3      Run M(x).
 4      If it accepts, accept.
 5      If it rejects, reject."
 6  Run M_EMPTY(⟨M'⟩).
 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. □

Theorem 5.7 (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 5.6 (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_EMPTY(⟨M⟩):
 1  Construct the string ⟨M'⟩ where M' is a TM that rejects
    every input.
 2  Run M_EQ(⟨M, M'⟩).
 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 $L(M) = \emptyset$. □

5.3   Undecidability Proofs by Reductions

Theorem 5.8 (HALTS $\leq$ EMPTY)**.**
HALTS $\leq$ EMPTY.

Proof. (This can be considered as an alternative proof of Theorem 5.6 (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 5.9 (EMPTY ≤ HALTS)**.**
EMPTY ≤ 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.
$M_{\text{EMPTY}}(\langle M \rangle)$:
1 Construct the following string, which we call $\langle M' \rangle$.
2 "$M'(x)$:
3   For $t = 1, 2, 3, \ldots$:
4   For each y with $|y| \leq t$:
5   Simulate $M(y)$ for at most t steps.
6   If it accepts, accept."
7 Run $M_{\text{HALTS}}(\langle M', \epsilon \rangle)$.
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. □

Chapter 6

Time Complexity

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* $g(n)$.

Definition 6.2 (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* $g(n)$.

Definition 6.3 (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 6.4 (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 $\log_b n = \Theta(\log_2 n)$. $\square$

6.2 Worst-Case Running Time of Algorithms

Definition 6.5 (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.

---

[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.

Definition 6.6 (Names for common growth rates).

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

Proposition 6.7 (Intrinsic complexity of $\{0^k 1^k : k \in \mathbb{N}\}$).
The intrinsic complexity of $L = \{0^k 1^k : k \in \mathbb{N}\}$ is $\Theta(n)$.

Proof. We want to show that the intrinsic complexity of $L = \{0^k 1^k : k \in \mathbb{N}\}$ 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 $x = 0^k 1^k$ (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'. $\qquad\square$

6.3   Complexity of Algorithms with Integer Inputs

Definition 6.8 (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.

Chapter 7

The Science of Cutting Cake

7.1 The Problem and the Model

Definition 7.1 (Cake cutting problem).
We refer to the interval $[0, 1] \subset \mathbb{R}$ as the *cake*, and the set $N = \{1, 2, \ldots, n\}$ as the set of *players*. A *piece of cake* is any set $X \subseteq [0, 1]$ which is a finite union of disjoint intervals. Let $\mathcal{X}$ denote the set of all possible pieces of cake. Each player $i \in N$ has a valuation function $V_i : \mathcal{X} \to \mathbb{R}$ that satisfies the following 4 properties.

- **Normalized:** $V_i([0, 1]) = 1$.

- **Non-negative:** For any $X \in \mathcal{X}$, $V_i(X) \geq 0$.

- **Additive:** For $X, Y \in \mathcal{X}$ with $X \cap Y = \emptyset$, $V_i(X \cup Y) = V_i(X) + V_i(Y)$.

- **Divisible:** For every interval $I \subseteq [0, 1]$ and $0 \leq \lambda \leq 1$, there exists a subinterval $I' \subseteq I$ such that $V_i(I') = \lambda V_i(I)$.

The goal is to find an *allocation* $A_1, A_2, \ldots, A_n$, where for each i, A_i is a piece of cake allocated to player i. The allocation is assumed to be a partition of the cake $[0, 1]$, i.e., the A_i's are disjoint and their union is $[0, 1]$. There are 2 properties desired about the allocation:

- **Proportionality:** For all $i \in N$, $V_i(A_i) \geq 1/n$.

- **Envy-Freeness:** For all $i, j \in N$, $V_i(A_i) \geq V_i(A_j)$.

Proposition 7.2 (An observation about the V_i's).
Let $A_1, \ldots, A_n$ be an allocation in the cake cutting problem. Then for each player i, we have $\sum_{j \in N} V_i(A_j) = 1$.

Proof. Given any player i, our goal is to show that $\sum_{j \in N} V_i(A_j) = 1$. This will follow from the additivity and normality properties of the valuation functions.
  First, recall that the A_i's form a partition of $[0, 1]$. So

$$A_1 \cup A_2 \cup \cdots \cup A_n = [0, 1],$$

and the A_i's are pairwise disjoint. Now take an arbitrary player i. By the normality property, we know $V_i([0, 1]) = 1$. Combining this with the additivity property, we have

$$1 = V_i([0, 1]) = V_i(A_1 \cup A_2 \cup \cdots \cup A_n) = V_i(A_1) + V_i(A_2) + \cdots + V_i(A_n),$$

i.e., $\sum_{j \in N} V_i(A_j) = 1$. □

Proposition 7.3 (Envy-freeness implies proportionality).
If an allocation is envy-free, then it is proportional.

Proof. Let's assume we have an allocation $A_1, \ldots, A_n$ that is envy-free. We want to show that it must also be proportional. Take an arbitrary player i. By the previous proposition, we have $\sum_{j \in N} V_i(A_j) = 1$. Therefore, there must be $k \in N$ such that $V_i(A_k) \geq 1/n$ (otherwise the sum could not be 1). The envy-freeness property implies that $V_i(A_i) \geq V_i(A_k)$, and so $V_i(A_i) \geq 1/n$. This establishes that the allocation must be proportional. □

Definition 7.4 (The Robertson-Webb model).
We use the Robertson-Webb model to express cake cutting algorithms and measure their running times. In this model, the input size is considered to be the number of players n. There is a referee who is allowed to make two types of queries to the players:

- $\text{Eval}_i(x, y)$, which returns $V_i([x, y])$,

- $\text{Cut}_i(x, \alpha)$, which returns y such that $V_i([x, y]) = \alpha$.
  (If no such y exists, it returns "None".)

The referee follows an *algorithm/strategy* and chooses the queries that she wants to make. What the referee chooses as a query depends only on the results of the queries she has made before. At the end, she decides on an allocation $A_1, A_2, \ldots, A_n$, and the allocation depends only on the outcomes of the queries. The *time complexity* of the algorithm, $T(n)$, is the number of queries she makes for n players and the worst possible V_i's. So

$$T(n) = \max_{(V_1, \ldots, V_n)} \text{ number of queries when the valuations are } (V_1, \ldots, V_n).$$

7.2 Cake Cutting Algorithms in the Robertson-Webb Model

Proposition 7.5 (Cut and Choose algorithm for 2 players).
When $n = 2$, there is always an allocation that is proportional and envy-free.

Proof. Given $n = 2$ players, we will describe a way to allocate the cake so that it is proportional and envy-free. We first describe how to find the allocation. We then argue why that allocation is envy-free and proportional.

We can describe how the allocation is found in the following way. The first player marks a point y in the cake so that $V_1([0, y]) = V_1([y, 1]) = 1/2$ (this can be done because of the divisibility property). Then player 2 chooses the piece (among $[0, y]$ and $[y, 1]$) that he values more. The remaining piece is what player 1 gets. In the Robertson-Webb model, this algorithm corresponds to the following. The referee first queries $\text{Cut}_1(0, 1/2)$. Say this returns the value y. Then the referee queries $\text{Eval}_2(0, y)$ and $\text{Eval}_2(y, 1)$.[1] Whichever gives the larger value, referee assigns that piece to player 2. The remaining piece is assigned to player 1.[2]

The allocation is envy-free: From player 1's perspective, both players get a piece of the cake of value $1/2$. Therefore $V_1(A_1) \geq V_1(A_2)$ is satisfied. From player 2's perspective, since he gets to choose the piece of larger value to him, $V_2(A_2) \geq V_2(A_1)$ is satisfied. (Also note that we must have $V_2(A_2) > 1/2$.)

The allocation is proportional: It is not hard to see that the algorithm is proportional since each player gets a piece of value at least $1/2$. $\qquad\square$

Theorem 7.6 (Dubins-Spanier Algorithm).
There is an algorithm of time complexity $\Theta(n^2)$ that produces an allocation for the cake cutting problem that satisfies the proportionality property.

---

[1] In fact, just querying $\text{Eval}_2(0, y)$ is enough.

[2] It is common to describe a cake cutting algorithm in terms of what players do to agree on an allocation. In the Robertson-Webb model we have described, this would correspond to a referee applying Eval and Cut queries to determine the allocation. The two points of views are equivalent as long as the actions of the players can be described using Eval and Cut queries.

Proof. Our goal is to describe a cake cutting algorithm with $\Theta(n^2)$ complexity that produces a proportional allocation. We first describe the algorithm. We then argue that it indeed produces a proportional allocation. Finally, we show that its complexity is $\Theta(n^2)$.

The algorithm is as follows. The referee first makes n queries: $\text{Cut}_i(0, 1/n)$ for all i. She computes the minimum among these values, which we'll denote by y. Let's assume j is the player that corresponds to the minimum value. Then the referee assigns $A_j = [0, y]$. So player j gets a piece that she values at $1/n$. After this, we remove player j, and repeat the process on the remaining cake. So in the next stage, the referee makes $n - 1$ queries, $\text{Cut}_i(y, 1/n)$ for $i \neq j$, figures out the player corresponding to the minimum value, and assigns her the corresponding piece of the cake, which she values at $1/n$. This repeats until there is one player left. The last player gets the piece that is left.[3]

We have to show that the algorithm's time complexity is $\Theta(n^2)$ and that it produces a proportional allocation. First we show that the allocation is proportional. Notice that if the queries that the referee makes never return "None", then at each iteration, until one player is left, the player j who is removed is assigned A_j such that $V_j(A_j) = 1/n$. So it suffices to argue that:

(i) the queries never return "None",

(ii) the last player, call it ℓ, gets A_ℓ such that $V_\ell(A_\ell) \geq 1/n$.

To show (i), assume we have just completed iteration k, where $k \in \{1, 2, \ldots, n-1\}$. Let j be an arbitrary player who has not been removed yet. The important observation is that all the pieces that have been removed so far have value at most $1/n$ to player j (take a moment to verify this). So the cake remaining after iteration k has value at least $1 - (k/n) \geq 1/n$ for player j. This argument holds for any $k \in \{1, 2, \ldots, n - 1\}$ and any player j that remains after iteration k. So the queries never return "None". Part (ii) actually follows from the same argument. The cake remaining after iteration $n - 1$ has value at least $1 - (n - 1)/n = 1/n$ for the last player. This completes the proof that the allocation is proportional.

Now we show that the time complexity is $\Theta(n^2)$. To do this, we'll first argue that the number of queries is $O(n^2)$, and then argue that it is $\Omega(n^2)$. Note that the algorithm has n iterations, and at iteration i, it makes $n + 1 - i$ queries. There is one exception, which is the last iteration when only one player is left. In that case, we don't make any queries. So the total number of queries is

$$ n + (n - 1) + (n - 2) + \cdots + 2. $$

We can upper bound this as follows:

$$ n + (n - 1) + (n - 2) + \cdots + 2 \; \leq \; \underbrace{n + n + \cdots + n}_{n \text{ times}} \; = \; n^2. $$

This implies that the number of queries is $O(n^2)$. We can also lower bound the number of queries by lower bounding the first $n/2$ terms in the sum by $n/2$:

$$ n + (n - 1) + (n - 2) + \cdots + 2 \; \geq \; \underbrace{\frac{n}{2} + \frac{n}{2} + \cdots + \frac{n}{2}}_{n/2 \text{ times}} \; = \; \frac{n^2}{4}. $$

This implies that the number of queries is $\Omega(n^2)$. Hence, the number of queries is $\Theta(n^2)$. $\qquad\square$

---

[3]Note that it is perfectly fine to describe an algorithm in a paragraph as long as you explain clearly what the algorithm does. A pseudocode is not required.

Theorem 7.7 (Even-Paz Algorithm).
Assume n is a power of 2, i.e., $n = 2^t$ for some $t \in \mathbb{N}$. There is an algorithm of time complexity $\Theta(n \log n)$ that produces an allocation for the cake cutting problem that satisfies the proportionality property.

Proof. Our goal is to present a cake cutting algorithm with $\Theta(n \log n)$ complexity that produces a proportional allocation. The assumption that n is a power of 2 is there for simplicity in describing and analyzing the algorithm. Below, we first present the algorithm. Next we show that its complexity is $\Theta(n \log n)$. And finally, we show that it produces a proportional allocation.

Our algorithm will be recursive, so we give some flexibility for the input by allowing it to consist of an interval $[x, y] \subseteq [0, 1]$ and a subset of players $S \subseteq \{1, 2, \ldots, n\}$. Our algorithm's name is EP, and we would initially call it with input in which $[x, y] = [0, 1]$ and $S = \{1, 2, \ldots, n\}$. Below is the description of EP. A verbal explanation of what the algorithm does follows its description.

$[x, y]$: interval in $[0, 1]$. k: integer in $\{0, 1, 2, \ldots, n\}$.
S: subset of $\{1, 2, \ldots, n\}$ with $|S| = k$.
EP($\langle [x, y], k, S \rangle$):

1. If $k = 1$ and $S = \{i\}$ for some i, then let $A_i = [x, y]$.

2. Else:

3.   For $i \in S$, let $z_i = \text{Cut}_i(x, \text{Eval}_i(x, y)/2)$.

4.   Sort the z_i so that $z_{i_1} \leq z_{i_2} \leq \cdots \leq z_{i_k}$. Let $z^* = z_{i_{k/2}}$.

5.   Run EP($\langle [x, z^*], k/2, \{i_1, \ldots, i_{k/2}\} \rangle$).

6.   Run EP($\langle [z^*, y], k/2, \{i_{k/2+1}, \ldots, i_k\} \rangle$).

The base case of the algorithm is when there is only one player. In this case we give the whole piece $[x, y]$ to that player. Otherwise, each player i makes a mark z_i such that $V_i([x, z_i]) = \frac{1}{2} V_i([x, y])$. Let z^* denote the $n/2$ mark from the left. We first recurse on $[x, z^*]$ and the left $n/2$ players, and then we recurse on $[z^*, y]$ and the right $n/2$ players.

We have to show that the algorithm's time complexity is $\Theta(n \log n)$ and that it produces a proportional allocation. First we show that the time complexity $T(n)$ is $\Theta(n \log n)$. Observe that the recursive relation that $T(n)$ satisfies is

$$T(1) = 0, \qquad T(n) = 2n + 2T(n/2) \quad \text{for } n > 1.$$

The base case corresponds to line 1 of the algorithm, and in this case, we don't make any queries. In $T(n) = 2n + 2T(n/2)$, the $2n$ comes from line 3 where we make 2 queries for each player. The $2T(n/2)$ comes from the two recursive calls on lines 5 and 6. To solve the recursion, i.e., to figure out the formula for $T(n)$, we draw the associated *recursion tree*.

39
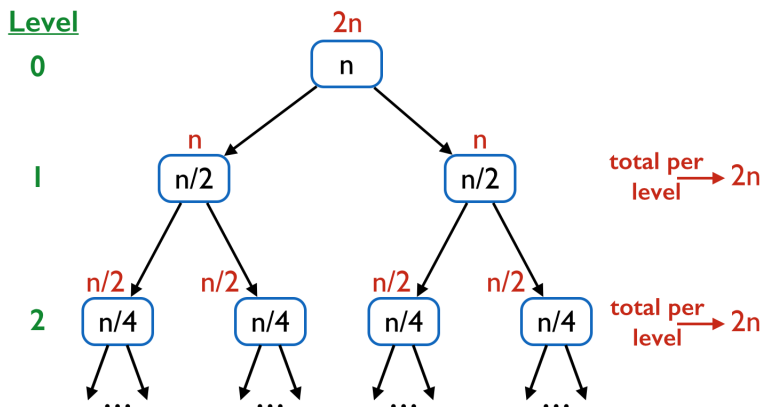
The root (top) of the tree corresponds to the input $S = \{1, 2, \ldots, n\}$ and is therefore labeled with an n. This branches off into two nodes, one corresponding to each recursive call. These nodes are labeled with $n/2$ since they correspond to recursive calls in which $|S| = n/2$. Those nodes further branch off into two nodes, and so on, until at the very bottom, we end up with nodes corresponding to inputs S with $|S| = 1$. The number of queries made for each node of the tree is provided with a label on top of the node. For example, at the root (top), we make $2n$ queries before we do our recursive calls. This is why we put a $2n$ on top of that node. Similarly, every other node can be labeled. 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 2^j nodes. The nodes that are at level j make $2n/2^j$ queries. Therefore, the total number of queries made for level j is $2n$. The only exception is the last level. The nodes at the last level correspond to the base case and don't make any queries. In total, there are exactly $1 + \log_2 n$ levels (since we are counting the root as well). Thus, the total number of queries, and hence the time complexity, is exactly $2n \log_2 n$, which is $\Theta(n \log n)$.

We now prove that the allocation obtained by the algorithm is proportional. Observe that when we make the recursive call on $[x, z^*]$ and the left $n/2$ players, all these players value $[x, z^*]$ at least at $1/2$. Similarly, when we make the recursive call on $[z^*, y]$ and the right $n/2$ players, all these players value $[z^*, y]$ at least at $1/2$. This property is preserved at each level of the recursion in the following way. At level ℓ of the recursion, the players are divided into groups of size $n/2^\ell$. If each player values the corresponding interval at least at $1/2^\ell$, then at level $\ell + 1$, the players will value the interval that they are "assigned to" at least at $1/2^{\ell+1}$. In particular, when $\ell = \log_2 n$, each group is a singleton, and each player gets assigned a piece of cake that she values at least at $1/2^{\log_2 n} = 1/n$. This shows that the allocation is proportional. $\qquad\square$

Theorem 7.8 (Edmonds-Pruhs Theorem).
Any algorithm that produces an allocation satisfying the proportionality property must have time complexity $\Omega(n \log n)$.

Chapter 8

Introduction to Graph Theory

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 $u, v \in V$.

Definition 8.2 (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 $\deg(v)$.

Definition 8.3 (d-regular graphs).
A graph $G = (V, E)$ is called *d-regular* if every vertex $v \in V$ satisfies $\deg(v) = d$.

Theorem 8.4 (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 $\sum_{v \in V} \deg(v) = 2m$. $\square$

Definition 8.5 (Paths and cycles).
Let $G = (V, E)$ be a graph. A *path* of length k in G is a sequence of <u>distinct</u> 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

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

---

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

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 8.6 (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 8.7 (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. $\square$

Definition 8.8 (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*.

Definition 8.9 (Directed graph).
A *directed graph G* is a pair (V, A), where

- V is a 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 $u = v$.

Definition 8.10 (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*.

8.2 Graph Algorithms

8.2.1 Graph searching algorithms

Definition 8.11 (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($\langle G, s \rangle$):
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.
```

Definition 8.12 (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.

Definition 8.13 (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.

8.2.2 Minimum spanning tree

Definition 8.14 (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 $c : E \rightarrow \mathbb{R}^+$. 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.[2] For convenience, we'll assume that the edges have unique edge costs, i.e. $e \neq e' \implies c(e) \neq c(e')$.

Theorem 8.15 (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 $c(e) < c(e')$).

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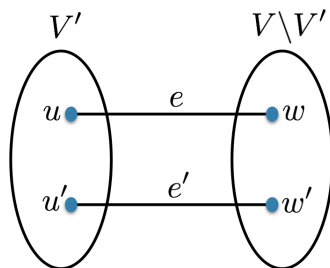
[2]Obviously this subset of edges would not contain a cycle since if it did, we could remove any edge on the cycle, preserve the connectivity property, and obtain a cheaper set. Therefore, this set forms a tree.

$T_{e-e'}$ 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. $\square$

Theorem 8.16 (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. $c : E \to \mathbb{R}^+$: edge costs.
MST($\langle G, c \rangle$):
  1   $V' = \{u\}$ (for some arbitrary $u \in V$)
  2   $E' = \emptyset$.
  3   While $V' \neq V$:
  4   Let $\{u, v\}$ be the minimum cost edge such that $u \in V'$ but $v \notin V'$.
  5   Add $\{u, v\}$ to E'.
  6   Add v to V'.
  7   Output E'.

By Theorem 8.15 (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. $\square$

8.2.3 Topological sorting

Definition 8.17 (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 $f(u) < f(v)$.
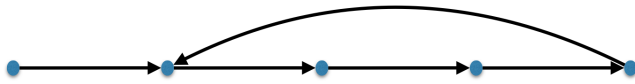
Definition 8.18 (Topological sorting problem)**.**
In the *topological sorting problem*, the input is a directed acyclic graph, and the output is a topological order of the graph.

Lemma 8.19 (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.

□

Theorem 8.20 (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 $f(u) < f(v)$.

□

Chapter 9

Matchings in Graphs

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.

Definition 9.2 (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 9.3 (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),

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

Theorem 9.4 (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 $v_1, v_2, \ldots, v_k$:
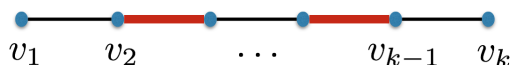

The highlighted edges represent edges in M. By the definition of an augmenting path, we know that v_1 and v_k are not matched by M. Since v_1 and v_k are not matched and the path is alternating, the number of edges on this path that are in the matching is one less than the number of edges not in the 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

---

[1]Note that a maximal matching is not necessarily a maximum matching, but a maximum matching is always a maximal matching.

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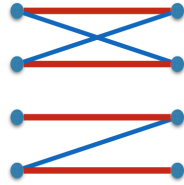
(Horizontal edges correspond to the red edges. The rest is blue.) Our goal is to find an augmenting path with respect to M in S (i.e., with respect to the blue edges), and once we do this, the proof will be complete.

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. $\qquad\square$

Definition 9.5 (Bipartite graph)**.**
A graph $G = (V, E)$ is called *bipartite* if there is a partition[2] 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 $G = (X, Y, E)$.

Definition 9.6 (k-colorable graphs)**.**
Let $G = (V, E)$ be a graph. Let $k \in \mathbb{N}^+$. A k-coloring of V is just a map $\chi : V \rightarrow C$ 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

---

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

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.

Theorem 9.7 (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. $\square$

Theorem 9.8 (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 9.4 (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. $\square$

Theorem 9.9 (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:
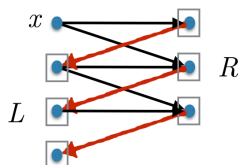
We will show that for $S = L \cup \{x\}$, we have $|S| > |N(S)|$. We need two claims to argue this.

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 (The-orem 9.4 (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. $\qquad \square$

Corollary 9.10 (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 $|S| \leq |N(S)|$.
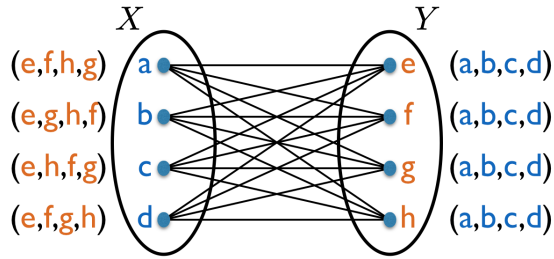
9.2 Stable Matchings

Definition 9.11 (Complete graph).
A graph $G = (V, E)$ is called *complete* if E contains all the possible edges, i.e., $\{u, v\} \in E$ for any distinct $u, v \in V$. A bipartite graph $G = (X, Y, E)$ is called complete if E contains all the possible edges between X vertices and Y vertices.

Definition 9.12 (Stable matching problem).
An instance of the *stable matching problem* is a complete bipartite graph $G = (X, Y, E)$ with $|X| = |Y|$, and a *preference list* for each node of the graph. A preference list for a node in X is an ordering of the Y vertices, and a preference list for a node in Y is an ordering of the X vertices. 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 defined as a matching that satisfies two properties:

(i) The matching is a perfect matching.

(ii) There are no *unstable* pairs. A pair of vertices (x, y) where $x \in X$ and $y \in Y$ is called *unstable* if they are not matched to each other, but they both prefer each other to the partners they are matched to.

Theorem 9.13 (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:

m is not matched at the end $\implies$ all women must be matched at the end

$\implies$ all men must be matched at the end.

This obviously leads to the desired contradiction. The second implication is quite 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.
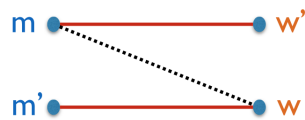
There are two cases to consider:

(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 9.14 (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 9.15 (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 $\{(m, \mathrm{best}(m)) : m \in X\}$.

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.
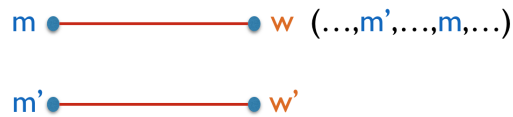
During Gale-Shapley algorithm:

right after
m got rejected
by w

m •   • w (…,m',…,m,…)

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.

Some other stable matching:
(where m and w are matched)

m •————————• w (…,m',…,m,…)

m' •————————• w'

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. □

Chapter 10

Boolean Circuits

10.1   Basic Definitions

Definition 10.1 (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.

Definition 10.2 (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 10.3 (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 10.4 (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]

10.2   3 Theorems on Circuits

Theorem 10.5 ($O(2^n)$ upper bound on circuit complexity).
Any language $L \subseteq \{0,1\}^*$ can be computed by a circuit family of size $O(2^n)$.

---

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

[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.

Proof. Let

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

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.
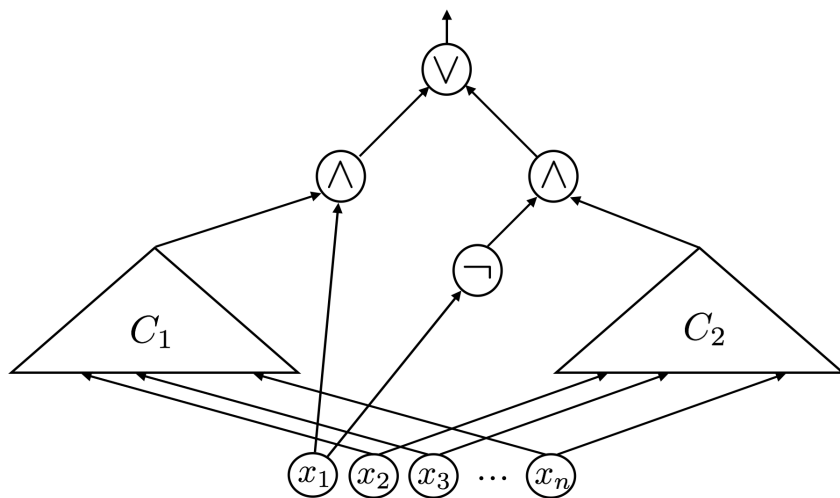
Since the circuits C_1 and C_2 compute functions on $n-1$ variables, their size is bounded by $S_{\max}(n-1)$ each. Then the size of the above circuit is at most $2S_{\max}(n-1)+5$ (the extra 5 gates are for x_1, the NOT gate, the two AND gates, and the output OR gate). Our construction above works for 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). $\square$

Proposition 10.6 (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. $\square$

Theorem 10.7 (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 (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 10.6 (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 10.8 (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. $\qquad\square$

Lemma 10.8 (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 $5s \log s$. $\qquad\square$

Theorem 10.9 (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. [5]

---

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

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 $O(T(n)^2)$.

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 3.2 (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.

tape position

c_1	$q_0 x_1$	$\Box x_2$	$\Box x_3$	$\Box x_4$	$\Box x_5$		$\cdots$		
c_2	$\Box \sqcup$	$q_1 1$	$\Box 1$	$\Box 1$	$\Box 1$		$\cdots$		
c_3	$\Box \sqcup$	$\Box 0$	$q_2 1$	$\Box 1$	$\Box 1$		$\cdots$		
c_4							$\cdots$		
$\vdots$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$		$\cdots$		
c_t							$\cdots$		

(time on the left axis; t)

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 $\Box$, 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 $A_{i-1,j+1}$.

$A_{i-1,j-1}$	$A_{i-1,j}$	$A_{i-1,j+1}$
	$A_{i,j}$	
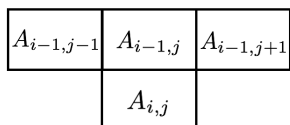
The transition function of TM M governs this transformation. Assume each cell encodes k bits of information. Note that k is a constant because $|Q|$ and $|\Gamma|$ are constant. So the transition function is of the form $\{0,1\}^{3k} \to \{0,1\}^k$. It determines the contents of a cell based on the contents of the three cells above it and it can be implemented by a circuit of constant size since k is constant.[6] (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.

---

[6]Recall that any function can be computed by a circuit (Theorem 10.5 ($O(2^n)$ upper bound on circuit complexity)).
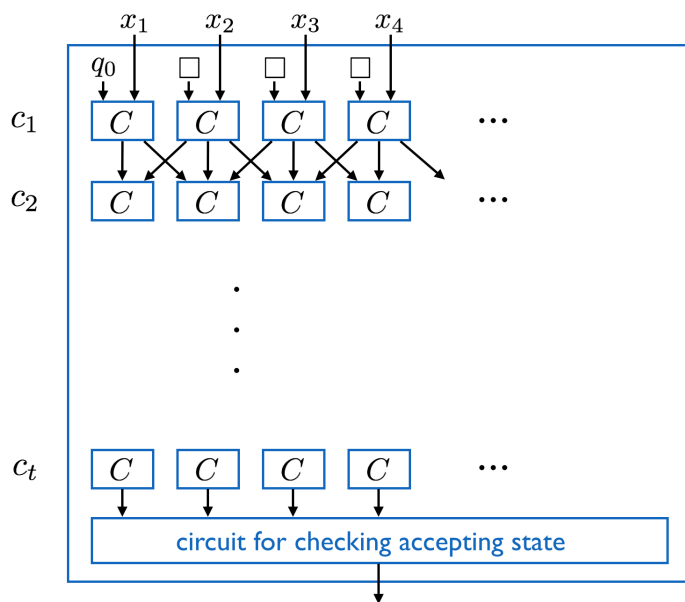
The size of the circuit is at most ct^2 for some constant c. □

Definition 10.10 (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 10.11 (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.

Chapter 11

Polynomial-Time Reductions

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 9.6
(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 *in-
dependent 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.

Definition 11.6 (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 $A \leq_m^P B$.

Theorem 11.7 (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. $\square$

Theorem 11.8 (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 (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.
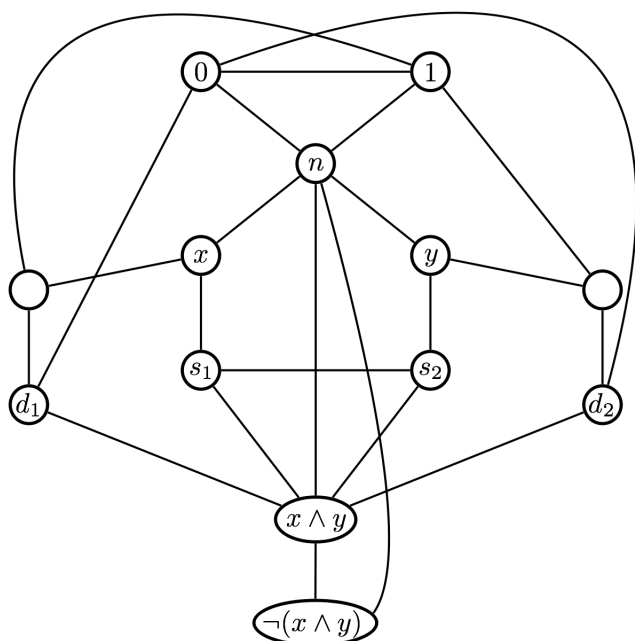
The vertices labeled with x and y correspond to the inputs of the NAND gate. The vertex labeled with $\neg(x \wedge y)$ corresponds to the output of the gate. We construct such a graph for each NAND gate of the circuit, however, we make sure that if, say, gate g_1 is an input to gate g_2, then the vertex 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. $\qquad\square$

11.2   Hardness and Completeness

Definition 11.9 ($\mathcal{C}$-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.)

Chapter 12

Non-Deterministic Polynomial Time

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).
$3\text{COL} \in \text{NP}$.

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.   □

Proposition 12.3 (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. $\square$

Proposition 12.4 (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. $\square$

Definition 12.5 (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 $C > 0$.

12.2 NP-**complete problems**

Theorem 12.6 (Cook-Levin Theorem)**.**
CIRCUIT-SAT *is* NP-complete*.*

Theorem 12.7 (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 11.8 (CIRCUIT-SAT reduces to 3COL). $\square$

**Theorem 12.8** (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.
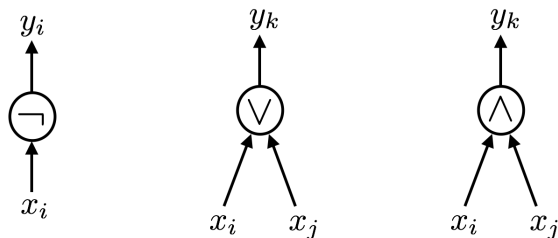
We will convert each such gate of the circuit C into a 3SAT formula. It is easy to verify that

$$y_i = \neg x_i \iff (x_i \vee y_i) \wedge (\neg x_i \vee \neg y_i),$$
$$y_k = x_i \vee x_j \iff (y_k \vee \neg x_i) \wedge (y_k \vee \neg x_j) \wedge (\neg y_k \vee x_i \vee x_j),$$
$$y_k = x_i \wedge x_j \iff (\neg y_k \vee x_i) \wedge (\neg y_k \vee x_j) \wedge (y_k \vee \neg x_i \vee \neg x_j).$$

So the behavior of each gate can be represented using a Boolean formula. As an example, consider the circuit below.
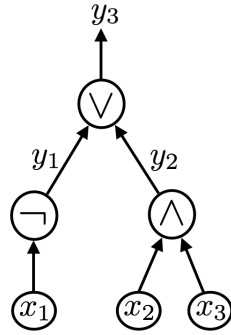
In this case, we would let

$$\text{Clause}_1 = (x_1 \vee y_1) \wedge (\neg x_1 \vee \neg y_1)$$
$$\text{Clause}_2 = (\neg y_2 \vee x_2) \wedge (\neg y_2 \vee x_3) \wedge (y_2 \vee \neg x_2 \vee \neg x_3)$$
$$\text{Clause}_3 = (y_3 \vee \neg y_1) \wedge (y_3 \vee \neg y_2) \wedge (\neg y_3 \vee y_1 \vee y_2),$$

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. $\qquad\square$

Theorem 12.9 (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 (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:

x_1 $\neg x_1$

$\neg x_2$ x_2

x_3 x_3

x_1 x_1 $\neg x_1$
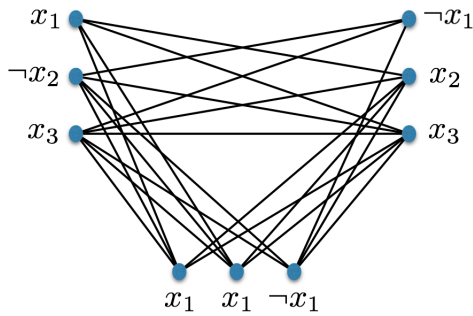
We now prove that φ is satisfiable if and only if G has a clique of size m. If φ is satisfiable, then there is an assignment to the variables such that in each clause, there is at least one literal set to True. We claim that the vertices that correspond to these literals form a clique of size m in G. 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 12.10 (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 (IS is in NP) asks you to show that IS is in NP, so we show that IS is NP-hard. By Theorem 12.9 (CLIQUE is NP-complete), we know that CLIQUE is NP-hard, and by Theorem 11.7 (CLIQUE reduces to IS) we know that CLIQUE $\leq^P_m$ IS. By the transitivity of reductions, we conclude that IS is also 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 12.3 (CIRCUIT-SAT is in NP). To show that is is NP-hard, we will show that for any $L \in$ NP, $L \leq^P_m$ 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

$$x \in L \iff \exists u, |u| = c|x|^k, V(x, u) = 1$$
$$\iff C_x \text{ is satisfiable.}$$

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). $\square$

Chapter 13

Computational Social Choice

13.1 Basic Definitions and Results

Definition 13.1 (Election, voters, alternatives, preference profile, voting rule)**.**
An *election* is specified by 4 objects:

- **Voters:** a set of n *voters* $N = \{1, 2, \ldots, n\}$;

- **Alternatives:** a set of m *alternatives* denoted by A;

- **Preference profile:** for each voter, a ranking over the alternatives from rank 1 to rank m;

- **Voting rule:** a function that maps a preference profile to an alternative.

The output of the voting rule is called the *winner* of the election.

Definition 13.2 (Pairwise election)**.**
In a *pairwise election*, $m = 2$ and an alternative x wins if the majority of voters prefer x over the other alternative.

Definition 13.3 (Condorcet winner)**.**
We say that an alternative is a *Condorcet winner* if it beats every other alternative in a pairwise election.

Definition 13.4 (Various voting rules)**.**
The following are definitions of various voting rules. [1]

- **Plurality:** Each voter awards one point to their top-ranked alternative. The alternative with the most points is declared the winner.

- **Borda count:** Each voter awards $m - k$ points to their k'th ranked alternative. The alternative with the most points is declared the winner.

- **Plurality with runoff:** There are 2 rounds. In the first round, a plurality rule is applied to identify the top two alternatives. In the second round, a pairwise election is done to determine the winner.

- **Single transferable vote (STV):** There are $m - 1$ rounds. In each round a plurality rule is applied to identify and eliminate the alternative with the lowest points. The alternative that survives every round is selected as the winner.

- **Copeland:** An alternative's score is the number of alternatives it would beat in a pairwise election. The winner of the election is the alternative with the highest score.

- **Dodgson:** Given a preference profile, define the *Dodgson score* of an alternative x as the number of swaps between adjacent alternatives needed in the preference profile in order to make x a Condorcet winner. In Dodgson voting rule, the winner is an alternative with the minimum Dodgson score.

---

[1] We'll assume that ties are broken deterministically according to some order on the alternatives.

Theorem 13.5 (Bartholdi-Tovey-Trick 1989).
Consider the following computational problem. Given as input an election, an alternative x in the election, and a number k, the output is True if and only if the Dodgson score of x is at most k. This problem is NP-complete.

Definition 13.6 (Types of voting rules).
We call a voting rule

- **majority consistent** if given a preference profile such that a majority of the voters rank an alternative x first, then x is the winner of the election;

- **Condorcet consistent** if given a preference profile such that there is an alternative x that beats every other alternative in a pairwise election (i.e. x is a Condorcet winner), then x is the winner of the election;

- **onto** if for any alternative, there is a preference profile such that the alternative wins;

- **dictatorial** if there is a voter v such that no matter what the preference profile is, the winner is v's most preferred alternative;

- **constant** if no matter what the preference profile is, the same alternative is the winner.

(The first 3 are considered to be desirable types of voting rules, whereas the last 2 are considered undesirable.)

Definition 13.7 (Manipulation, strategy-proof (SP) voting rule).
Consider an election in which alternative x wins. We say that a voter can *manipulate* the voting rule of the election if by changing their preference list, they can change the winner of the election to an alternative y that the voter ranks higher than x. A voting rule is called *strategy-proof* if no voter can manipulate the voting rule.

Theorem 13.8 (Gibbard-Satterthwaite).
If $m \geq 3$ then any voting rule that is strategy-proof and onto is dictatorial. Equivalently, any voting rule that is onto and nondictatorial is manipulable.

Definition 13.9 (r-Manipulation problem).
Let r be some voting rule. In the r-*Manipulation problem*, the input is an election, a voter (called the *manipulator*), and an alternative (called the *preferred candidate*). The output is True if there exists a ranking over the alternatives for the manipulator that makes the preferred candidate a *unique winner* of the election.

Theorem 13.10 (Greedy algorithm solves r-Manipulation problem for various voting rules).
The greedy algorithm above is a polynomial-time algorithm that correctly solves the r-Manipulation problem for

$$r \in \{plurality, Borda\ count, plurality\ with\ runoff, Copeland\}.$$

Theorem 13.11 (Bartholdi-Orlin 1991).
The r-Manipulation problem is NP-complete for r being the single transferable voting (STV) rule.

Chapter 14

Approximation Algorithms

14.1 Basic Definitions

Definition 14.1 (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 14.2 (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 14.1 (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 $f(x, y) = \text{no}$.

Definition 14.3 (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, $f(x, A(x)) \leq \alpha \cdot \text{OPT}(x)$.

- 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, $f(x, A(x)) \geq \beta \cdot \text{OPT}(x)$.

14.2 Examples of Approximation Algorithms

Lemma 14.4 (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, $|S| \geq |M|$.

---

[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*.

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.) $\square$

Theorem 14.5 (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 14.4 (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. $\square$

Definition 14.6 (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 14.7 ((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 8.4 (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 14.8 (Traveling salesperson problem (TSP)).
In the *Traveling salesperson problem*, the input is a connected graph $G = (V, E)$ together with edge costs $c : E \rightarrow \mathbb{N}$. 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 14.9 (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. $c : E \rightarrow \mathbb{N}$: 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 8.16 (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.
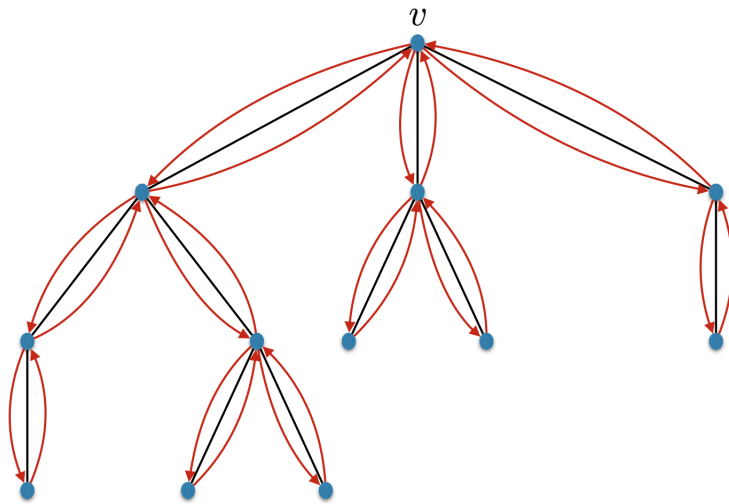
Let $c(L)$ denote the total cost of the tour that the algorithm outputs. Let L^* be a optimal solution (so $c(L^*) = \mathrm{OPT}(\langle G, c \rangle)$). Our goal is to show that $c(L) \leq 2c(L^*)$. The graph induced by L^* is a connected graph on all of the vertices. Let T^* be a spanning tree within 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. $\qquad\qquad\square$

Definition 14.10 (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.

Chapter 15

Probability Theory

15.1 Probability I: The Basics

15.1.1 Basic Definitions

Definition 15.1 (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*.

Definition 15.2 (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 15.3 (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 $\Omega \backslash E$.

Definition 15.4 (Disjoint events).
We say that two events A and B are *disjoint* if $A \cap B = \emptyset$.

Definition 15.5 (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

$$\mathbf{Pr}[A \mid B] = \frac{\mathbf{Pr}[A \cap B]}{\mathbf{Pr}[B]}. \tag{15.1}$$

15.1.2 Three Useful Rules

Proposition 15.6 (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. $\square$

Proposition 15.7 (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,

$$\mathbf{Pr}[B] = \mathbf{Pr}[A_1] \cdot \mathbf{Pr}[B \mid A_1] + \mathbf{Pr}[A_2] \cdot \mathbf{Pr}[B \mid A_2] + \cdots + \mathbf{Pr}[A_n] \cdot \mathbf{Pr}[B \mid A_n].$$

Proposition 15.8 (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 $\mathbf{Pr}[A \mid B]$. $\square$

15.1.3 Independence

Definition 15.9 (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 $S \subseteq \{1, 2, \ldots, n\}$,

$$\mathbf{Pr}\left[\bigcap_{i \in S} A_i\right] = \prod_{i \in S} \mathbf{Pr}[A_i].$$

15.2   Probability II: Random Variables

15.2.1   Basics of random variables

Definition 15.10 (Random variable).
A *random variable* is a function $X : \Omega \to \mathbb{R}$.

Definition 15.11 (Common events through a random variable).
Let X be a random variable and $x \in \mathbb{R}$ be some real value. We use

$$
\begin{aligned}
X = x \quad &\text{to denote the event } \{\ell \in \Omega : X(\ell) = x\}, \\
X \leq x \quad &\text{to denote the event } \{\ell \in \Omega : X(\ell) \leq x\}, \\
X \geq x \quad &\text{to denote the event } \{\ell \in \Omega : X(\ell) \geq x\}, \\
X < x \quad &\text{to denote the event } \{\ell \in \Omega : X(\ell) < x\}, \\
X > x \quad &\text{to denote the event } \{\ell \in \Omega : X(\ell) > x\}.
\end{aligned}
$$

For example, $\mathbf{Pr}[X = x]$ denotes $\mathbf{Pr}[\{\ell \in \Omega : X(\ell) = x\}]$. More generally, for $S \subseteq \mathbb{R}$, we use

$$
X \in S \quad \text{to denote the event } \{\ell \in \Omega : X(\ell) \in S\}.
$$

Definition 15.12 (Probability mass function (PMF)).
Let $X : \Omega \to \mathbb{R}$ be a random variable. The *probability mass function* (PMF) of X is a function $p_X : \mathbb{R} \to [0, 1]$ such that for any $x \in \mathbb{R}$, $p_X(x) = \mathbf{Pr}[X = x]$.

Definition 15.13 (Expected value of a random variable).
Let X be a random variable. The *expected value* of X, denoted $\mathbf{E}[X]$, is defined as follows:

$$
\mathbf{E}[X] = \sum_{\ell \in \Omega} \mathbf{Pr}[\ell] \cdot X(\ell).
$$

Equivalently,

$$
\mathbf{E}[X] = \sum_{x \in \mathrm{range}(X)} \mathbf{Pr}[X = x] \cdot x,
$$

where $\mathrm{range}(X) = \{X(\ell) : \ell \in \Omega\}$.

Proposition 15.14 (Linearity of expectation).
Let X and Y be two random variables, and let $c_1, c_2 \in \mathbb{R}$ be some constants. Then $\mathbf{E}[c_1 X + c_2 Y] = c_1 \mathbf{E}[X] + c_2 \mathbf{E}[Y]$.

Proof. Define the random variable Z as $Z = c_1 X + c_2 Y$. Then using the definition of expected value, we have

$$
\begin{aligned}
\mathbf{E}[c_1 X + c_2 Y] &= \mathbf{E}[Z] \\
&= \sum_{\ell \in \Omega} \mathbf{Pr}[\ell] \cdot Z(\ell) \\
&= \sum_{\ell \in \Omega} \mathbf{Pr}[\ell] \cdot (c_1 X(\ell) + c_2 Y(\ell)) \\
&= \sum_{\ell \in \Omega} \mathbf{Pr}[\ell] \cdot c_1 X(\ell) + \mathbf{Pr}[\ell] \cdot c_2 Y(\ell) \\
&= \left(\sum_{\ell \in \Omega} \mathbf{Pr}[\ell] \cdot c_1 X(\ell) \right) + \left(\sum_{\ell \in \Omega} \mathbf{Pr}[\ell] \cdot c_2 Y(\ell) \right) \\
&= c_1 \left(\sum_{\ell \in \Omega} \mathbf{Pr}[\ell] \cdot X(\ell) \right) + c_2 \left(\sum_{\ell \in \Omega} \mathbf{Pr}[\ell] \cdot Y(\ell) \right) \\
&= c_1 \mathbf{E}[X] + c_2 \mathbf{E}[Y],
\end{aligned}
$$

as desired. $\qquad \square$

Corollary 15.15 (Linearity of expectation 2).
*Let $X_1, X_2, \ldots, X_n$ be random variables, and $c_1, c_2, \ldots, c_n \in \mathbb{R}$ be some constants.
Then*

$$
\mathbf{E}[c_1 X_1 + c_2 X_2 + \cdots + c_n X_n] = c_1 \mathbf{E}[X_1] + c_2 \mathbf{E}[X_2] + \cdots + c_n \mathbf{E}[X_n].
$$

In particular, when all the c_i's are 1, we get

$$
\mathbf{E}[X_1 + X_2 + \cdots + X_n] = \mathbf{E}[X_1] + \mathbf{E}[X_2] + \cdots + \mathbf{E}[X_n].
$$

Definition 15.16 (Indicator random variable).
Let $E \subseteq \Omega$ be some event. The *indicator random variable* with respect to E is
denoted by I_E and is defined as

$$
I_E(\ell) = \begin{cases} 1 & \text{if } \ell \in E, \\ 0 & \text{otherwise.} \end{cases}
$$

Proposition 15.17 (Expectation of an indicator random variable).
Let E be an event. Then $\mathbf{E}[I_E] = \mathbf{Pr}[E]$.

Proof. By the definition of expected value,

$$
\begin{aligned}
\mathbf{E}[I_E] &= \mathbf{Pr}[I_E = 1] \cdot 1 + \mathbf{Pr}[I_E = 0] \cdot 0 \\
&= \mathbf{Pr}[I_E = 1] \\
&= \mathbf{Pr}[\{\ell \in \Omega : I_E(\ell) = 1\}] \\
&= \mathbf{Pr}[\{\ell \in \Omega : \ell \in E\}] \\
&= \mathbf{Pr}[E].
\end{aligned}
$$

$\qquad \square$

Definition 15.18 (Conditional expectation).
Let X be a random variable and E be an event. The *conditional expectation* of X given the event E, denoted by $\mathbf{E}[X \mid E]$, is defined as

$$\mathbf{E}[X \mid E] = \sum_{x \in \text{range}(X)} x \cdot \mathbf{Pr}[X = x \mid E].$$

Proposition 15.19 (Law of total expectation).
Let X be a random variable and $A_1, A_2, \ldots, A_n$ be events that partition the sample space Ω. Then

$$\mathbf{E}[X] = \mathbf{E}[X \mid A_1] \cdot \mathbf{Pr}[A_1] + \mathbf{E}[X \mid A_2] \cdot \mathbf{Pr}[A_2] + \cdots + \mathbf{E}[X \mid A_n] \cdot \mathbf{Pr}[A_n].$$

Definition 15.20 (Independent random variables).
Two random variables X and Y are *independent* if for all $x, y \in \mathbb{R}$, the events $X = x$ and $Y = y$ are independent. The definition generalizes to more than two random variables analogous to Definition 15.9 (Independent events).

15.2.2 The most fundamental inequality in probability theory

Theorem 15.21 (Markov's inequality).
Let X be a non-negative random variable with non-zero expectation. Then for any $c > 0$,

$$\mathbf{Pr}[X \geq c\,\mathbf{E}[X]] \leq \frac{1}{c}.$$

Proof. Let I be the indicator random variable for the event $[X \geq c\,\mathbf{E}[X]]$. Then $\mathbf{E}[I] = \mathbf{Pr}[X \geq c\,\mathbf{E}[X]]$, which corresponds to the LHS of the inequality above. We claim that

$$I \leq \frac{X}{c\,\mathbf{E}[X]}.$$

First, observe that the result follows from this claim: if we take the expectations of both sides of the inequality, we get $\mathbf{E}[I] \leq \mathbf{E}[\frac{X}{c\,\mathbf{E}[X]}] = \frac{\mathbf{E}[X]}{c\,\mathbf{E}[X]} = \frac{1}{c}$, as desired.

We now prove the above claim. The indicator random variable I can be equal to either 0 or 1. If it is 0, then the inequality holds trivially because by assumption X is non-negative (so $\mathbf{E}[X]$ is also non-negative) and $c > 0$. If on the other hand $I = 1$, then the above inequality becomes equivalent to $1 \leq \frac{X}{c\,\mathbf{E}[X]} \Leftrightarrow X \geq c\,\mathbf{E}[X]$, which must hold by the definition of I.[1] $\qquad \square$

---

[1]The notation $I = 1$ may seem strange at first. After all, 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 $I = 1$, we are considering all $\ell \in \Omega$ such that $I(\ell) = 1$. We have a similar situation for $I = 0$. When we want to prove an inequality involving random variables, we must show that the inequality holds for all inputs $\ell \in \Omega$.

15.2.3 Three popular random variables

Definition 15.22 (Bernoulli random variable).
Let $0 < p < 1$ be some parameter. If X is a random variable with probability mass function $p_X(1) = p$ and $p_X(0) = 1 - p$, then we say that X has a *Bernoulli distribution with parameter p* (we also say that X is a Bernoulli random variable). We write $X \sim \text{Bernoulli}(p)$ to denote this. The parameter p is often called the *success* probability.

Definition 15.23 (Binomial random variable).
Let $X = X_1 + X_2 + \cdots + X_n$, where the X_i's are independent and for all i, $X_i \sim \text{Bernoulli}(p)$. Then we say that X has a *binomial distribution with parameters n and p* (we also say that X is a binomial random variable). We write $X \sim \text{Bin}(n, p)$ to denote this.

Definition 15.24 (Geometric random variable).
Let X be a random variable with probability mass function p_X such that for $n \in \{1, 2, \ldots\}$, $p_X(n) = (1 - p)^{n-1}p$. Then we say that X has a *geometric distribution with parameter p* (we also say that X is a geometric random variable). We write $X \sim \text{Geometric}(p)$ to denote this.

Chapter 16

Randomized Algorithms

16.1 Monte Carlo and Las Vegas Algorithms

Definition 16.1 (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 16.2 (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.

16.2 Monte Carlo Algorithm for the Minimum Cut Problem

Definition 16.3 (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 14.6 (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 $V \setminus S$ is minimized. Such a set S is called a *cut* and the size of the cut is the number of edges between S and $V \setminus S$ (note that the size of the cut is not the number of vertices). We denote this problem by MIN-CUT.

Definition 16.4 (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 16.5 (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. $V' = V \setminus \{u, v\} \cup \{uv\}$. The multi-set of edges E' is defined as follows:

---

[1]Note that this definition does not allow for self-loops.

- 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:
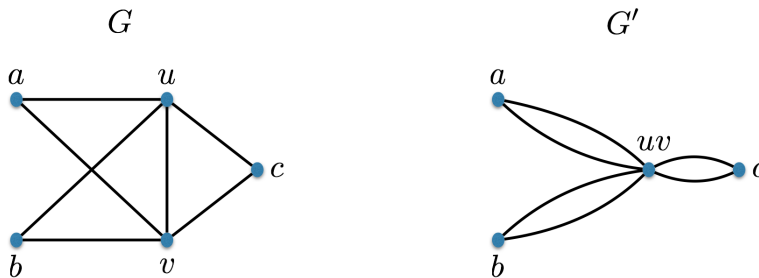
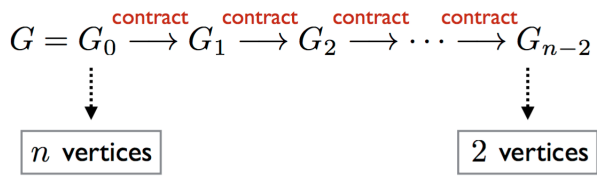**Theorem 16.6** (Contraction algorithm for min cut).
There is a polynomial-time Monte-Carlo algorithm that solves the MIN-CUT *problem with error probability at most* $1/e^n$*, where n is the number of vertices in the input graph.*

Proof. The algorithm has two phases. The description of the *first phase* is as follows.

> 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.

$$G = G_0 \xrightarrow{\text{contract}} G_1 \xrightarrow{\text{contract}} G_2 \xrightarrow{\text{contract}} \cdots \xrightarrow{\text{contract}} G_{n-2}$$

n **vertices** $\qquad\qquad$ 2 **vertices**

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

$$\mathbf{Pr}[\text{algorithm finds } F] = \mathbf{Pr}[\overline{E}_1 \cap \overline{E}_2 \cap \ldots \cap \overline{E}_{n-2}].$$

Using Proposition 15.6 (Chain rule), we have

$$\mathbf{Pr}[\overline{E}_1 \cap \overline{E}_2 \cap \ldots \cap \overline{E}_{n-2}] =$$
$$\mathbf{Pr}[\overline{E}_1] \cdot \mathbf{Pr}[\overline{E}_2 \mid \overline{E}_1] \cdot \mathbf{Pr}[\overline{E}_3 \mid \overline{E}_1 \cap \overline{E}_2] \cdots \mathbf{Pr}[\overline{E}_{n-2} \mid \overline{E}_1 \cap \overline{E}_2 \cap \ldots \cap \overline{E}_{n-3}]. \tag{16.1}$$

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

$$2m = \sum_{v \in V} \deg(v) \geq |F| \cdot n, \tag{16.2}$$

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 ((16.1)) above, we can write

$$\mathbf{Pr}[\text{algorithm finds } F] \geq$$
$$\left(1 - \frac{2}{n}\right) \cdot \mathbf{Pr}[\overline{E}_2 \mid \overline{E}_1] \cdot \mathbf{Pr}[\overline{E}_3 \mid \overline{E}_1 \cap \overline{E}_2] \cdots \mathbf{Pr}[\overline{E}_{n-2} \mid \overline{E}_1 \cap \overline{E}_2 \cap \ldots \cap \overline{E}_{n-3}].$$

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

$$\mathbf{Pr}[\overline{E}_2 \mid \overline{E}_1] = 1 - \mathbf{Pr}[E_2 \mid \overline{E}_1] = 1 - \frac{|F|}{\ell_1}.$$

---

[2]We are not giving a detailed argument for this, but please do verify this for yourself.

As before, using Observation 2, for any v in G_1, $|F| \leq \deg_{G_1}(v)$. Therefore, the analog of Inequality ((16.2)) 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

$\mathbf{Pr}[\text{algorithm finds } F] \geq$

$$\left(1 - \frac{2}{n}\right) \cdot \left(1 - \frac{2}{n-1}\right) \cdot \mathbf{Pr}[\overline{E}_3 \mid \overline{E}_1 \cap \overline{E}_2] \cdots \mathbf{Pr}[\overline{E}_{n-2} \mid \overline{E}_1 \cap \overline{E}_2 \cap \ldots \cap \overline{E}_{n-3}].$$

Applying the same reasoning for the rest of the terms in the product above, we get

$\mathbf{Pr}[\text{algorithm finds } F] \geq$

$$\left(1 - \frac{2}{n}\right) \cdot \left(1 - \frac{2}{n-1}\right) \cdot \left(1 - \frac{2}{n-2}\right) \cdots \left(1 - \frac{2}{n-(n-3)}\right) =$$
$$\left(\frac{n-2}{n}\right) \cdot \left(\frac{n-3}{n-1}\right) \cdot \left(\frac{n-4}{n-2}\right) \cdots \left(\frac{2}{4}\right) \cdot \left(\frac{1}{3}\right).$$

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

$$\mathbf{Pr}[\text{our algorithm fails to find a min-cut}] = \mathbf{Pr}[A_1 \cap \cdots \cap A_t]$$
$$= \mathbf{Pr}[A_1] \cdots \mathbf{Pr}[A_t]$$
$$= \mathbf{Pr}[A_1]^t.$$

From the analysis of the first phase, we know that
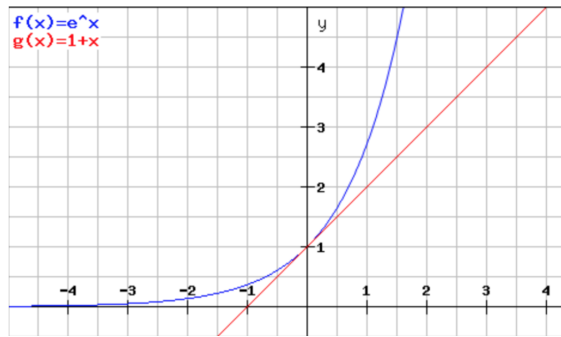
$$\mathbf{Pr}[A_1] \leq 1 - \frac{1}{n^2}.$$

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

$$\mathbf{Pr}[\text{our algorithm fails to find a min-cut}] \le (1 + x)^t \le e^{xt} = e^{-t/n^2}.$$

For $t = n^3$, this probability is upper bounded by $1/e^n$, as desired. $\square$