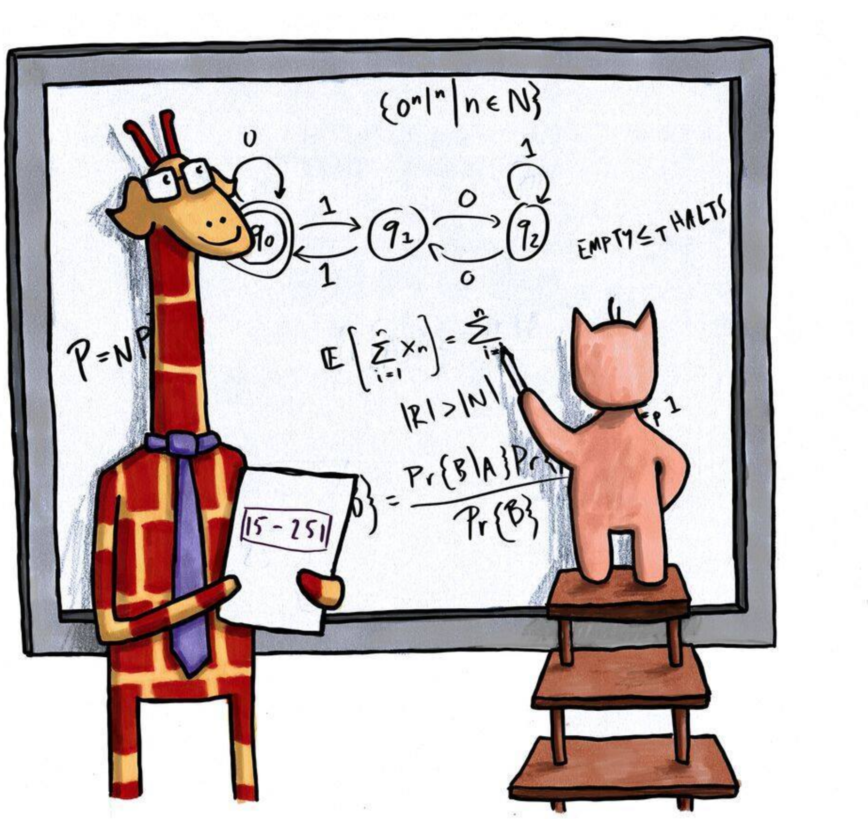
CMU 15-251, Fall 2017
Great Ideas in Theoretical Computer Science

Course Notes: Condensed 1



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

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

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

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

Contents

1 Strings and Encodings **1**
1.1 Alphabets and Strings . . . 2
1.2 Languages . . . 2
1.3 Encodings . . . 3
1.4 Computational Problems and Decision Problems . . . 4

2 Deterministic Finite Automata **5**
2.1 Basic Definitions . . . 6
2.2 Irregular Languages . . . 7
2.3 Closure Properties of Regular Languages . . . 7

3 Turing Machines **9**
3.1 Basic Definitions . . . 10
3.2 Decidable Languages . . . 11

4 Countable and Uncountable Sets **13**
4.1 Basic Definitions . . . 14
4.2 Countable Sets . . . 15
4.3 Uncountable Sets . . . 15

5 Undecidable Languages **17**
5.1 Existence of Undecidable Languages . . . 18
5.2 Examples of Undecidable Languages . . . 18
5.3 Undecidability Proofs by Reductions . . . 18

6 Time Complexity **19**
6.1 Big-O, Big-Omega and Theta . . . 20
6.2 Worst-Case Running Time of Algorithms . . . 20
6.3 Complexity of Algorithms with Integer Inputs . . . 21

7 The Science of Cutting Cake **23**
7.1 The Problem and the Model . . . 24
7.2 Cake Cutting Algorithms in the Robertson-Webb Model . . . 25

8 Introduction to Graph Theory **27**
8.1 Basic Definitions . . . 28
8.2 Graph Algorithms . . . 29
8.2.1 Graph searching algorithms . . . 29
8.2.2 Minimum spanning tree . . . 30
8.2.3 Topological sorting . . . 31

9 Matchings in Graphs **33**
9.1 Maximum Matchings . . . 34
9.2 Stable Matchings . . . 35

10 Boolean Circuits **37**
10.1 Basic Definitions . . . 38

10.2 3 Theorems on Circuits 38

11 Polynomial-Time Reductions **41**
11.1 Cook and Karp Reductions 42
11.2 Hardness and Completeness 43

12 Non-Deterministic Polynomial Time **45**
12.1 Non-Deterministic Polynomial Time NP 46
12.2 NP-complete problems 46
12.3 Proof of Cook-Levin Theorem 47

13 Computational Social Choice **49**
13.1 Basic Definitions and Results 50

14 Approximation Algorithms **53**
14.1 Basic Definitions 54
14.2 Examples of Approximation Algorithms 54

15 Probability Theory **57**
15.1 Probability I: The Basics 58
15.1.1 Basic Definitions 58
15.1.2 Three Useful Rules 59
15.1.3 Independence 59
15.2 Probability II: Random Variables 59
15.2.1 Basics of random variables 59
15.2.2 The most fundamental inequality in probability theory . 61
15.2.3 Three popular random variables 61

16 Randomized Algorithms **63**
16.1 Monte Carlo and Las Vegas Algorithms 64
16.2 Monte Carlo Algorithm for the Minimum Cut Problem 64

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 $a_1a_2a_3\ldots$, where each $a_i \in \Sigma$. The string with no symbols is called the *empty string* and is denoted by ϵ.

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,

$$\Sigma^* = \{a_1a_2\ldots a_n : n \in \mathbb{N}, \text{and } a_i \in \Sigma \text{ for all } i\}.$$

Definition 1.5 (Reversal of a string)**.**
For a string $w = a_1a_2\ldots a_n$, the *reversal* of w, denoted w^R, is the string $w^R = a_na_{n-1}\ldots a_1$.

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_1L_2 or $L_1 \cdot L_2$, as the language

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

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,

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

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,

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

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 $\text{Enc} : A \to \Sigma^*$. We denote the encoding of $a \in A$ by $\langle a \rangle$.[3]

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

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

[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.*

Theorem 2.8 (A unary non-regular language).
*Let $\Sigma = \{a\}$. The language $L = \{a^{2^n} : n \in \mathbb{N}\}$ is **not** regular.*

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.

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.

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.

[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.

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 not contain the *blank symbol* $\sqcup$
(which we refer to as the *input alphabet*);
- Γ is a finite set such that $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$
(which we refer to as the *tape alphabet*);
- δ is a function of the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\text{L}, \text{R}\}$
(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^*.
$U(\langle M, x\rangle)$:
Simulate M on input x (i.e. run $M(x)$).
If it accepts, accept.
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:

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

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

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

Theorem 3.10 (EQ_{DFA} is decidable)**.**
The language EQ_{DFA} *is decidable.*

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.

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

Proposition 4.8 (Σ^* is countable)**.**
Let Σ be a finite set. Then Σ^* is countable.

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

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.

4.3 Uncountable Sets

Theorem 4.11 (Cantor's Theorem)**.**
For any set A, $|\mathcal{P}(A)| > |A|$.

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.

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.

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 not decidable.

5.2 Examples of Undecidable Languages

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

Theorem 5.3 (Turing's Theorem).
The language HALTS *is undecidable.*

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

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

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

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

Theorem 5.5 (ACCEPTS is undecidable).
The language ACCEPTS *is undecidable.*

Theorem 5.6 (EMPTY is undecidable).
The language EMPTY *is undecidable.*

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

5.3 Undecidability Proofs by Reductions

Theorem 5.8 ($\text{HALTS} \leq \text{EMPTY}$).
$\text{HALTS} \leq \text{EMPTY}$.

Theorem 5.9 ($\text{EMPTY} \leq \text{HALTS}$).
$\text{EMPTY} \leq \text{HALTS}$.

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 $b > 1$,

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

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

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 $\sum_{j \in N} V_i(A_j) = 1$.

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

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.

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.

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

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

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

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

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*.

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

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.

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(⟨G, s⟩):
Put s into bag.
While bag is non-empty:
    Pick an arbitrary vertex v from bag.
    If v is unmarked:
        Mark v.
        For each neighbor w of v:
            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⟩):
Put (⊥, s) into bag.
While bag is non-empty:
    Pick an arbitrary tuple (p, v) from bag.
    If v is unmarked:
        Mark v.
        parent(v) = p.
        For each neighbor w of v:
            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 $c : E \to \mathbb{R}^+$. The output is a subset of the edges of minimum total cost such that, in the graph restricted to these edges, all the vertices of G are connected.[2] For convenience, we'll assume that the edges have unique edge costs, i.e. $e \neq e' \implies c(e) \neq c(e')$.

[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.

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.

Theorem 8.16 (Jarník-Prim algorithm for MST).
There is an algorithm that solves the MST problem in polynomial time.

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.

Theorem 8.20 (Topological sort via DFS).
There is a $O(n + m)$-time algorithm that solves the topological sorting problem.

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.

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

Theorem 9.7 (Characterization of bipartite graphs)**.**
A graph is bipartite if and only if it contains no odd-length cycles.

---

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

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

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.

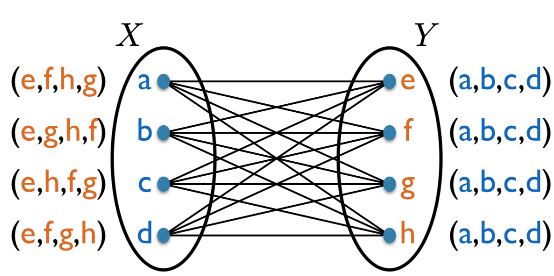
Theorem 9.9 (Hall's Theorem)**.**
Let $G = (X, Y, E)$ be a bipartite graph. For a subset S of the vertices, let $N(S) = \bigcup_{v \in S} N(v)$. Then G has a matching covering all the vertices in X if and only if for all $S \subseteq X$, we have $|S| \leq |N(S)|$.

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.

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 $\{(m, \text{best}(m)) : m \in X\}$.

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.

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}.

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* $2^n/5n$.

Lemma 10.8 (Counting circuits)**.**
The number of possible circuits of size at most s *is less than or equal to* $2^{5s \log s}$.

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

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 $L \in \mathsf{P}$*, 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* $L \notin \mathsf{P}$.

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 *independent set* if there is no edge between any two vertices in the subset. We say that G contains an independent set of size k if there is a subset of the vertices of size k that forms an independent set.

In the *independent set problem*, the input is an undirected graph $G = (V, E)$ and a number $k \in \mathbb{N}^+$, and the output is True if and only if the graph contains an independent set of size k. We denote this problem by IS. The corresponding language is

$$\{\langle G, k \rangle : G \text{ is a graph, } k \in \mathbb{N}^+, G \text{ contains an independent set of size } k\}.$$

Definition 11.4 (Circuit satisfiability problem)**.**
We say that a circuit is *satisfiable* if there is 0/1 assignment to the input gates that makes the circuit output 1. In the *circuit satisfiability problem*, the input is a Boolean circuit, and the output is True if and only if the circuit is satisfiable. We denote this problem by CIRCUIT-SAT. The corresponding language is

$$\{\langle C \rangle : C \text{ is a Boolean circuit that is satisfiable}\}.$$

Definition 11.5 (Boolean satisfiability problem)**.**
Let $x_1, \ldots, x_n$ be Boolean variables, i.e., variables that can be assigned True or False. A *literal* refers to a Boolean variable or its negation. A *clause* is an "OR" of literals. For example, $x_1 \vee \neg x_3 \vee x_4$ is a clause. A Boolean formula in *conjunctive normal form* (CNF) is an "AND" of clauses. For example,

$$(x_1 \vee \neg x_3 \vee x_4) \wedge (x_2 \vee x_2 \vee x_4) \wedge (x_1 \vee \neg x_1 \vee \neg x_5)$$

is a CNF formula. We say that a Boolean formula is *satisfiable* if there is a 0/1 assignment to the Boolean variables that makes the formula evaluate to 1.

In the CNF *satisfiability problem*, the input is a CNF formula, and the output is True if and only if the formula is satisfiable. We denote this problem by SAT. The corresponding language is

$$\{\langle\varphi\rangle : \varphi \text{ is a satisfiable CNF formula}\}.$$

In a variation of SAT, we restrict the input formula such that every clause has exactly 3 literals (we call such a formula a 3CNF formula). This variation of the problem is denoted by 3SAT.

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.

Theorem 11.8 (CIRCUIT-SAT reduces to 3COL)**.**
CIRCUIT-SAT $\leq_m^P$ 3COL.

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)**.**
3COL $\in$ NP.

Proposition 12.3 (CIRCUIT-SAT is in NP)**.**
CIRCUIT-SAT $\in$ NP.

Proposition 12.4 (P is contained in NP)**.**
P $\subseteq$ NP.

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.

Theorem 12.8 (3SAT is NP-complete)**.**
3SAT *is* NP-complete.

Theorem 12.9 (CLIQUE is NP-complete)**.**
CLIQUE *is* NP-complete.

Theorem 12.10 (IS is NP-complete)**.**
IS *is* NP-complete.

12.3 Proof of Cook-Levin Theorem

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* $N = \{1, 2, \dots, n\}$;
- **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

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

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*.

Theorem 14.5 (Gavril's Algorithm)**.**
There is a polynomial-time 2*-approximation algorithm for the optimization problem* MIN-VC*.*

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*.*

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 $c : E \to \mathbb{N}$. The output is a Hamiltonian cycle that minimizes the total cost of the edges in the cycle, if one exists.

A popular variation of this problem is called *Metric-TSP*. In this version of the problem, instead of outputting a Hamiltonian cycle of minimum cost, we output a "tour" that starts and ends at the same vertex and visits every vertex of the graph at least once (so the tour is allowed to visit a vertex more than once). In other words, the output is a list of vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}, v_{i_1}$ such that the vertices are not necessarily unique, all the vertices of the graph appear in the list, any two consecutive vertices in the list form an edge, and the total cost of the edges is minimized.

Theorem 14.9 (2-approximation algorithm for Metric-TSP)**.**
There is a polynomial-time 2*-approximation algorithm for* Metric-TSP*.*

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

$$\begin{aligned}&\mathbf{Pr}[A_1 \cap \cdots \cap A_n] = \\ &\qquad \mathbf{Pr}[A_1] \cdot \mathbf{Pr}[A_2 \mid A_1] \cdot \mathbf{Pr}[A_3 \mid A_1 \cap A_2] \cdots \mathbf{Pr}[A_n \mid A_1 \cap A_2 \cap \cdots \cap A_{n-1}].\end{aligned}$$

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,

$$\mathbf{Pr}[A \mid B] = \frac{\mathbf{Pr}[A] \cdot \mathbf{Pr}[B \mid A]}{\mathbf{Pr}[B]}.$$

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 $\boldsymbol{X} : \Omega \to \mathbb{R}$.

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

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

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

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

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

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

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

Equivalently,

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

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

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

Corollary 15.15 (Linearity of expectation 2)**.**
Let $\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_n$ be random variables, and $c_1, c_2, \ldots, c_n \in \mathbb{R}$ be some constants. Then

$$\mathbf{E}[c_1\boldsymbol{X}_1 + c_2\boldsymbol{X}_2 + \cdots + c_n\boldsymbol{X}_n] = c_1\,\mathbf{E}[\boldsymbol{X}_1] + c_2\,\mathbf{E}[\boldsymbol{X}_2] + \cdots + c_n\,\mathbf{E}[\boldsymbol{X}_n].$$

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

$$\mathbf{E}[\boldsymbol{X}_1 + \boldsymbol{X}_2 + \cdots + \boldsymbol{X}_n] = \mathbf{E}[\boldsymbol{X}_1] + \mathbf{E}[\boldsymbol{X}_2] + \cdots + \mathbf{E}[\boldsymbol{X}_n].$$

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

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

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

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

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

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

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

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

15.2.2 The most fundamental inequality in probability theory

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

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

15.2.3 Three popular random variables

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

Definition 15.23 (Binomial random variable)**.**
Let $\boldsymbol{X} = \boldsymbol{X}_1 + \boldsymbol{X}_2 + \cdots + \boldsymbol{X}_n$, where the $\boldsymbol{X}_i$'s are independent and for all i, $\boldsymbol{X}_i \sim \text{Bernoulli}(p)$. Then we say that $\boldsymbol{X}$ has a *binomial distribution with parameters n and p* (we also say that $\boldsymbol{X}$ is a binomial random variable). We write $\boldsymbol{X} \sim \text{Bin}(n, p)$ to denote this.

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

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

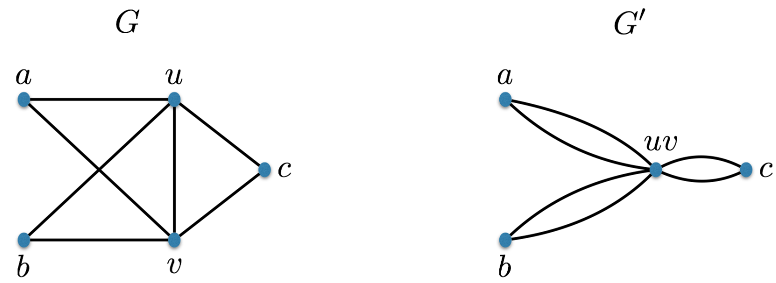
Definition 16.4 (Multi-graph)**.**
A *multi-graph* $G = (V, E)$ is an undirected graph in which E is allowed to be a multi-set. In other words, a multi-graph can have multiple edges between two vertices.[1]

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. $V' = V \backslash \{u, v\} \cup \{uv\}$. 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.*