15-251 Great Ideas in Theoretical Computer Science

Notes by Lómenoirë Mortecc.

Taught by Professor Anil Ada and Professor Pravesh Kothari

A big picture of theoretical computer science, from formalising computation to complexity theory and to various subfields of theoretical computer science, each topic being an introduction to courses that focus on that topic.

Fall 2023

Contents

Chapter 1: Course Introduction

1.1 A Computer Science Perspective

> Computer science is no more about computers than astronomy is about telescopes.

Computer is more than just a machine. Many phenomena have properties of computation, and if we examine them from the lens of computer science, they seem just like computers.

Ponder 1.1.1

A physics analogy

- Theoretical physics — mathematical models (*why is nature mathematical? we don't know*)
- Experimental physical — observe and test theory
- Application / Engineering — use new understanding to solve things

Other analogy…

- Is human brain a computer? Probably
- Evolution in nature
- Chemistry
- …

1.1.1 What's Theoretical Computer Science (TCS)

But first, what is computation?

Definition 1.1.1

- **Computation** — the manipulation of data
- **Algorithm** — how to manipulate data
- **Computational problem** — input-output pairs

A computer performs computation, so it's a box that takes input data and produces output data.

$$\text{input} \to \boxed{\text{computer}} \to \text{output}$$

Like theoretical physics, TCS makes observations about computation in the real world, builds a mathematical model of computation in the abstract world, explores new knowledge in the abstract mathematical world, and eventually applies new knowledge in the real world.

So, TCS starts by defining some **mathematical model of computation**.

1.1.2 Formalising computation

David Hilbert asked some mathematical questions that arguably led to the development of computer science:

> Does there exist a *finite procedure* (later known as algorithm) to determine whether a multivariable polynomial with integer coefficients has an integer solution?
> (one of Hilbert's 23 open problems)

> Is there a *finite procedure* to determine if a mathematical statement is true or false?
> (known as the **Entscheidungsproblem**)

The questions concerned not specific mathematical statements, but the procedure of doing math itself. They turned out fundamental to mathematical reasoning… and that the answers to both questions were no.

But to answer the two questions, mathematicians soon realised they didn't have a formal definition for *algorithm*… until lambda calculus and Tring Machine were proposed. **Turing Machine** captures computation in a simple and convincing way, but lambda calculus was less obvious. It turns out that the two were equivalent. The **Church-Turing** thesis says that any computational problem solvable by physical processes can also be solved by a Turing Machine.

Returning to Hilbert's problems… no Turing Machine can compute those two.

1.1.3 Main Problems in TCS

Like Hilbert's problems, we care about how well something can be computed.

- **Computability**
- **Computational complexity** (aka practical computability)
  - Time
  - Space
  - Randomness as a resource
  - Quantum resources

There are different camps for how to approach computability problems:

- **Algorithm designers** — try to come up with more efficient algorithms
- **Complexity theorists** — prove the lowest possible complexity[1]

Some example problems in problems in TCS

- P vs NP problem
- Does time and space efficiency imply each other?
- Can we solve some problems by randomisation/determinism?
- Fairness in socioeconomics
- Computational game theory
- Learning theory (what's happening with those machine learning models?)
- Quantum computation (how to use quantum to solve problems?)

1.1.4 Math vs Cilantro

> "Mathematics is like… cilantro"
> "There are 5 kinds of people when it comes to cilantro."

1. Don't know what it is
2. Like it
3. It's fine

[1] usually hard!

4. Don't like it
5. Genetic condition that makes cilantro taste like soap

1.2 A Mathematical Perspective

1.2.1 Computers and Proofs

Ponder 1.2.2

There may exist some very hard-to-prove statements. For example:
- $313(x^3 + y^3) = z^3$ has no positive integer solution
- You can't break a solid 3D ball into finite pieces, put them back together, and get 2 identical balls.
- $1 + 1 = 2$

How does one prove it?
- Proof by lack of counterexample? No counterexample after trying many numbers? **No**
- "it's obvious" **Hmmm is it really?**

Proof by computer search? It's doable but does not give insight nor explain the logic behind the proof. But is a requirement for proofs to give insight? Maybe not. We have zero-knowledge proofs and that does not give insight.

1.2.2 From Good Old Regular Mathematics (GORM) to FORM

Traditionally mathematical reasoning takes the form of:

$$\text{axioms} \rightarrow \text{logical reasoning} \rightarrow \text{theorems (new knowledge)}$$

But in the late 1800s, problems start to arise with GORM:

- Can we agree on the same set of axioms to build math on?
- What is obvious truth? Could it depend on environment and interpretation *(think Flatland)*
- How to understand infinity
- Russel's paradox
- Using human language *(ambiguous!)*

So people wanted absolute formalism beyond GORM. We call it **FORM**. FORM needs to be a mathematical model for GORM, but then… what's the right model? It turns out we need computation.

1.2.3 Contributors to FORM and Their Ideas

- **Aristotle** - unambiguity, reasoning, axioms
- **Euclid** - geometry reasoning
- **Leibniz** - calculate what's right
- **Boole** - propositional calculus
- **Cantor** - set theory with infinity
- **Frege** - first-order logic (predicate calculus), axiom for set theory
- **Russel** - "set of all sets that do not contain themselves" — *boom contradiction*
- **Russel + Whitehead** - Principia Mathematica
  - formalizes $1 + 1 = 2$[2]
  - showed that formalisation is possible
- **Hilbert** - Hilbert's programme
  - find precise formal language, manipulate by well-defined rules

  - completeness and consistency
  - algorithm to decide provability
- **Gödel** - Incompleteness theorem
- **Turing** - No algorithm for provability

Despite that…

- We can formalise mathematical proofs
- There is limits to provability
- Birth of computer science
- Computers… can help with formal proofs

Example 1.2.1

Proving Kepler's conjecture (1611): the best way to pack spheres is to put them in a pyramid kind of shape…

2005, Tom Hales at UPitt came up with a 120-page proof (at the same time lots of code to solve optimisation problems)… 20 reviewers couldn't decide its correctness for 4 years and gave up (one died).

In 2015, the proof was formalised and checked by computer. So we can have **proof checkers**!

Example 1.2.2

Robbins conjecture: all Robbins algebras are Boolean algebras

Opened for 63 years, but was proved by computer.

[2] their footnote says "this above proposition is occasionally useful"

Chapter 2: String, Encoding, and Language

Ponder 2.0.1

$$\text{input} \rightarrow \boxed{\text{computer}} \rightarrow \text{output}$$

Computer takes some input information, does some computation, and produces some output information, but how do we formalise the input and output information?

Also, what's information in the first place?

- System with only 1 state $\longrightarrow$ no information
- System with 2 states $\longrightarrow$ 1 bit of information
- System with probabilistic state $\longrightarrow$ hmm

Maybe has to do with the number of states...

Definition 2.0.1

Information is the **number of possible states**, and its unit is **bits**.

2.1 Mathematical Representation of Data

2.1.1 Characters and Strings

Definition 2.1.2

- **Alphabets** Σ = non-empty, finite set of symbols
- **Symbol / Character** = elements of some Σ
- **String / Word** = sequence of elements from some Σ
  - $s = a_1 a_2 a_3 ... a_n$ where $a_i \in \Sigma$
  - $s = \varepsilon$ for empty string
  - $|s|$ for length of string
- Σ^k = set of all length-k strings over Σ
- Σ^* = set of all finite-length strings over Σ

Example 2.1.1

Alphabets

- English $\Sigma = \{a, b, c, d, ..., z\}$
- Greek $\Sigma = \{\alpha, \beta, \gamma, \delta, ..., \omega\}$
- Binary $\Sigma = \{0, 1\}$

Sigma stars

- $\{0, 1\}* = \{\varepsilon, 0, 1, 00, 01, 10, 11, ...\}$
- $\{a\}* = \{\varepsilon, a, aa, aaa, ...\}$

Fact 2.1.1

Σ^* is countably infinite.

2.1.2 String Operations

Let $w = a_1 a_2 ... a_n \in \Sigma^*$, $u, v \in \Sigma^*$ be strings.

- The **reversal** of w is $w^R = a_n a_{n-1} ... a_1$
- The **concatenation** of u and v is written $u \cdot v = uv$
- The k-th **power** of u is $u^k = \underbrace{uuu...u}_{k \text{ times}}$
- A **substring** of w is some s such that $w = xsz$
- A **prefix** of w is some p such that $w = px$
  - A **proper prefix** would require $x \neq \varepsilon$
- A **suffix** of w is some s such that $w = xs$
  - A **proper suffix** would require $x \neq \varepsilon$

2.1.3 String Encoding

We have a few motivations for defining encoding:

1. Every object should have some encoding
2. Encoding should be unique for each concept, viz. no to concepts map to the same encoding
3. Not every string has to be valid encoding for some concept

This corresponds to the properties of an injective function.

Definition 2.1.3

An **encoding** for $A \in \Sigma^*$ is an injective function $\text{Enc} : A \hookrightarrow \Sigma^*$. For $a \in A$, write $\text{Enc}(a) = \langle a \rangle$ to denote the encoding for a.

A **valid encoding** w is a sting $w \in \Sigma^*$ such that $\exists a \in A, w = \langle a \rangle$

If an encoding function exists for A, we say A is **encodable**.

Fact 2.1.2

Countability $\Leftrightarrow$ Encodability[3]

So not all sets are encodable.[4]

[3] will come back to this

[4] for uncountable sets... we can still encode by approximation. An example is floating point for the uncountable $\mathbb{R}$.

Example 2.1.2

Example encodings

1. $A = \mathbb{N}, a = 36 \in \mathbb{N}$
   $\langle 36 \rangle =$ "36" (decimal)
   $=$ "100100" (binary)
   $=$ "111111111111111111111111111111111111" (unary)
2. $A = \mathbb{Z}, a = -36 \in \mathbb{Z}$
   $\langle 36 \rangle =$ "36" $(\Sigma = \{-, 0, 1, ..., 9\})$
   $=$ etc.
3. $A = \mathbb{N} \times \mathbb{N}, a = \langle (3, 36) \rangle = \langle 3, 36 \rangle$

   Idea: use additional # symbol so $\langle 3, 36 \rangle =$ "3#36"
   Then, this can get encoded into binary by encoding $\Sigma^* = \{\#, 0, ..., 9\}$
4. $A = \{\text{All undirected graph}\}$

   Ideas:
   - Adjacency matrix
   - Listing vertices and edges
5. $A = \{\text{All python function}\}$

   Idea: just write the string

But does $|\ \Sigma\ |$ matter? We can always encode all $a \in \Sigma, |\Sigma| = k$ with $t = \lceil \log_2 k \rceil$ bits using some binary Σ' with $|\Sigma'| = 2$.

Unary encoding is also possible, except it would be counting 1's. Converting binary string to unary string:

```
 0 ->    1
 1 ->   11
01 ->  111
10 -> 1111
etc.
```

So we can change Σ, and that changes our encoded length. Let $n = |\Sigma|$, then $a \in \Sigma$ can be encoded by length:

$$\begin{cases} n \text{ if in unary} \\ \lfloor \log_2 n \rfloor + 1 \text{ if in binary} \\ \lfloor \log_k n \rfloor + 1 \text{ if in base } k \end{cases}$$

Observe that unary encoding is exponentially longer than other bases.

2.2 Language

Definition 2.2.4

A **language** L using alphabets from Σ is a subset of all possible finite-length strings $L \subseteq \Sigma^*$.

The **size** of a language L is simply $|L|$.

Notation: write ALL to denote the set of all languages $\mathcal{P}(\Sigma^*)$.

2.2.1 Operations on Languages

Let $L, L_1, L_2 \subseteq \Sigma^*$ be a language,

- **Reversal:** $L^R = \{w^R \in \Sigma^* \mid w \in L\}$, in which w^R is w in reverse.
- **Concatenation:** $L_1 L_2 = \{uv \in \Sigma^* \mid u \in L_1 \wedge v \in L_2\}$.
- **Power:** for some $n \in \mathbb{N}$, $L^n = \{u_1 u_2 \cdots u_n \mid u_{1..n} \in L\}$.
- **Star operation:** $L^* = \bigcup_{n \in \mathbb{N}} L^n$. Note ε is always in L^*. Intuitively, this is 0-or-more concatenation.
- **Plus operation:** $L^+ = \bigcup_{n \in \mathbb{N}^+} L^n$. Intuitively, this is 1-or-more concatenation.

2.3 Computation as Function on Strings

For now, we work with a few assumptions:

- **Determinism:** computer produces same output given same input
- Input can be any **finite-length** string
- Computation is a **finite** process viz. it terminates
- There's some output for every input (viz. perhaps some error string output for bad input)
- Output is **finite-length** string

Then, the computer acts like $f : \Sigma^* \to \Sigma^*$. But this alone doesn't tell us *how to go from input to output.* Nonetheless, it models a form of computational problem. A computer solves the problem if its input-output pairs matches that of f.

Example 2.3.3

- Reverse problem $101100 \mapsto 010011$
- Sort problem $101100 \mapsto 000111$
- Decision problem $11111010 \mapsto 0$

More generally, computational problems can be thought of as $f : I \to S$, in which I is an instance of input, and S is the solution. But in TCS, we can usually encode both I and S with some string in Σ^*.

Definition 2.3.5

A **function problem** is a problem with the form:

$$\begin{array}{ccc} f : I & \to & S \\ \downarrow & & \downarrow \\ \text{Enc} & & \text{Enc'} \\ \downarrow & & \downarrow \\ f' : \Sigma^* & \to & \Sigma^* \end{array}$$

A **decision problem** is a subset of function problems with the form

$$f : \Sigma^* \to \{0, 1\}$$

Note 2.3.1

For CTS purposes, **string** is the only type of data.

A special type of such problem is **decision problem**. It's a restricted function with binary output, so $f : \Sigma^* \to \{0, 1\}$. It's a simpler mathematical object to work with, with a one-to-one correspondence with language—simply the subset of inputs that map to true. That is, there is a bijection between $\mathcal{P}(\Sigma^*)$ and the set of decision problems, namely:

$$f(x) = \begin{cases} 1 \text{ if } x \in L \\ 0 \text{ if } x \notin L \end{cases} \quad \text{for all } x \in \Sigma^*$$

Interestingly, it is almost always true that all functional problems have a corresponding decision problem with the same computability.

Example 2.3.4

Integer factorisation problem.

- The function problem: $N \in \mathbb{N} \mapsto$ prime factors of N
- The decision problem: $N, k \in \mathbb{N} \mapsto$ whether N has factor x s.t. $2 \leq x \leq k$

Given the latter, one can construct an algorithm for the former by:

```
for i in 2..n:
  if hasPrimeFactor(n, 2, i):
    record i as a prime factor
    n' divide n by i
    return {find all prime factors for n' and record them} union {i}
```

Chapter 3: Deterministic Finite Automata

We want a good model of computation, but let's first see how far one can go with a simple and restricted model of computation.

3.1 Terminology

To be able to define computation, we first build models for computation.

Definition 3.1.1

Some terminology

- **Computational model** — set of rules allowed for information processing
- **Machine** — an instance of a computational model. This could be a physical realisation or mathematical representation. We usually work with the latter in TCS.
- **Universal Machine / Programme** — a programme that can run many other programmes, such as a laptop.

For now, $\text{Machine} = \text{Computer} = \text{Programme} = \text{Algorithm}$

For this chapter, we also assume:
1. There's no universal machine, viz. each machine does one thing
2. We only consider decision problems
   - the machine either accepts or rejects the input string
   - focus only on functional problems with some corresponding decision problem

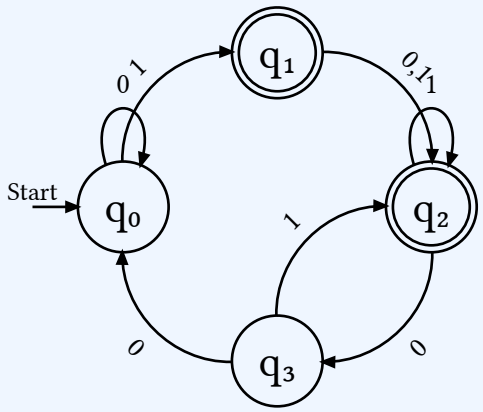
3.2 Defining Deterministic Finite Automata (DFA)

This is a *restricted* model of computation. Often in the real world we do have restrictions. At the same time, designing a model that is as simple as possible can bring interesting properties.

Restrictions of DFA:
- Only one pass over the input string
- Very limited memory

Notation:
- Each node represents a state, usually indicated by q_i for some i
- Double circles represent accept state, otherwise it's a reject state
- Initial state q_0 has arrow coming from nowhere
- Transition rules are arrow with alphabet corresponding to the transition
- To declutter, it's acceptable to say that missing arrow all goes to reject, etc.

Example 3.2.1



3.2.1 Language and DFA

Definition 3.2.2

Let M be a DFA and $L \subseteq \Sigma^*$. We say that M solves L if:

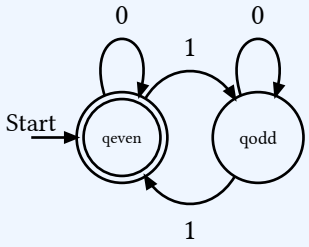
- for all $w \in L$, M accepts w
- for all $w \notin L$, M rejects w

We denote the language for which all strings a DFA M accepts by $L(M)$

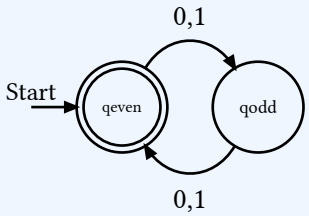
Example 3.2.2

Example languages and DFAs

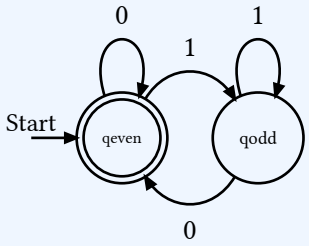
1. $L(M) =$ strings with even number of 1's



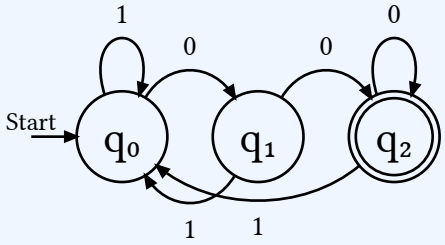
2. $L(M) =$ strings with even number of characters



3. $L(M) =$ strings that end in 0's and the empty string



4. $L(M) =$ strings that end with 00



Notice we can do various things like:
- Maybe build a DFA for some language
- Translate DFA into some programme in code
- Get the complement of the language by flipping accept and reject states

3.2.2 Formal Definition of DFA

We want a few properties:
- Every state need to have $|\Sigma|$ transitions out of it
- There is no direction for states (so you can go back to a previously visited state)
- DFA needs to tell you the output for every input

Definition 3.2.3

A DFA is $M = (Q, \Sigma, \delta, q_0, F)$, in which
- Q is the set of states
- Σ is the alphabet the DFA operates on
- δ is the transition function $\delta : Q \times \Sigma \to Q$. It takes the current state, the characters being read, and outputs the next state.
- q_0 is the start state
- $F \subseteq Q$ is the set of accept states

Let $q \in Q, w \in \Sigma^*$, we write $\delta^*(q, w)$ to indicate the state after running on w, viz.

$$\delta^*(q, w) = \delta(...\delta(\delta(q, w_1), w_2), ...)$$

M **accepts** w if $\delta^*(q_0, w) \in F$, otherwise M **rejects** w.

A good way to capture the input and output of δ is perhaps a table.

Definition 3.2.4

Computationpath for DFA Given a DFA $M = (Q, \Sigma, \delta, q_0, F)$ as previously defined, the **computational path** of M on input $w \in \Sigma^*$ is a sequence of states $r_0, r_1, ..., r_n$ ($r_i \in Q$) that the DFA visits as it reads w. Note this means $r_0 = q_0$.

3.2.3 Applications of DFAs

DFA is always linear time, so if we can efficiently build a DFA that solves some decision problem, we get $O(n)$ for free. An example is deciding whether w contains some substring s.

3.3 Regular and Non-Regular Languages

Definition 3.3.5

We define regular languages REG to be the set of all languages that can be solved by some DFA.

$$\text{regular languages } \mathsf{REG} \subseteq \text{all languages } \mathsf{ALL}$$

But is it true in the other direction? Probably not. DFAs have limited memory and only scans in one direction (viz. forgets what it reads), but some language may require more memory to solve. We want to find the simplest counterexample to show that there exists non-regular language. One idea is to make use of the fact that DFAs are bad at counting—because $|Q|$ is finite, they can only keep track of $|Q|$ distinct situations.

Key limitations of DFAs:
- Finite number of states ⟶ limited memory
- Only reads input in one direction ⟶ can't go back to check something if not in memory

Strategy 3.3.1

Proving a language L is not regular using the **Pigeonhole Principle** (PHP).

1. Assume that the language is regular and there is a DFA with $k \in \mathbb{N}^+$ states that solves L.
2. Come up with a **fooling pigeon set** P such that $|P| > k$
3. By **PHP**, $\exists x, y \in P$ with $x \neq y$ must end up at the same state in the DFA, so xz and yz for some $z \in \Sigma^*$ end up in the same state
4. Pick z such that $xz \in F$ xor $yz \in F$. Contradiction.

Note this strategy also works if one wants to prove a lower bound on the number of states a DFA needs to solve a certain language—assume a DFA with fewer states solves it, and derive a contradiction.

Theorem 3.3.1

$L = \{0^n 1^n \mid n \in \mathbb{N}\}$ is not regular.

Proof.

AFSOC L is regular and there exists DFA $M = (Q, \Sigma, \delta, q_0, F)$ that solves L with $|Q| = k, k \in \mathbb{N}^+$.

Consider $P = \{0^i \mid 0 \leq i < k + 1\}$. Then by PHP:

$$\exists x = 0^n, y = 0^m \in P, x \neq y, \delta^*(q_0, x) = \delta^*(q_0, y)$$

But then let $z = 1^n$. So $xz \in L$ but $yz \notin L$. But $\delta^*(q_0, xz) = \delta^*(q_0, yz)$. Contradiction.

□

3.3.1 Closure Properties of Regular Languages

Regular languages are closed under:

- Complement
- Union
- Intersection
- Concatenation
- Star operation

For the proofs below, let $M^1 = (Q^1, \Sigma, \delta^1, q_0^1, F^1)$ solve L_1 and $M^2 = (Q^2, \Sigma, \delta^2, q_0^2, F^2)$ solve L_2.

Theorem 3.3.2

Regular languages are closed under complement

$$L_1 \subseteq \Sigma^* \text{ is regular} \Rightarrow \overline{L_1} = \Sigma^* \setminus L_1 \text{ is regular}$$

Proof.

$M' = (Q^1, \Sigma, \delta^1, q_0^1, Q^1 \setminus F^1)$ solves $\overline{L_1}$. □

Theorem 3.3.3

Regular languages are closed under union

$$L_1, L_2 \subseteq \Sigma^* \text{ are regular} \Rightarrow L_1 \cup L_2 \text{ is regular}$$

Proof idea: construct new DFA with state $Q' = Q^1 \times Q^2$, initial state $q_{0'} = (q_0^1, q_0^2)$ step both DFAs and accept if one of the final states is accept for one of the DFAs viz. $F' = \{(q_1, q_2) \mid q_1 \in F^1 \vee q_2 \in F^2\}$.

Theorem 3.3.4

Regular languages are closed under intersection

$$L_1, L_2 \subseteq \Sigma^* \text{ are regular} \Rightarrow L_1 \cap L_2 \text{ is regular}$$

Proof idea 1: Write $L_1 \cap L_2$ as $\overline{\overline{L_1} \cup \overline{L_2}}$

Proof idea 2: very similar to union, but with $F' = \{(q_1, q_2) \mid q_1 \in F^1 \wedge q_2 \in F^2\}$.

Theorem 3.3.5

Regular languages are closed under concatination

$$L_1, L_2 \subseteq \Sigma^* \text{ are regular} \Rightarrow L_1 L_2 \text{ is regular}$$

Proof.

Define M' by:

$$\begin{aligned}
Q' &= Q^1 \times \mathcal{P}(Q^2) \\
\Sigma' &= \Sigma \\
{q_0}' &= \begin{cases} (q_0^1, \emptyset) \text{ if } q_0^1 \notin F^1 \\ (q_0^1, \{q_0^2\}) \text{ if } q_0^1 \in F^1 \end{cases} \\
\delta'(a, B) &= \begin{cases} (\delta^1(a,\sigma), \{\delta^2(b,\sigma) \mid b \in B\}) \text{ if } \delta^1(a,\sigma) \notin F^1 \\ (\delta^1(a,\sigma), \{\delta^2(b,\sigma) \mid b \in B\} \cup \{q_0^2\}) \text{ if } \delta^1(a,\sigma) \in F^1 \end{cases} \\
F' &= \{(a, B) \in Q' \mid \exists b \in B, b \in F^2\}
\end{aligned}$$

□

Definition 3.3.6

Notice it's useful to have a set of states and step the entire set, so we define a **generalised transition function** $\delta_{\mathcal{P}} : \mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$ by:

$$\delta_{\mathcal{P}}(S, \sigma) = \{\delta(q, \sigma) \mid q \in S\}$$

Theorem 3.3.6

Regular languages are closed under star operation

$$L_1 \subseteq \Sigma^* \text{ are regular} \Rightarrow L_1^* \text{ is regular}$$

Proof.

We can solve L_1^+ by constructing M':

$$\begin{aligned} Q' &= \mathcal{P}(Q^1) \\ \Sigma' &= \Sigma \\ {q_0}' &= \{q_0^1\} \\ \delta'(S) &= \begin{cases} \delta_{\mathcal{P}}^1(S,\sigma) \cup \{q_0^1\} \text{ if } \delta_{\mathcal{P}}^1(q,\sigma) \cap F^1 \neq \emptyset \\ \delta_{\mathcal{P}}^1(S,\sigma) \text{ else} \end{cases} \\ F' &= \{S \subseteq Q^1 \mid S \cap F^1 \neq \emptyset\} \end{aligned}$$

Then we can use closure under union to construction M'' for $L_1^+ \cup \{\varepsilon\}$ □

3.3.2 Recursive Definition of Regular Language

It turns out the definition simple language is equivalent to that of regular language. Namely:

Theorem 3.3.7

A regular language can be recursively defined as:
- $\emptyset$ is regular
- $\{\sigma\}$ is regular for all $\sigma \in \Sigma$
- If L_1, L_2 are regular, then $L_1 \cup L_2$ is regular
- If L_1, L_2 are regular, then $L_1 L_2$ is regular
- If L_1 is regular, then L_1^* is regular

Proof omitted.

Chapter 4: Turing Machines (TMs)

DFA with tape — the model for computation!

4.1 Preamble

Recall what we want from a model of computation:
1. As simple as possible
2. As general as possible

Observations:
1. Computational devices need to be a finite object. Finite algorithms need to solve arbitrary-length input.
2. It has unlimited memory. That is, it can access more working memory if needed.

> An algorithm is a finite answer to infinite number of questions

4.1.1 Coming Up with TMs (Modern Perspective)

Ponder 4.1.1

Attempt 1: Define computable by what Python can compute!

Problems:
- But what is Python? 100 pages of definition?
- Why Python? What's special about Python among the other programming languages

This is general enough, but not simple

> We want a totally minimal (TM) programming language

Ponder 4.1.2

Attempt 2: Upgrade DFA + Tape — enable write to tape and allow moving around tape.

```
STATE 0:
  switch sigma:
    case 'a':
      write 'b';
      move LEFT;
      goto STATE 2;
    case 'b':
      ...
    ...
...
```

That seems to work!

To represent a TM, we can draw state diagrams similar to those for DFAs. Just put the character to write and the direction to move on the transition.

4.1.2 Coming Up with TMs (Turing's Perspective)

Ponder 4.1.3

Recall that in Hilbert's time, mathematicians were trying to formalise what it means to have a finite procedure, such as in the Entscheidungsproblem. Computers back then were humans, and so a finite procedure would be instructions given to humans so that they could write proofs mechanically.

But then we needed to define what the instruction looks like and what instructions a human can follow and execute.

A TM, then, must capture human computers' workflow. Observe that **a human computer reads and writes symbols on paper**.

- Human has finite mental states, thus finite number of states in a TM.
- Human computers are deterministic, so kind of like DFAs with deterministic state changes.
- Human knows finite number of symbols, and they can have a set of working symbols that is a superset of the input symbols.
- Human can work with papers with square cells and put one symbol in one cell.
- Human can always grab more papers.
- Human can work by reading/writing at one location at a time. (WLOG, if someone really needs to read/write more cells at a time, we can just have larger cells with a larger set of composite symbols)
- There is nothing special about 2D papers, so moving Left/Right is sufficient for computation.

So then if we define a TM to model a computer with finite number of states, finite number of symbols, and an infinite tape, they can perform equivalent computation as a human computer. Simple and general!

4.1.3 Key Differences Between DFAs and TMs

DFAs

- Multiple accept/reject states
- Halt at string end
- Always terminates
- Finite cell access

TMs

- One accept state, one reject state
- Halt on accept/reject
- May loop forever
- Can read/write infinite number of cells

4.2 Formal Definition of TMs

Definition 4.2.1

A turing machine can be described by a mighty 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$

- Q is the set of states
- Σ is the set of input characters, such that $\sqcup \notin \Sigma$
- Γ is the set of tape characters, satisfying $\Sigma \subset \Gamma \wedge \sqcup \in \Gamma$
- $\delta : (Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ the transition function
- $q_0 \in Q$ is the initial state
- $q_{\text{acc}} \in Q$ is the accept state
- $q_{\text{rej}} \in Q$ is the reject state, it must be that $q_{\text{rej}} \neq q_{\text{acc}}$

The TM then operates on an infinite tape, with infinite blank symbols surrounding the input string and the tape head initially at the start of the input string.

M can then be thought of as a function $\Sigma^* \to \{0, 1, \infty\}$, with 0 being reject, 1 being accept, and ∞ being loop forever.

Note 4.2.1

One can also set up a TM with a tape that's infinite only in one direction, but this does not make the TM any less powerful.

Definition 4.2.2

Configuration of a TM

A **configuration** of a TM consists of:
1. The content on the tape
2. Its state
2. The location of the tape head

We can specify the configuration formally by writing down $uqv \in (\Gamma \cup Q)^*$ where $u, v \in \Gamma^*$ and $q \in Q$. The convention is that the tape head is at the first character of v, and the state is q. We say that a configuration is **accepting** if $q = q_{\text{acc}}$ and rejecting if $q = q_{\text{rej}}$.

Definition 4.2.3

Language of a TM

$$L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$$

Note that M may loop forever for some input, but $L(M)$ only contains those that M accepts.

4.3 Universality of Computation

The idea is that a TM can perform any computation we want just with a finite set of instructions and an infinite scratchpad.

4.3.1 TM Subroutines, Tricks, and Description

- Move Left/Right until hitting $\sqcup$
- Shift entire input by one cell to Left/Right

- Convert $\sqcup x_1 x_2 x_3 \sqcup$ to $\sqcup x_1 \sqcup x_2 \sqcup x_3 \sqcup$
- Simulate Γ with $\{0, 1, \sqcup\}$ viz. alphabet reduction
- Mark cells with $\Gamma' = \{0, 1, 0^{\cdot}, 1^{\cdot}, \sqcup\}$
- Copy paste tape segment
- Simulate 2 TMs on the same tape
- Implement some data structure
- Simulate RAM by reading address and moving
- ...
- Simulate assembly

So **a Turing Machine can simulate all our programmes written in other programming languages**. Therefore, to describe a TM, it is sometimes sufficient to describe it at a higher level, knowing that it can compile down to a 7-tuple describing an actual TM.

Levels of description:

- **Low** - state diagram, 7-tuple
- **Mid** - movement, behaviour
- **High** - pseudocode

4.3.2 Universal Machines

Definition 4.3.4

A universal machine is a machine that can simulate any machine.

It follows that Turing's TM can simulate any TM, just like how human takes instructions and input, a TM can take the blueprint of another TM, some input, and simulate the input string on the input TM.

An important realisaion is that **code is data**, so any algorithm/programme can be encoded and fed into some other machine. The good thing is that we get universal machines, the maybe not so good thing is that we now have to deal with self reference.

4.4 The Church-Turing Thesis

Is computation equivalent to any physical process?

Definition 4.4.5

The Church-Turing Thesis states that any computation allowed by the laws of physics can be done by a TM.

Definition 4.4.6

The Church-Turing Thesis^{+} states that any computational problem solvable by physical process is solvable by a probabilistic TM.

Definition 4.4.7

The Church-Turing Thesis^{++} states that any computational problem efficiently solvable by physical process can be efficiently solved by a Quantum TM.

Note that the Church-Turing Thesis is not a theorem, but something we believe to be true. At least we don't know of any counterexample yet.

4.5 Decidability

Definition 4.5.8

A **decidable language** is a language a TM can solve. Let $L \subseteq \Sigma^*$, L is dicidable if there exists M a TM such that:

- $\forall w \in L$, M halts and accepts w
- $\forall w \notin L$, M halts and rejects w

In which case we call M a **decider** for L.

We write R to indicate the set of all decidable languages.

Definition 4.5.9

A **decider** TM is a TM that never loops.

Example 4.5.1

Deciding $\text{isPrime} = \{\langle n \rangle \mid n \text{ is a prime}\}$

One can simply write some code to do it:

Proof.

```
def M(n: int):
  if n < 2:
    return 0
  for i in [2, 3, ..., n - 1]:
    if n % i == 0:
      return 0
  return 1
```

Since we can write code to solve it, we can compile it to a TM that solves it, so it is decidable.

□

Example 4.5.2

Deciding behaviour of DFAs

- $\text{ACCEPTS}_{\text{DFA}} = \{\langle D : \text{DFA}, x : \text{str}\rangle \mid D \text{ accepts } x\}$ is decidable because we can just simulate the DFA and decide
- $\text{SELF-ACCEPTS}_{\text{DFA}} = \{\langle D : \text{DFA}\rangle \mid D \text{ accepts } \langle D\rangle\}$ decidable, same as above
- $\text{SAT}_{\text{DFA}} = \{\langle D : \text{DFA}\rangle \mid D \text{ is satisfiable}\}$ decidable, we can do a graph search to see if any of the accept states are reachable
- $\text{NEQ}_{\text{DFA}} = \{\langle D_1 : \text{DFA}, D_2 : \text{DFA}\rangle \mid L(D_1) \neq L(D_2)\}$ decidable... but a bit more tricky.

Proving that NEQ_{DFA} is decidable by reduction.

Proof. First, we observe that this is equivalent to deciding if $(L(D_1) \cup L(D_2)) \setminus (L(D_1) \cap L(D_2))$ is empty.

But we can rewrite that as $\left(L(D_1) \cap \overline{L(D_2)}\right) \cup \left(\overline{L(D_1)} \cap L(D_2)\right)$. By closure properties of regular languages, we can build a DFA to solve that using D_1 and D_2, and we can use a decider for SAT_{DFA} to figure out if that region is empty.

We just reduced NEQ_{DFA} to SAT_{DFA}. We write $\text{NEQ}_{\text{DFA}} \leq \text{SAT}_{\text{DFA}}$.

□

4.5.1 Closure Properties of Decidable Languages

Theorem 4.5.1

Decidable language is closed under complement.

Proof.

Let L be a decidable language. Let M de a decider for L. We can build a decider for $\overline{L}$ by:

```
fn x => (case M x of 1 => 0 | 0 => 1)
```

□

Theorem 4.5.2

Decidable language is closed under union.

Proof. Let L_1, L_2 be decidable languages. Let `M1`, `M2` be their deciders. We can build a decider for $L_1 \cup L_2$ by:

```
fn x => (case M1 x of 1 => 1 | 0 => M2 x)
```

□

Theorem 4.5.3

Decidable language is closed under intersection.

Proof. Let L_1, L_2 be decidable languages. Let `M1`, `M2` be their deciders. We can build a decider for $L_1 \cup L_2$ by:

```
fn x => (case (M1 x, M2 x) of (1, 1) => 1 | _ => 0)
```

□

4.5.2 Semi-Decidability

Decidability requires that the decider always outputs 0 or 1, but we may relax the requirement and find that some languages are only semi-decidable.

Definition 4.5.10

A TM M **semi-decides** L if:

$$\forall w \in \Sigma^*, w \in L \iff M(w) = 1$$

viz.

- $w \in L \Rightarrow M(w) = 1$
- $w \notin L \Rightarrow M(w) \in \{0, \infty\}$

Which means our TM always says yes if the input is in the language, but may say no xor loop forever if it's not.

A language is **semi-decidable** if there exists a semi-decider for it.

We write RE for the set of all semidecidable languages.

Corollary 4.5.1

All decidable languages are semi-decidable.

Corollary 4.5.2

So REG $\subseteq R \subseteq$ RE $\subseteq$ ALL.

Theorem 4.5.4

A language L is decidable iff both L and $\overline{L}$ are semi-decidable.

Proof idea: for the forward direction, use complement closure property. For the other direction, construct a decider using the two semi-deciders (hint: step each TM in incremental number of steps until one of them halts).

Chapter 5: Limits of Computation

Galileo
$S = \{0, 1, 4, ...\}$
So $|\mathbb{N}| > |S|$?
$R = \left\{\sqrt{0}, \sqrt{1}, \sqrt{4}, ...\right\}$
but...
$|S| = |SR| = |\mathbb{N}|$
Maybe size comparison on ∞ is undefined.

Gauss
"The notion of a completed infinity doesn't belong in mathematics"

Cantor
Infinity as a first class citizen
(To Galileo: so close, why not say they *are* equal?)

Turing
TM is finite, that is a limit to computation

Gödel
Limit to mathematical reasoning

5.1 Limits of Counting

5.1.1 Cantor and Infinite Sets

To deal with size comparison on infinite sets, Cantor proposed to define relative cardinality using injections.

Definition 5.1.1

Let A and B be sets, define[5]:

$$|A| = |B| \iff \exists \text{ bijection } f : A \leftrightarrow B$$
$$|A| \leq |B| \iff \exists \text{ injection } f : A \hookrightarrow B \iff \exists \text{ surjection } g : B \twoheadrightarrow A$$
$$|A| \geq |B| \iff \exists \text{ surjection } f : A \twoheadrightarrow B \iff \exists \text{ injection } g : B \hookrightarrow A$$

Notice we don't need to define $|A|$ nor $|B|$ in isolation to define a partial order on $|A|$ and $|B|$. Also notice that cardinality comparison defined this way forms a valid partial order relation (reflexive, not symmetric, antisymmetric, transitive).

Corollary 5.1.1

If $A \subseteq B$, then $|A| \leq |B|$. We can just inject by identity function.

[5]Hmmm something may break if $\emptyset$ is involved, but in that case... meh whatever

Theorem 5.1.1

$$|\Sigma^*| = |\mathbb{N}|$$

Proof.

Define $f : \Sigma^* \twoheadrightarrow \mathbb{N}$ by $f(s) = |s|$. That's surjective so $|\Sigma^*| \geq |\mathbb{N}|$.

Now list words in Σ^* by length-lexicographical order, so every word eventually appears. One can then define surjection $g : \mathbb{N} \twoheadrightarrow \Sigma^*$ by $g(n) = n\text{-th item in the list}$. That's surjective so $|\Sigma^*| \leq |\mathbb{N}|$.

□

Theorem 5.1.2

If S is infinite and $|S| \leq |\mathbb{N}|$, then $|S| = |\mathbb{N}|$.

Proof.

Haha whatever. Fill this in later[6].
◝ (◡ ▿ ◟) ◜

□

5.1.2 Countable Sets

It seems then, there are three types of cardinality:
1. Finite
2. Infinite and same as $|\mathbb{N}|$
3. Infinite but greater than $|\mathbb{N}|$

Since we can enumerate both 1 and 2, we usually call them countable, so the remaining 3 is uncountable.

Definition 5.1.2

- A is **countable** if $|A| \leq |\mathbb{N}|$
- A is **countably infinite** if it's countable and infinite
- A is **uncountable** if $|A| > |\mathbb{N}|$

But recall that $|\mathbb{N}| = |\Sigma^*|$, we can redefine countability in terms of Σ^*. But then recall that some set is encodable if it can be injected into Σ^*. Voila countability is the same as encodability!

Definition 5.1.3

- A is **countable** if A is encodable viz. $\exists\ \text{Enc} : A \hookrightarrow \Sigma^*$
- A is **uncountable** if A is not encodable viz. $\nexists\ \text{Enc} : A \hookrightarrow \Sigma^*$

[6] 3 months later: what the heck is this?

Strategy 5.1.1

Countability heuristics

1. If one can list the elements of a set A, then there is some injection into $\mathbb{N}$ so it's probably countable.
2. If one can write down each $a \in A$ in a unique way using some $w \in \Sigma^*$, then it's probably encodable and thus countable.

Example 5.1.1

Known countable sets

- $\mathbb{N}$
- $\mathbb{Z}$
- $\mathbb{Z} \times \mathbb{Z}$ — can list in a spiral shape on the $\mathbb{Z} \times \mathbb{Z}$ plane, but probably easier encode tuples by something like $\langle (14, 25) \rangle = 14\#25 \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \#\}^*$
- $\mathbb{Q}$ — likewise, use $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, /\}$

5.1.3 Diagonalisation

Motivation: given a set $\mathcal{F}$, can we construct something that's not in the set? If so, perhaps we can also prove some set is not countable.

The $\mathcal{F}$ of interest could be:

- The set of all decidable languages
- The set of all languages decidable in $O(n^2)$
- The set of all provable math statements

For now, we try to treat $\mathcal{F}$ as a set of functions, and many sets can actually be represented as functions.

Strategy 5.1.2

Representing sets as functions

- Take set A. We can have that $f(a) \iff a \in A$
- Let $r \in [0, 1)$, we can represent $r = 0.f(0)f(1)f(2)...$

Lemma 5.1.1

Diagonalisation Lemma

Let there be a set of functions $\mathcal{F} = \{f : X \to Y\}$. If $|X| \geq |\mathcal{F}|$ and $|Y| \geq 2$, we can construct $f_D : X \to Y$ such that $f_D \notin \mathcal{F}$.

Strategy 5.1.3

Diagonalising against $\mathcal{F}$ to get a **diagonal element** f_D by:

1. Define some injection $\phi : \mathcal{F} \to X$
2. Define f_D such that $\forall f \in \mathcal{F}, f_D(\phi(f)) \neq f(\phi(f))$

Corollary 5.1.2

Let $\mathcal{F}(X)$ denote the set of all functions $f : X \to Y$ ($|Y| \geq 2$).

Then $|X| < |\mathcal{F}|$ because otherwise we can diagonalise and get some f_D that's not in $\mathcal{F}(X)$... but we said $\mathcal{F}$ is the set of all functions $f : X \to Y$.

So then $|\mathbb{N}| < |\mathcal{F}(\mathbb{N})|$ so $\mathcal{F}(\mathbb{N})$ is uncountable.

Likewise $\mathcal{F}(\Sigma^*)$ would be uncountable. Pick $Y = \{0, 1\}$, then the set of all decision problems is uncountable.

Theorem 5.1.3

Cantor's Theorem

For all set A, $|A| < |\mathcal{P}(A)|$.

Proof.

Consider $\mathcal{F}(A) = \{f : A \to \{0, 1\}\}$. Then each $f \in \mathcal{F}$ is a characteristic function for some subset of A. But meanwhile $|A| < |\mathcal{F}(A)| = |\mathcal{P}(A)|$.

□

But... but... the set of all Turing Machines is countable and the set of all decision problems is uncountable... we may be onto something... there must be undecidable problems!

Observe that many undecidable problems are not encodable, we may want to eventually find an explicit, encodable undecidable problem. Those are the more interesting **finitely decidable problems**.

5.1.4 Uncountable Sets

Definition 5.1.4

Σ^∞ is the set of all infinite strings over some finite alphabet Σ

Theorem 5.1.4

$\{0, 1\}^\infty$ is uncountable

To prove this, one way is to establish some correspondence between $\{0, 1\}^\infty$ and $\mathcal{P}(\mathbb{N})$.

Strategy 5.1.4

Strategies to prove a set A is uncountable

1. AFSOC A is countable, diagonalise against A and derive a contradiction.
2. Inject a known uncountable set into A, or surject A onto a known uncountable set.

5.2 Limits of Computation

Ponder 5.2.1

Consider the following languages, are they decidable?

- $\text{ACC}_{\text{TM}} = \{\langle M : \text{TM}, x : \text{str}\rangle \mid M(x) = 1\}$
- $\text{SELF-ACC}_{\text{TM}} = \{\langle M : \text{TM}\rangle \mid M(\langle M\rangle) = 1\}$
- $\text{HALTS}_{\text{TM}} = \{\langle M : \text{TM}, x : \text{str}\rangle \mid M(x) \in \{0, 1\}\}$
- $\text{SAT}_{\text{TM}} = \{\langle M : \text{TM}\rangle \mid L(M) \neq \emptyset\}$
- $\text{NEQ}_{\text{TM}} = \{\langle M_1 : \text{TM}, M_2 : \text{TM}\rangle \mid L(M_1) \neq L(M_2)\}$

5.2.1 Diagonalising Turing Machines

Let $\mathcal{F}$ be the set of all TMs so $\mathcal{F} = \{\langle M\rangle \mid M : \Sigma^* \to \{0, 1, \infty\}\}$. We need $|\Sigma^*| \geq |\mathcal{F}|$ to diagonalise, that's indeed true because TMs are encodable.

now we define f_D by:

$$\forall M_i \in \mathcal{F}, f_D(\langle M_i\rangle) = \begin{cases} 0 \text{ if } M_i(\langle M_i\rangle) = 1 \\ 1 \text{ if } M_i(\langle M_i\rangle) = 0 \\ 1 \text{ if } M_i(\langle M_i\rangle) = \infty \end{cases}$$

$$\forall x \in \Sigma^* \setminus \mathcal{F}, f_D(x) = 1 \text{ (whatever)}$$

So we just constructed f_D corresponding to the language

$$\text{NOT-SELF-ACC}_{\text{TM}} = \{\langle M : \text{TM}\rangle \mid M(\langle M\rangle) \neq 1\}$$

Theorem 5.2.5

$\text{NOT-SELF-ACC}_{\text{TM}}$ is undecidable.

Proof.

As above □

5.2.2 Proving Undecidability by Reduction

Theorem 5.2.6

$\text{SELF-ACC}_{\text{TM}}$ is undecidable.

Proof.

AFSOC suppose it is, then we can flip the output of a decider to solve $\text{NOT-SELF-ACC}_{\text{TM}}$. Contradiction. □

Theorem 5.2.7

ACC_{TM} is undecidable.

Proof.

AFSOC suppose it is, then we can use it to decide $\text{SELF-ACC}_{\text{TM}}$ by passing in the encoding of the TM of interest. Contradiction. □

Theorem 5.2.8

HALT_{TM} is undecidable.

Proof.

AFSOC suppose it is with some decider `M_H`, then we can use it to decide ACC_{TM} by

```
fn (M: TM, x: str) => (case M_H (M, x) of 0 => 0 | _ => M x)
```

Contradiction. □

Definition 5.2.5

Decision problem A reduces to decision problem B if, assuming there exists a decider TM M_B for B, we can construct a decider M_A for A. We write $A \leq B$ to denote "A reduces to B".

Corollary 5.2.3

Suppose $A \leq B$, then:
- B decidable $\Rightarrow$ A decidable
- A undecidable $\Rightarrow$ B undecidable

Example 5.2.2

Prove that SAT_{TM} is undecidable.

Proof.

AFSOC there exists TM `M_S` that decides SAT_{TM}. Now define `M_HALTS` by:

```
fun M_HALTS (M: TM, x: str) =
  let
    fun M' y = let val _ = M x in 1 end
  in
    M_S M'
  end
```

If `M` halts on `x`, `M'` accepts any `y`, so `M_S` accepts so `M_HALTS` accepts.

If `M` does not halt on `x`, `M'` never accepts anything on any input, so `M_S` rejects so `M_HALTS` rejects.

We decided the undecidable. Contradiction.

□

Note the idea of reduction can be formalised as Turing reduction. Which involves Oracle TMs etc., but not having that does not stop us from proving undecidability.

5.2.3 More Undecidable Languages

Theorem 5.2.9

$\text{SAT}_{\text{TM}} = \{\langle M : \text{TM}\rangle \mid L(M) \neq \emptyset\}$ is undecidable

Proof.

Suppose `M_SAT` decides SAT_{TM}, we can decide $\text{ACCEPTS}_{\text{TM}}$ by:

```
fun M_ACCEPTS (M: TM, x: str) =
  let
    fun M' y = M x
  in
    M_SAT M'
  end
```

□

Theorem 5.2.10

$\text{NEQ}_{\text{TM}} = \{\langle M_1 : \text{TM}, M_2 : \text{TM}\rangle \mid L(M_1) \neq L(M_2)\}$ is undecidable

Proof.

Suppose `M_NEQ` decides NEQ_{TM}, we can decide SAT_{TM} by:

```
fun M_SAT (M: TM) =
  let
    fun M_EMPTY y = 0
  in
    M_NEQ (M_EMPTY, M)
  end
```

□

Theorem 5.2.11

$\text{FINITE}_{\text{TM}} = \{\langle M : \text{TM}\rangle \mid L(M) \text{ is finite}\}$ is undecidable.

Proof.

Suppose `M_FINITE` decides $\text{FINITE}_{\text{TM}}$, we can decide HALT_{TM} by:

```
fun M_HALT (M: TM, x: str) =
  let
    fun M' y =
      if y = "meh" then 1
      else let
        val _ = M x
      in
        1
      end
  in
    (case M_FINITE M' of 1 => 0 | 0 => 1)
  end
```

□

Theorem 5.2.12

$\text{SAT}_{\text{TM}} \leq \text{HALTS}_{\text{TM}}$.

Idea: build a TM that, in increasing number of maximum steps and input length, tries all strings within the length bound. Deciding if it halts tells us whether or not it's satisfyable.

5.2.4 Consequences of Undecidability

- No way to write general programmes that verifies other programmes
- No way to decide if bruteforce search for counterexample to math conjecture halts
- By Church-Turing Thesis, ... physical universe can't compute everything

Other undecidable problems that don't deal with TMs:

- Entscheidungsproblem
- Hilbert's 10th problem
- Mortal matrices: is there a way to multiply 21×21 integer matrices U and V some number of times in some order to get the zero matrix?

5.2.5 Non-Semi-Decidable Decidable Languages

Theorem 5.2.13

If L es semi-decidable and undecidable $\Rightarrow \overline{L}$ is not semi-decidable.

Idea: otherwise L would be decidable.

Some non-semi-decidable languages to know. Notice we know the complement of these are semi-decidable but not decidable.

- $\overline{\text{SELF-ACCEPT}_{\text{TM}}}$

- $\overline{\text{ACCEPTS}_{\text{TM}}}$
- $\overline{\text{HALTS}_{\text{TM}}}$

5.3 Limits of Human Reasoning: Gödel's Incompleteness Theorems

What we wanted: precise model for axioms, deduction rules, mathematical, statements, and proofs

- **Axioms** – obvious truth
- **Statement** – well formed, has truth value
- **Deduction rule** – how to go from known truth to new truth
- **Proof** – chain of deduction from axioms to statement

The hope is that truth $\equiv$ provable... but this isn't how things turn out.

5.3.1 FORM Essentials

GORM can be viewed as computation. We at least want the verifier to be decidable, as we always want to know if a proof is valid or not.

$$\text{statement } S \to \boxed{\text{prover}} \to \text{SOME proof } P \text{ / NONE}$$

$$\text{statement } S \to \boxed{\text{decider prover}} \to \text{accept if provable, else reject}$$

$$(\text{statement } S, \text{proof } P) \to \boxed{\text{verifier}} \to \text{accept if proof works, else reject}$$

Formally, we want:

- $\forall$ valid statement S in GORM, $\exists \langle S \rangle \in \Sigma^*$ in FORM
- $\forall$ proof P in GORM, $\exists \langle P \rangle \in \Sigma^*$ in FORM
- $\exists$ verifier TM V such that $V(\langle S \rangle, \langle P \rangle)$ accepts $\Leftrightarrow$ P proves S
- S provable $\Leftrightarrow \exists w \in \Sigma^*, V(\langle S \rangle, w)$ accepts

This means, we can build inefficient, semidecider provers that search through Σ^* for proof.

```
prover (<S>):
    for all strings w in length-lex order:
        if V(<S>, w) accepts, return w

isProvable(<S>):
    for all strings w in length-lex order:
        if V(<S>, w) accepts, accept
        if V(<negation S>, w) accepts, reject
```

To proceed, we need the **GORM-to-ZFC** (Zermelo-Fraenkle-Choice) thesis, which says that the ZFC axiomatic system is a right system for GORM. All proofs and statements in GORM compile to ZFC, much like algorithms compile to TMs.

5.3.2 Proving Things About FORM

Things we want:

- **Consistency** – at most one of S, $\neg S$ provable
- **Soundness** – S provable $\Rightarrow$ S is true (this implies S false $\Rightarrow$ S not provable and consistency)
- **Completeness** – $\forall S$, at least one of S, $\neg S$ provable

Then soundness would imply `isProvable` is always correct (not necessarily terminates), and completeness implies `isProvable` always halts.

So

$$\text{consistency} \wedge \text{completeness} \Rightarrow \text{exactly one of } S, \neg S \text{ provable}$$
$$\text{soundness} \wedge \text{completeness} \Rightarrow \text{truth} \equiv \text{provability}$$

5.3.3 Incompleteness Theorems

WTS: soundness $\wedge$ completeness $\Rightarrow$ incomputable things are computable

Set up:

- Assume ZFC captures GORM
- Assume verifier exists
- Say statement S is independent if neigher S nor $\neg S$ provable. Notice the existence of independent statement implies incompleteness

Theorem 5.3.14

0th incompleteness theorem[7]: maybe we can't reason about all of mathematics.

It makes sense that the set of all mathematical concepts is infinite, but only so much is finitely describable. Mathematicians work with those that are finitely describable, so maybe we can't work with a lot of math.

Theorem 5.3.15

1st incompleteness theorem (soundness)

AFSOC ZFC is sound and complete. So `isProvable` is a decider, so `isProvable` $\equiv$ `isTrue`. Now we can decide HALT by:

```
M_HALTS(M, x) = isTrue "M(x) halts"
```

Then ZFC cannot be both sound and complete. So ZFC soundness $\Rightarrow$ incompleteness.

[7] really just a heuristic

Theorem 5.3.16

1st incompleteness theorem (soundness with explicitly unprovable S)

Consider NOT-SELF-ACCEPT (NSA):

```
M_NSA(M) = isTrue "M does not self-accept"
```

Since NSA is undecidable, `M_NSA` must not decide it, so there exists some TM such that `isTrue "M does not self-accept"` doesn't return right answer, in which case the statement "M does not self-accept" is independent.

We can diagonalise to get such M, and it turns out to be M_{NSA}. If we feed $S =$ "$\langle M_{\text{NSA}} \rangle$ does not self-accept" into M_{NSA}, it doesn't give the correct answer. We found an S that's independent.

Theorem 5.3.17

1st incompleteness theorem (consistency)

Maybe try relaxing soundness to consistency.

Let I = "$\langle M_{\text{NSA}} \rangle$ does not self-accept"

But then if ZFC is consistent, I is independent...

Definition 5.3.6

Statement S reduces to statement T if "$T \Rightarrow S$" is provable.

Corollary 5.3.4

Assuming S reduces to T:
- If T provable, S provable
- If S unprovable, T unprovable.

Theorem 5.3.18

2nd incompleteness theorem

Recall the unprovable I. We showed that "ZFC is consistent" $\Rightarrow I$. But then I reduces to "ZFC is consistent".

So "ZFC is consistent" is also not provable.

Chapter 6: Time Complexity

Here is where we go from computability to practical computability. We will be interested in how efficiently we can compute something.

Resources we have access to:

- time
- memory
- randomness
- quantum

Applications:

- simulations
- optimisation
- AI
- security, cryptography

Questions to answer:

- how to define complexity?
- what's the right level of abstraction?
- how to analyse complexity?

Ponder 6.0.1

Gödel's 1956 letter to von Neumann—an early formalisation of complexity

Let F be math statement. We brute force search for proof with max length length n.

Let $\Psi(F, n)$ be number of steps needed to search
Let $\varphi(n) = \max_F \Psi(F, n)$

How fast does $\varphi(n)$ grow? If the proof is too long we might not even want to think about F.

Essentially, what's the asymptotic behaviour of $\varphi(n)$?

6.1 Analysing Complexity

Given function A. There are many ways to measure complexity of A

Worse case number of steps on inputs with length n.

$$\max_{x,\ |x|\ =n} \text{number of steps } A(x) \text{ takes}$$

Asymptotic big-O bound
- abstracts away many detail
- sharp enough to compare algorithms

Polynomial-time concerns if it is in $O\left(n^k\right)$ for some k, which is more abstract than a big-O bound.
- Categorieses polytime as efficient, non-polytime as inefficient
- Polytime could come from exploiting some structure in the problem, whereas non-polytime tends to involve brute force
- Robust: polytime categorisation of problems is consistent across machines

- Closed: using polytime algorithm inside polytime algorithm is still polytime
- Qualitative difference between in polytime and not in polytime

Note 6.1.1

In TCS, we always input length as the notion of input size. Given input x, $n = |x|$ is the number of symbols in x.

Example 6.1.1

Primality test by trying divider from 2 to input integer N.

Notice $n = |N| = \lg N$, so number of iterations is in $O(N) = O(2^n)$

6.2 Asymptotics

Definition 6.2.1

Given $f, g : \mathbb{R}^+ \to \mathbb{R}^+$, $f(n) \in O(g(n))$ iff $\exists c > 0, n_0 > 0$ such that $\forall n \geq n_0, f(n) \leq cg(n)$.

Fact 6.2.1

$\forall m > 0, \lg m < m$

Theorem 6.2.1

$\forall \varepsilon > 0, k > 0, \lg^k n \in O(n^\varepsilon)$, viz. polylog is always in poly.

Proof sketch: $\lg\left(n^{\frac{\varepsilon}{k}}\right) < n^{\frac{\varepsilon}{k}}$, rearrange, take k-th power, and set $c = \left(\frac{k}{\varepsilon}\right)^k$

Fact 6.2.2

$\forall b > 1, \log_b n \in \Theta(\lg n)$ i.e. base doesn't matter for log.

Definition 6.2.2

Some names

- Polynomial $O\left(n^k\right)$ with constant $k > 0$
- Exponential $O\left(2^{n^k}\right)$ with constant $k > 0$

Definition 6.2.3

P refers to the set of polytime decidable languages

Theorem 6.2.2

Composing polytime algorithms is still polytime.

Definition 6.2.4

Intrinsic complexity is the complexity of the most efficient possible algorithm for some problem.

Strategy 6.2.1

Sometimes this is not well defined. Sometimes this is hard to find. Sometimes one can bound this by contradiction (suppose an algorithm computes this in less than some $T(n)$, then the algorithm cannot be right) or by its output size (since writing output takes at least length of output).

6.3 Complexity Model

Time complexity may be different for one-tape TM, reg machine, and our computers... so to get something closer to real situation we use the **Random Access Machine (RAM)** model. It's like a machine with random access memory. The machine can also operate on short numbers efficiently.

Definition 6.3.5

Short vs Long numbers

Short numbers are defined to be those with length in $O(\lg n) = O(\lg \lg N)$ where n is length of input. These can be thought to fit in a register. Equivalently, a number x is small if $x \in O(n^k)$ for constant k.
Long numbers are those that are not short. When operating on long numbers, one has to work with them as strings.

Arithmetic operations `+ - * / > <` on short numbers	1 step
Memory access `A[0xffe67a41]`	1 step

6.4 Long Number Operations

Example 6.4.2

Integer addition

Consider `fn B: int => B + B` with $n = |B| = \lg B$.

This has to be done by string manipulation, since $|B| \notin O(\lg n)$. We go through the bits from right to left to add while pushing carry onto the next bit, so $O(n)$.

Example 6.4.3

Integer multiplication

Consider `fn (A: int) (B: int) => A * B`. We can multiply `A` by every bit in `B` and add things up (scaled appropriately by some power of the base). This is $O(|A||B|)$.

We can try divide and conquer. WLOG say $|A| = |B| = n$. Then break each into 2 pieces, $A = ab, B = cd$, so $AB = \left(a \cdot 10^{\frac{n}{2}} + b\right)\left(c \cdot 10^{\frac{n}{2}} + d\right) = ac \cdot 10^n + (ad + bc) \cdot 10^{\frac{n}{2}} + bd$ but this is still $O(n^2)$...

Maybe there's a better way (Karatsuba algorithm). Observe

$$(a+b)(c+d) = ac + ad + bc + bd$$
$$\Rightarrow ad + bc = (a+b)(c+d) - ac - bd$$

And we already have ac, bd, so we only need 3 recursive multiplication.

So $O\left(n^{\lg 3}\right)$

Fact 6.4.3

There is a $O(n \lg n)$ algorithm for integer multiplication

Example 6.4.4

Matrix multiplication

Naive dot products: $O(n^3)$.

Strassen's algorithm by divide and conquer and addition trick to save one recursive call $O\left(n^{\lg 7}\right)$.

Current world record $O(n^{2.37155})$; whether we can get $O(n^2)$ is open problem.

Chapter 7: Graph Theory

Ponder 7.0.1

Many problems can be modelled in the form of graphs

Example 7.0.1

- Facebook (friendship graph)
- Enemybook (enemyship graph)
- Zachary Karate Club (from anthropology paper "An Information Flow Model for Conflict and Fission in Small Groups")
- Google page rank (links as directed edges)
- Google map
- Kidney exchange (compatibility search, pairing donations[8])

7.1 Undirected Graphs

Definition 7.1.1

A **simple undirected graph** G is a tuple V, E where

- V is a finite set of edges
- E is a set of edges represented as sets of size-2 subsets of V

Note 7.1.1

Edge cases in definition

- $E = \emptyset$ is fine, with every vertex being orphan
- $V = \emptyset$ is a null graph? Maybe not very useful

The convention is to denote $n = |V|, m = |E|$.

7.1.1 Neighbourhood

Definition 7.1.2

Let $u, v \in V$, u and v are **neighbours** if $\{u, v\} \in E$.

The **neighbourhood function** maps vertex to its set of neightbours $N(v) = \{u \in V, \{u, v\} \in E\}$.

Degree of v is $\deg(v) = |N(v)|$

A ***d*-regular** graph is graph $G = (V, E)$ such that $\forall v \in V, \deg(v) = d$.

[8] US national kidney exchange programme uses some algorithm by some CMU prof

Lemma 7.1.1

The handshake lemma

Given graph $G = (V, E)$, we have

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

The proof is by double counting. Put tokens on edges next to each vertex and:
1. Each edge has 2 tokens, thus RHS
2. Each vertex v has $\deg(v)$ tokens, thus LHS

Example 7.1.2

Does there exist 5-regular graph with 251 vertices?

No. By handshake lemma $\sum_{v \in V} \deg(v) = 2m \Rightarrow 5n = 5 \cdot 251 = 2m \Rightarrow m \notin \mathbb{N}$.

7.1.2 Walks and Paths

Definition 7.1.3

A **walk** in G is a sequence $u_1, ..., u_k$ such that $\forall i \in \{2, ..., k\}, \{u_{i-1}, u_i\} \in E$. Its length is the number of edges in the sequence

A **path** is a walk without repeated vertices, viz. $u_1, ..., u_k$ distinct.

The **distance** between $u, v \in V$ is the length of the shortest path between u, v, denoted $\text{dist}(u, v)$. If no such path exists, the distance is $\text{dist}(u, v) = \infty$.

A **circuit** in G is a walk from some $u \in V$ to $u \in V$.

A **cycle** is a circuit without repeated vertices (with length ≥ 3).

An **acyclic** graph has no cycle.

7.1.3 Connectedness

Definition 7.1.4

A graph is **connected** if $\forall u, v \in V, \exists$ a path between u and v.

A **connected component** (CC) is a connected part of a graph.

Theorem 7.1.1

At least G connected $\Rightarrow m \geq n - 1$.

Proof. We take out all edges and put them back in one by one.

Each edge we add can be:

1. A **connector** that connects distinct components, so number of CCs goes down by 1 but no cycle is introduced.
2. A **cycle creator** that connects vertices in the same CC, creating a cycle but leaving number of CCs unchanged.

We must go from n CCs to 1 to connect the graph, so we must use at least $n - 1$ connectors. If we add $m > n - 1$ edges, we must introduce a cycle creator at some point. □

Theorem 7.1.2

At least G connected $\wedge\ m = n - 1 \Leftrightarrow G$ acyclic.

($\Rightarrow$) by adding edges one by one argument and in the process we never create a cycle.
($\Leftarrow$) go by contradiction, using the cases when adding edges one by one.

Fact 7.1.1

$n - 1$ edges are sufficient to connect a graph

7.2 Trees

Definition 7.2.5

An ***n*-vertex tree** is a graph that satisfies the following:

1. Connected
2. $m = n - 1$
3. Acyclic

Fact 7.2.2

Any two of the above automatically imply the third.

Definition 7.2.6

Given a tree $T = (V, E)$ and $v \in V$, v is

- a **leaf** if $\deg(v) = 1$
- an **internal node** if $\deg(v) \neq 1$

Lemma 7.2.2

A tree with $n \geq 2$ has at least 2 leaves.

Proof: try using handshake lemma to get contradiction

$$1 + 2n \leq \sum_{v \in V} \deg(v) = 2m = 2(n-1)$$

Theorem 7.2.3

A tree T with l leaves and max degree Δ must have $\Delta \leq L$.

Ideas:
1. Use handshake lemma and directly derive an inequality. Realise that the degree of leaves are 1 and assume for the worse case every internal node has degree Δ.
2. Induct on number of vertices. Use $n \leq 3$ for base case. For inductive step remove some leaf and apply IH. Case on Δ' after leaf removal to finish the proof.
3. Delete a vertex in the graph with degree Δ to get a forest of Δ trees. Case on the number of leaves (at least 1 or at least 2) for each tree. Observe that each tree in the forest must contain one leaf for the original graph.

7.3 Mininum Spanning Tree (MST)

Definition 7.3.7

A tree is a **minimum spanning tree (MST)** if it's a subgraph tree connecting all vertices with minimised total edge weight.

WLOG we can assume edge weights are distinct. In fact unique edge cost implies unique MST.

Theorem 7.3.4

MST Cut Property (aka light edge property)

Given graph $G = (V, E)$ and non-empty subset $S \subset V$, the cheapest edge connecting S and $V \setminus S$ must be in the MST.

Algorithm 7.3.1

Jarník-Prim Algorithm returns MST by:

1. Initialise $X = \{a\}$ for some random $a \in V$; $T = \{\}$
2. While $X \neq V$, find min cost edge out of T and add it to T, add the new vertex to X

The correctness follows directly from MST cut property. The cost is in $O(mn)$. This algorithm works even if there exist negative weight edges. It's possible to get $O(m \lg m)$ with a better implementation.

Fact 7.3.3

Pettie & Ramachandran 2002 proved an algorithm is optimal for MST, but we don't know the big-O bound.

7.4 Directed Graph

Definition 7.4.8

A **directed graph** G is (V, E) such that

- V is set of vertices similar to undirected graph
- E contains size-2 tuples (u, v) of vertices representing edges from u to v

Definition 7.4.9

Neighbours in a directed graph can be **in-neightbour** and **out-neightbour**.

- a **in-neightbour** of u is v such that $(v, u) \in E$
- a **out-neightbour** of u is v such that $(u, v) \in E$

A **sink** is a vertex without out-neightbour; a **source** is a vertex without in-neightbour

7.5 Graph Search

7.5.1 Any-First Search (AFS)

Generaic graph search algorithm. It keeps track of some visited set and traverses the graph via neighbours

Algorithm 7.5.2

Any-First Search

```
fun AFS (G = (V, E)) s =
  let
    fun AFS' X F =
      if F = {} then X else
      let
        v = pick vertex from  F
        X = X union {v}
        F = (F union N(v)) \ X
      in
        AFS' X F
      end
  in
    AFS {} {s}
  end
```

Sometimes we want to traverse every vertex, even if the graph is not connected. We can do

```
AFSAll = iterate (AFS G) V
```

Graph traversal by AFS induces some tree, with the root at the source node s and edges tracing the path through which every vertex is first visited.

7.5.2 Breadth-First Search (BFS)

Algorithm 7.5.3

Same as AFS, with `F` acting like a queue.

The run time is $O(m)$ for a single CC, or $O(m+n)$ for the whole graph.

Observe that BFS-tree has these properties:

1. Shortest path from s to some vertex t can be found by t's depth in the BFS tree.
2. Edges in the original graph but not the BFS tree can only connect vertices more than 1 layer away in the BFS tree.

7.5.3 Depth-First Search (DFS)

Algorithm 7.5.4

Same as AFS, with `F` acting like a stack.

Likewise run time is $O(m)$ for a single CC, or $O(m+n)$ for the whole graph.

Observe that BFS-tree has these properties:

1. Edges in the original graph but not the DFS tree can only go between descendent and ancestor. This means no cross edge.

7.6 Graph Matching

Ponder 7.6.2

We often want to match things... such as

$$\begin{aligned} \text{machines} &\leftrightarrow \text{jobs} \\ \text{professors} &\leftrightarrow \text{courses} \\ \text{room} &\leftrightarrow \text{courses} \\ \text{student} &\leftrightarrow \text{internships}^{9} \end{aligned}$$

How to solve? Graph!

Definition 7.6.10

A **matching** in graph G is some $M \subseteq E$ s.t. no two edges in M share endpoint. The size of a matching is $|M|$.

A **perfect matching** matches every vertex.

7.6.1 Maximum Matching

Definition 7.6.11

A **maximum matching** is a matching such that it has the maximum possible size among all possible matchings. This can be thought of as a global optimal.

A **maximal matching** is a matching such that no more edge can be added to the current matching to form a bigger matching. This can be thought of as a local optimal.

The **maximum matching**, then, is the problem of finding a maximum matching. A trivial brute force solution exists but is in $\Omega(2^m)$. Greedy algorithm doesn't seem to work. Turns out we can have an algorithm based on path analysis.

Definition 7.6.12

Give some matchin M for G, a path p in G is **alternating** w.r.t M if the edges alternate between $\in M$ and $\notin M$.

An alternating path is an **augmenting path** if the endpoints of the path are not part of M.

Observe that the existence of an augmenting path means we can have a better matching. We can at least match one more pair by switching the matching status of each edge in the augmenting path.

[9] sadly a difficult one

Lemma 7.6.3

If every vertex in G has max degree 2, then every CC in G is either a path or a cycle.

Proof. By inducting on number of vertices. For G with max degree 2, pick a vertex v to remove. After removal the graph G' still has max degree 2, so G' has only disjoint paths or cycles. We case on $\deg(v)$ when adding it back:

- If $\deg(v) = 0$, then v is isolated so it's a length-0 path on its own
- If $\deg(v) = 1$, then the vertex v is connected to must have degree 1 in G', which is an endpoint of a path, so adding v back in extends that path.
- If $\deg(v) = 2$, then removing v breaks a cycle to create a path. Adding it back in recreates the cycle.

□

Theorem 7.6.5

$$\nexists \text{ augmenting path} \Leftrightarrow M \text{ is maximum}$$

Proof. Show instead $\exists$ augmenting path $\Leftrightarrow M$ not maximum

($\Rightarrow$) Trivial. Take the augmenting path and improve matching.

($\Leftarrow$) If M not maximum, $\exists$ maximum matching $M^*, |M^*| > |M|$.

Consider $S = M^* \triangle M = (M^* \cup M) \setminus (M^* \cap M)$ as in figure.

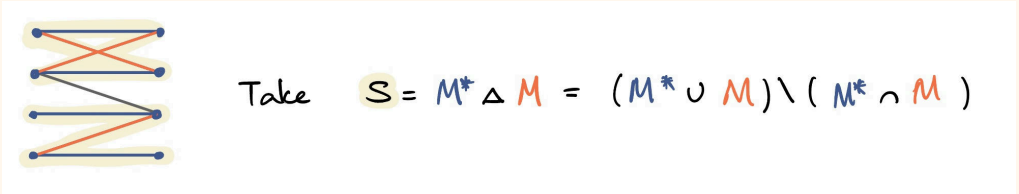


Figure 7.6.1

Observe $\forall$ vertex v involved in S, $\deg(v) \in \{1, 2\}$ (otherwise it should contradict M and M^* being matchings), so S is a set of disjoint cycles and paths by lemma.

Also, each path or cycle in S must alternate between being in M and in M^*. Since cycles have even lengths, one alternating path must exist, call it $p \subseteq S$. Then some such p must have more edges from M^* than M in order for $|M^*| > |M|$ to hold. This is an augmenting path.

□

Algorithm 7.6.5

Finding maximum matching

Start with some matching M. While there is augmenting path, find it and augment the matching. Return final matching.

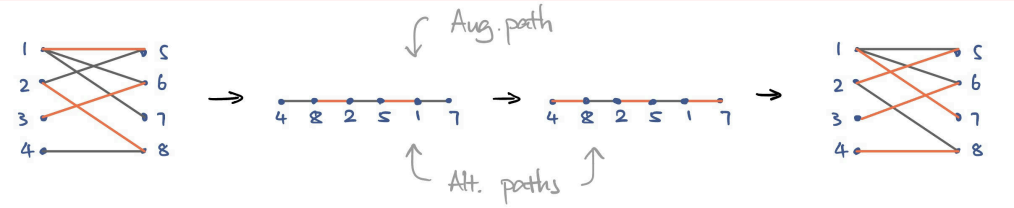


Figure 7.6.2

Correctness: this must terminate since we add one edge every iteration. When it terminates we must have no more augmenting path left so it must return a maximum matching.

But then we need to be able to find augmenting path. Here's an algorithm.

Algorithm 7.6.6

Algorithm for finding augmenting path (bipartite graph case)

Given graph $G = (V, E)$, partitions X, Y, and matching M.

1. Point unmatched edges from X to Y
2. Point matched edges from Y to X
3. For each unmatched vertex $x \in X$, run DFS starting from x to find some unmatched $y \in Y$. If found, retrace the search path; it's an augmenting path. If nothing found, no augmenting path.

This is polytime by inspection.

Lemma 7.6.4

A tree has at most one perfect matching.

Proof. (Sketch) Suppose we have two perfect matchings M and M'. Take their symmetric difference S. There must not exist cycle in S since it's a tree, so S has a path p with edges alternating between in M and M'. But that means one of M and M' fail to match one of the endpoints of p, contradiction on them being perfect matching. □

7.6.2 Bipartite Graphs and k-Colourability

Definition 7.6.13

A graph G is bipartite if $\exists U \subseteq V, \forall e = \{u, v\} \in E, u \in U \Leftrightarrow v \in V \setminus U$.

We sometimes explicitly write bipartite graph by $G = (X, Y, E)$ where $X = U, Y = V \setminus U$.

Definition 7.6.14

A graph is k-colourable if you can colour each vertex with one of k colours such that $\forall e = \{u, v\} \in E, u$ and v have different colour.

Fact 7.6.4

$$\text{bipartite} \equiv \text{2-colourable}$$

by inspection.

Theorem 7.6.6

$$\text{no oll-length cycle} \Leftrightarrow \text{2-colourable}$$

Proof. ($\Rightarrow$) Trivial. Suppose there exists odd-length cycle. Then go along the cycle to put things in the right partition. This would always lead to the first vertex having to be in both partition, that's bad.

($\Leftarrow$) By construction. WOLG assume graph connected and we 2-colour by BFS. Simply colour with alternating colour for each level.

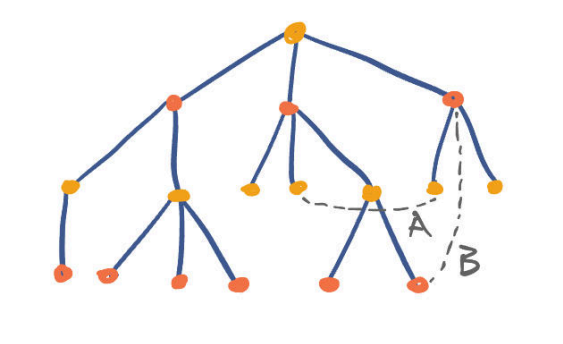


Figure 7.6.3

Observe:

- edges connecting vertices at the same level (like A) cannot happen because that implies odd-length cycle.
- edges that skip even number of levels (like B) cannot happen because BFS tree cannot have those.

Then all non-tree edges go between consecutive cycles, which is correctly coloured. Tree-edges are also correctly coloured by construction. □

7.7 Stable Matching

We often have situations involving a 2-sided market. There are 2 types of participants, each with their own preference. If we assume all entities in the market behave rationally, an interesting is whether there always exists a stable matching. This is a Nobel prize problem in Economics. Also interesting is that the existence of stable matching was proved by designing an algorithm.

Example 7.7.3

Types of matchings in real world:

- residents $\leftrightarrow$ hospitals
- students $\leftrightarrow$ internships

In each situation, each side has preference rankings for which entity on the other side they want to get matched to.

Informally, we can see a matching could be unstable if there exists some unmatched pair with endpoints who prefer each other more than their current match. In this case, they have an incentive to deviate from the matching—not something we want if we want a single system to decide matchings.

Definition 7.7.15

Formally, a **stable matching problem** involves two sets (X, Y) with $|X| = |Y| = n$, each vertex has a preference with vertices in the other set, which are complete total orderings. The goal is to find a perfect matching between X and Y without unstable pairs.

An **unstable pair** is defined to be some unmatched $(x, y) \in A \times B$ such that x, y prefer each other more than their current match.

7.7.1 Gale-Shapley Algorithm

It can be shown that a stable matching always exists by presenting an algorithm.

Algorithm 7.7.7

The **Gale-Shapley** algorithm

Given a stable matching problem with sets (X, Y), we find stable matching by:

- while $\exists$ unmatched $x \in X$: x gives offer to the $y \in Y$ such that y is highest in x's preference list and x has not given offer to y.
- for y in Y, if it gets an offer to x, case on:
  1. If y is currently unmatched: accept and form the pair (x, y)
  2. If y prefers x over current match x', accept x's offer, and reject x'
  3. If y prefers its current match x' over x, decline x's offer
- return the matching

Theorem 7.7.7

Gale-Shapley always returns a stable matching.

Proof.

First, by inspection, we run a maximum of $O(n^2)$ iterations, so the algorithm always terminates.

Next, it must return a perfect matching. AFSOC otherwise, then there exists some unmatched $x \in X$, so x must have been rejected by all $y \in Y$. Observe that $y \in Y$ never become unmatched once matched. But that means x gave offer to every $y \in Y$, so every $y \in Y$ is matched. That can only happen in a perfect matching. Contradiction.

Finally we want to show there is no unstable pair. Observe that $x \in X$ only does down their preference list, and $y \in Y$ only goes up their preference list. Consider every unstable (x, y). There are two cases:

1. x never gave y an offer, then x prefers its current match y' over y
2. x gave y offer, then the only reason x not currently matched to y is because y rejected x at some point, so y prefers its current match x' over x.

Therefore we don't have any unstable pair.

□

Definition 7.7.16

W $x \in X$ is a **valid partner** with some $y \in Y$ if (x, y) can be a matched pair in some stable matching. The **best valid partner** of x is the valid partner on x's list ranked the highest, and the **worst valid partner** of x is the valid partner on x's list ranked the lowest. We denote these two $\text{best}(x)$ and $\text{worst}(x)$ respectively.

Note WLOG these definitions work in either direction.

Definition 7.7.17

A stable matching is ***X*-optimal** if the matching is $\{(x, \text{best}(x)) \mid x \in X\}$, and is ***X*-pessimal** if the matching is $\{(x, \text{worst}(x)) \mid x \in X\}$.

Theorem 7.7.8

Gale-Shapley is X-optimal.

Proof.

AFSOC some $x \in X$ didn't get matched to $\text{best}(x)$. Consider the first time x gets rejected by a valid partner y.

Suppose y rejected x because it preferred x'. Then in another stable matching $x' \leftrightarrow y'$, $x \leftrightarrow y$. Then x' prefers y' over y. So x got rejected by valid partner y' before x got rejected by y, we fixed y to be the first thing to reject x. Contradiction. □

Theorem 7.7.9

Gale-Shapley is Y-optimal.

Ponder 7.7.3

If we use Gale-Shapley, is there an incentive for $x \in X$ or $y \in Y$ to lie?

- for $x \in X$, no because they get matched to their best valid partner
- for $y \in Y$... probably

Ponder 7.7.4

Is there a better algorithm that is not biased to one side?

Theorem 7.7.10

Roth's theorem: no matter what matching algorithm we use, there always exists a slide with incentive to lie

Chapter 8: P vs NP

Beyond computability, we examine practical computability—given a problem, is there an algorithm to compute it *efficiently*?

Some terms:

- **tractable** — efficiently decidable
- **intractable** — not efficiently decidable

8.1 Polynomial-Time Reduction

8.1.1 Some Problems of Interest

***k*-Colouring Problem:** Given graph $G = (V, E)$, decide whether G is k-colourable

$$k\text{COL} = \{\langle G \rangle \mid G \text{ is } k\text{-colourable}\}$$

Clique Problem: Given graph $G = (V, E)$, decide if G contains a clique of size k. A clique is a subset of vertices $S \subseteq V$ that are neighbours of all other vertices in the clique viz. $\forall u, v \in S, u \neq v \Rightarrow \{u, v\} \in E$.

$$\text{CLIQUE} = \{\langle G, k \in \mathbb{N}^+ \rangle \mid G \text{ contains a } k\text{-clique}\}$$

Independent Set Problem: Given a graph $G = (V, E)$, decide if G contains an independent set of size k. An independent set is a subset of vertices such that there is no edge between any two vertices in the subset.

$$\text{IND-EST} = \{\langle G, k \in \mathbb{N}^+ \rangle \mid G \text{ contains a size } k \text{ independent set}\}$$

CNF Satisfiability Problem: Given a logic formula φ in CNF[10] form with boolean variables $x_1, ..., x_n$, does there exist to these boolean variables such that φ is true?

$$\text{SAT} = \{\langle \varphi : \text{CNF} \rangle \mid \varphi \text{ is satisfiable}\}$$

Some variants include constraints on the number of literals in each clause

$$3\text{SAT} = \{\langle \varphi : \text{CNF in which each clause has exactly 3 literals} \rangle \mid \varphi \text{ is satisfiable}\}$$

Boolean Circuit Satisfiability: (informally) Given a circuit represented as an acyclic graph, in which node is a gate—one of AND, OR, NOT, a constant voltage—0 or 1, one of the n input nodes, or an output gate, connected to a reasonable number of in/out wires aka edges, is there some input combination that makes the output 1?

$$\text{CIRCUIT-SAT} = \{\langle C : \text{valid circuit} \rangle \mid C \text{ is satisfiable}\}$$

Other interesting problems that we won't formally define:

- **Gödel's Bounded Entscheidungsproblem**—given statement S, decide if there exists proof of S with length at most k
- **Subset sum**—given $X \subseteq \mathbb{Z}$ does there exist subset $A \subseteq X$ s.t. $\sum_{a \in A} a = k$?

[10] conjeunctive normal form, which requires that the formula is a conjunction of clauses, which are disjunction of literals

- **Scheduling**—given n students, m courses, k time slots, is it possible to schedule final exams without conflict? (a similar problem would be to minimise conflict)
- **Sudoku**—fill in $n \times n$ sudoku board

Note 8.1.1

We can assume there is some implicit type checker for these problems. The input space, as in all decision problems, is Σ^*, but only a subset of inputs are valid encodings of the problems. Invalid inputs should be rejected.

Ponder 8.1.1

In general, many problems feel decidable but not in polytime because

- Solution space has exponential size (so decidable by brute force)
- No known mathematical structure to exploit (so unable to not check exponentially many candidate solutions)

8.1.2 The P vs NP Problem

Problem: can NP problems be solved in polytime?

It turns out that, after many years, we didn't find any polytime solution for problems we believe to be in NP. The conjecture is that $\nexists$ polytime decider for them, but we don't have a proof either.

The P vs NP problem is rather intractable. To approach the problem we ended up first finding evidence that P $\neq$ NP. One discovery connecting NP problems is that many NP problems can reduce to each other, so not having polytime decider for a set of problems is stronger evidence than not having polytime decider for one problem. Suppose a set of n languages $\{L_1, ..., L_n\}$ all polytime reduce to A. If many such L_i not in P, it's some evidence that A is also not in P.

8.1.3 Polynomial Time Reduction Methods

This helps us expand the landscapes of tractable problems and intractable problems.

Strategy 8.1.1

Cook reduction aka **Polynomial-time Turing reduction**

Given languages A and B, assume there is some decider M_B for B, build a polytime decider for A that makes use of M_B (assuming M_B is an $O(1)$ oracle).

We then say A polytime reduces to B, write $A \leq^P B$.

Corollary 8.1.1

Observe that if M_B is a polytime decider, we just decided A in polytime. Taking the contrapositive, if A is not polytime decidable, then B cannot be polytime decidable.

Corollary 8.1.2

Cook reduction is transitive.

Strategy 8.1.2

Karp reduction aka **Polynomial-time mapping reduction**

Given languages A and B, assume there is some decider M_B for B, transform a problem instance for language A into that for language B, and call M_B as a tail call to decide A.

We essentially define a polytime transformation function $f : \Sigma^* \to \Sigma^*$ such that $x \in A \Leftrightarrow f(x) \in B$.

We then say A Karp reduces to B, write $A \leq_m^P B$.

Corollary 8.1.3

Karp reduction is transitive.

8.2 Computational Hardness and Completeness

Definition 8.2.1

A is $\mathcal{C}$-hard if $P \subseteq \mathcal{C}$ and $\forall L \in \mathcal{C}, L \leq^P A$.

Intuitively, A is at least as hard as $\mathcal{C}$, since nothing in $\mathcal{C}$ is harder than A.

Definition 8.2.2

A is $\mathcal{C}$-complete if A is $\mathcal{C}$-hard and $A \in \mathcal{C}$.

This means A represents the hardest language in $\mathcal{C}$.

Note 8.2.2

These can also be defined in terms of $\leq_m^P$. Karp reduction is more general, and gives different notions of NP-completness.

Corollary 8.2.4

If A is $\mathcal{C}$-complete:

$$A \in P \Leftrightarrow \mathcal{C} = P$$

Proof.

($\Leftarrow$) trivial
($\Rightarrow$) then $\forall L \in \mathcal{C}, L \leq^P A \Rightarrow L \in P$, so $\mathcal{C} \subseteq P$, so $\mathcal{C} = P$ □

8.3 Non-Deterministic Polynomial Time (NP)

Definition 8.3.3

A language L is in NP if $\exists$ polytime verifier TM V and constant $k > 0$ s.t.

- $\forall x \in L, \exists u$ with $|u| \leq |x|^k, V(x, u)$ accepts
- $\forall x \notin L, \forall u \in \Sigma^*, V(x, u)$ rejects

Informally, if a $x \in L$, there exists some polylength proof that a polytime verifier would accept, and the verifier never accepts if $x \notin L$.

Corollary 8.3.5

Every NP problem is solvable by brute force

Strategy 8.3.3

To show a language L is in NP, construct a working verifier V.

1. Show V is polytime
2. Proof V is correct
   - Let $x \in L$, show there is a polylen proof u such that $V(x, u)$ accepts
   - Let $x \notin L$, show no proof u results in $V(x, u)$ accepts

Some known NP languages:

- 3COL
- CIRCUIT-SAT
- CLIQUE
- IS
- 3SAT

Example 8.3.1

Proving CLIQUE $\in$ NP

Construct verifier V by:

```
V ((G = (V, E)), c), S =
  if S doesn't encode c vertices then reject else
  if exists pair {x, y} in S, {x, y} not in E then reject else
  accept
```

8.3.1 NP-Completeness

Theorem 8.3.1

Cook-Levin: SAT is NP-complete

Theorem 8.3.2

3SAT is NP-complete

Theorem 8.3.3

3COL is NP-complete

Theorem 8.3.4

CLIQUE is NP-complete

Idea: build a verifier to show it's in NP. Then show 3SAT $\leq_m^P$ CLIQUE. The mapping function takes some 3CNF formula φ with m clauses, for each clause (x_i, y_i, z_i), make 3 vertices corresponding to each literal. We want some variable assignments, which we represent as a m-clique. For vertices not from the same clause, connect them if making both literals true doesn't lead to contradiction. Then, show that the constructed graph corresponds to an accept instance of "CLIQUE" with $k = m$ iff φ is satisfiable.

8.4 Example Reductions

Example 8.4.2

CLIQUE $\leq_m^P$ IND-SET

Idea: given graph G and clique size k, construct a complement graph G' (flip membership status of all possible edges in E) and feed (G', k) to a decider for IND-SET. Then show that G has k-clique iff G' has size-k independent set.

Example 8.4.3

IND-SET $\leq_m^P$ CLIQUE

Idea: do the same reduction as above, and flip the direction of implications in the correctness proof.

Example 8.4.4

3SAT $\leq_m^P$ CLIQUE

Idea: given 3CNF formula φ with k clauses. For each clause $C = (x_i, y_i, z_i)$, turn x_i, y_i, z_i into (unique) vertices. Connect the vertices such that no edge connects literals from the same clause and no edge connects contradicting assignments. Then look for a k-clique in the graph. Such clique must connect all clauses without contradiction on assignment.

Chapter 9: Randomised Algorithms

One way to deal with NP-complete problems is by approximation, and randomisation helps.

9.1 Probability Theory

Recall probability model from measure theory $(\Omega, \mathbb{P})$ where Ω is sample space and $\mathbb{P} : \Omega \to [0, 1]$ is measure.

9.1.1 CS Approach to Probability Theory

Turn real world into code, and build probability Tree.

Example 9.1.1

Flip coin; if head, throw 3-sided die; else throw 4-sided die

```
flip = Ber(1/2)
if flit = 1:
  die = randInt(1, 3)
else:
  die = randInt(1, 4)
```

9.1.2 Probability Concepts to Know

- Events e.g. $E = \{\text{die roll 3 or higher}\}$
- Conditional probability $\mathbb{P}[E_2|E_1]$
- Independence — run code in opposite order doesn't change behaviour
- Events A, B are independent if $\mathbb{P}[A \cap B] = \mathbb{P}[A]\mathbb{P}[B]$
- Random Variable $X : \Omega \to \mathbb{R}$
- Expectation $\mathbb{E}[X]$
- Indicator RV

Theorem 9.1.1

Markov's Inequality

Given non-negative RV X,

$$\mathbb{P}[X \geq c\mathbb{E}[X]] \leq \frac{1}{c}$$

or equivalently

$$\mathbb{P}[X \geq c] \leq \frac{\mathbb{E}[X]}{c}$$

Theorem 9.1.2

Chain rule. Given events $A_1, ..., A_n$,

$$\mathbb{P}\left[\bigcap_{A_k \in \{A_1,...,A_n\}} A_k\right] = \mathbb{P}[A_1]\mathbb{P}[A_2|A_1]...\mathbb{P}\left[A_n|\bigcap_{A_k \in \{A_1,...,A_{n-1}\}} A_k\right]$$

Theorem 9.1.3

Law of total probability. If $B_1, ..., B_n$ partitions Ω then

$$\mathbb{P}[A] = \prod_{A_k \in \{B_1,...,B_n\}} \mathbb{P}[A|B_k]\mathbb{P}[B_k]$$

Theorem 9.1.4

Linearity of expectation. Given RVs $X_1, ..., X_n$

$$\mathbb{E}\left[\sum c_k X_k\right] = \sum c_k \mathbb{E}[X_k]$$

9.2 Randomised Algorithms

Ponder 9.2.1

When do we need randomness in CS?

- Dinner problem—given region, how many ways to tile it with 2×1 rectangles?
  - Useful in physics
  - We have efficient randomised approximation algorithm, but not deterministic algorithm
- Distributed systems—how to break symmetry in situations like two cars running the same programme running into each other
  - Proven to require randomness
- Chicken game—Nash equilibrium: all players have no incentive to choose some other action
  - A theorem says every game has Nash equilibrium if players pick probabilistic strategies
- Cryptography
- Error correction
  - Assume probabilistic noise in data, how to recover the information?
- Communication conflict—two servers having giant files, are they the same? can we check without sending $O(n)$ bits?
  - Randomised approach doable with $O(\lg n)$ bits
- Quantum compute

9.2.1 Randomised Algorithms Definitions

Definition 9.2.1

A randomised algorithm has access to random bits.

For simplicity, we will say it has access to `RandInt` and `Bernoulli`, two random functions with $O(1)$ cost.

There are things random algorithms need to sacrifice, and there is usually a tradeoff between time and correctness. We have different types of random algorithms that choose to gamble with one of these.

Definition 9.2.2

A **Monte Carlo algorithm** A_{MC} computes $f : \Sigma^* \to \Sigma^*$ if $\forall x \in \Sigma^*$

- $\mathbb{P}[\text{result incorrect}] = \mathbb{P}[A_{\text{MC}}(x) \neq f(x)] \leq \varepsilon$ for some error rate $\varepsilon \in [0, 1)$
- $\mathbb{P}[A_{\text{MC}}(x) \text{ finishes within } T(|x|) \text{ steps}] = 1$ for some time function $T(n)$

Definition 9.2.3

A **Las Vegas algorithm** A_{LA} computes $f : \Sigma^* \to \Sigma^*$ if $\forall x \in \Sigma^*$

- $\mathbb{P}[\text{result correct}] = \mathbb{P}[A_{\text{LA}}(x) = f(x)] = 1$
- $\mathbb{E}[\text{number of steps } A_{\text{LA}}(x) \text{ takes}] \leq T(|x|)$ for some time function $T(n)$

It turns out that we can convert between these two types of random algorithms.

Theorem 9.2.5

Given $T(n)$-time LA algorithm A_{LA}. We can build $O(T(n))$ MC algorithm with any desired error rate $\varepsilon > 0$.

```
A_MC(x) =
  match A_LA(x) for maximum (1/epsilon)T(|x|) steps:
    not terminate => return whatever
    terminate with output y => return y
```

The runtime is in $O\left(\frac{1}{\varepsilon}T(n)\right) = O(T(n))$.

The error rate is the probability of A_{LA} taking too long.

$$\begin{aligned}
\mathbb{P}[\text{error}] &= \mathbb{P}\left[A_{\text{LA}}(x) \text{ step count} > \frac{1}{\varepsilon}T(|x|)\right] \\
&\leq \frac{\mathbb{E}[A_{\text{LA}}(x) \text{ step count}]}{\frac{1}{\varepsilon}T(|x|)} \\
&\leq \frac{T(|x|)}{\frac{1}{\varepsilon}T(|x|)} \\
&= \varepsilon
\end{aligned}$$

Theorem 9.2.6

Given MC algorithm A_{MC} with $T_A(n)$ run time and a checker V for A_{MC}'s result that runs in $T_V(n)$ time, we can build an LV algorithm.

```
A_LV(x) =
  loop:
    if A_MC(x) correct, return result
```

Notice we only return correct results.

The run time is distributed geometrically. The success rate is some p, so it takes $\frac{1}{p}$ iterations in expectation. That implies an expected step count in $O\left(\frac{1}{p}(T_A(|x|) + T_V(|x|))\right)$.

Example 9.2.2

Example uses of randomised algorithm with problems related to primes.

isPrime—check if input x is prime.

- We could try dividing x by 2 to $\sqrt{x}$. This is exponential time $O\left(\sqrt{2^n}\right)$.
- 2002 result: isPrime $\in P$, but known algorithm is $O(n^6)$.
- 1975 Miller-Rabin algorithm: $O(n^2)$ randomised algorithm with error rate 2^{-300}.

genPrime—generate random prime with n bits.

- Fact: about $\frac{1}{n}$ of n-bit numbers are prime.
- A natural random prime algorithm is to generate random bits of length n, check if it is prime, and repeat until we get a prime. This is $O(n^3)$.
- Whether a deterministic random prime generation algorithm exists is an open problem[11].

(**primeFactorise**—given number, return prime factorisation. This is known to be hard, and we use its hardness in cryptography.)

9.2.2 Randomised Maximum Cutting

Consider the max cut problem. Is there a way to approximate it efficiently?

Definition 9.2.4

The **maximum cut problem** is as follows, with each being equivalent:

- Given graph G, colour vertices with two different colours (each used at least once), maximising the number of edges with different coloured endpoints
- Partition V into non-empty X and Y, maximising the number of edges that go between X and Y
- Get as close to bipartite as possible

This problem is in fact NP-hard!

But we can approximate this with a trivial random algorithm

[11] In fact a PolyMath Project problem

Algorithm 9.2.1

Random maximum cutting algorithm. We essentially include each vertex in one of the partitions with $\frac{1}{2}$ probability.

```
A((G = U, V)):
  S = empty
  for v in V:
    if Ber(0.5):
      S = S union {v}
  return S
```

Let X_e be indicator for edge e being cut.

$$\mathbb{E}[\text{number of edges cut}] = \sum_{e \in E} \mathbb{E}[X_e] = \frac{m}{2}$$

Which is within a factor of 2 from optimal.

9.2.3 Randomised Minimum Cutting

Consider the opposite problem:

Definition 9.2.5

The **maximum cut problem** is as follows, with each being equivalent:

- Given graph G, colour vertices with two different colours (each used at least once), minimising the number of edges with different coloured endpoints
- Partition V into non-empty X and Y, minimising the number of edges that go between X and Y
- Estimate how connected the graph is

This problem is in P, but we will try to come up with some efficient polytime random algorithm.

Algorithm 9.2.2

Monte Carlo edge contraction algorithm for minimum cutting

Given G, start with empty S:

1. Pick radom edge e and add it to S
2. Contract e (allow multi-edge but not self-loop)
3. If $|E| > 2$, repeat from 1, else return S

This is polytime by inspection. The number of iterations is $n - 2$.

Success rate: (sketch) observe that at each step, a size k cut in the contracted graph corresponds to a size k cut in the original graph. Also, the minimum cut at any point is upper bounded by the graph's max degree. Fix some minimum cutting F, notice the algorithm returns F iff it never contracts any $f \in F$. Introduce E_i as indicator for $\nexists f \in F, f$ gets contracted at iteration i. Write $\mathbb{P}[\text{algorithm returns } F] = \mathbb{P}[\cap_{i=0}^{n-3} E_i]$. Use chain rule and complements. Upper bound $\frac{k}{m_i}$ with handshake lemma where m_i is the number of edges left at iteration i.

This should come out to be

$$\mathbb{P}[\text{algorithm returns } F] = \prod_{i=0}^{n-3}\left(1 - \frac{2}{n-i}\right) \geq \frac{2}{n(n-1)} \geq \frac{1}{n^2}$$

Fact 9.2.1

$$\forall x \in \mathbb{R}, 1 + x \leq e^x$$

Once we have a $\frac{1}{n^2}$ error rate algorithm, boosting it to a lower error rate is easy. Run it $t = n^3$ times, the error rate becomes $\left(1 - \frac{1}{n^2}\right)^t \leq e^{-\frac{t}{n^2}}$.

Strategy 9.2.1

Monte Carlo algorithms can usually be boosted by:

- Run many times and return best (in case of approximation / optimisation)
- Run many times and return the most common output (in case of decision problems)

Chapter 10: Cryptography

10.1 Modular Arithmetics

10.1.1 Many Definitions

Definition 10.1.1

$a \in \mathbb{Z}$ divides $b \in \mathbb{Z}$ if $\exists q \in \mathbb{Z}, b = qa$.

$a \mid b$ denotes a divides b.

Definition 10.1.2

$p \in \mathbb{N}, p \geq 2$ is prime if its only positive divisors are 1 and p.

Definition 10.1.3

$a \in \mathbb{Z}$ is congurent to $b \in \mathbb{Z}$ modulo $n \in \mathbb{Z}$ if $\exists k \in \mathbb{Z}, a = kn + b$. We write $a \equiv_n b$.

An equivalent definition is that $a \equiv_n b \Leftrightarrow n \mid (b - a)$

Definition 10.1.4

Let $a, b \in \mathbb{Z}$

A $c \in \mathbb{Z}$ is a **common divisor** of a and b if $(c \mid a) \wedge (c \mid b)$.

A **greatest common divisor** of a and b, denoted $\gcd(a, b)$, is $d \in \mathbb{Z}$ s.t.
- d is a common divisor of a and b
- d can be divided by all other common divisors $\forall c \in \mathbb{Z}, (((c \mid a) \wedge (c \mid b)) \Rightarrow c \mid d)$.

a and b are **coprime** aka **relative prime** if $\gcd(a, b) = 1$.

10.1.2 More Definitions

Definition 10.1.5

Define $\mathbb{Z}_N$ to be $\{x \in \mathbb{Z}, x < N\} = \{0, 1, ..., N - 1\}$.

Definition 10.1.6

Operations in $\mathbb{Z}_N$, indicated by subscript $\{+_N, \cdot_N\}$, is defined as the operation then modulo.

Example 10.1.1

- $a +_N b = (a + b) \bmod N$
- $a \cdot_N b = (a \cdot b) \bmod N$

The inverse of those operations, $\{-_N, /_N\}$, would be defined as the inverse of the respective operation.

Example 10.1.2

- $a -_N b = a +_N -b$
- $a /_N b = a *_N b^{-1}$, if a^{-1} exists

Inverses and identities:

- Addictive identity of $a \in \mathbb{Z}_n$ is 0
- Addictive inverse of $a \in \mathbb{Z}_n$ is $(-a \bmod n) \in \mathbb{Z}$
- Multiplicative identity of $a \in \mathbb{Z}_n$ is 0
- Multiplicative inverse of $a \in \mathbb{Z}_n$ is $a^{-1} \in \mathbb{Z}_n, a^{-1} \cdot_n a = 1$

Fact 10.1.1

Facts[12]

- $(a + b) \bmod n = (a \bmod n) +_n (b \bmod n)$
- $(a \cdot b) \bmod n = (a \bmod n) \cdot_n (b \bmod n)$
- Multiplicative of $a \in \mathbb{Z}_n$ exists iff $\gcd(a, n) = 1$

Definition 10.1.7

Define $\mathbb{Z}_N^*$ to be the subset of $\mathbb{Z}_N$ such that all elements have multiplicative inverse:

$$\mathbb{Z}_N^* = \{x \in \mathbb{Z}_N \mid \gcd(x, N) = 1\}$$

Fact 10.1.2

$\mathbb{Z}_N^*$ is closed under multiplication[13].

Also, if $\{x_1, ..., x_m\} = \mathbb{Z}_N^*$ then forall $x_i \in \mathbb{Z}_N^*$, $(x_i \cdot_N x_1, ..., x_i \cdot_N x_m)$ is a pemutation of $\mathbb{Z}_N^*$.

[12] too lazy to prove them
[13] also too lazy to prove

Definition 10.1.8

The Euler toient function φ returns the size of $\mathbb{Z}_N^*$:

$$\varphi(N) = |\mathbb{Z}_N^*|$$

Fact 10.1.3

If P, Q are distinct primes, $\varphi(PQ) = (P-1)(Q-1)$

10.1.3 Modular Exponentiation

Definition 10.1.9

For $a \in \mathbb{Z}_N, E \in \mathbb{Z}$, define A^E to be A multiplied together by $\cdot_N$ E times.

Theorem 10.1.1

Euler's theorem

$$\forall A \in \mathbb{Z}_N^*, A^{\varphi(N)} = 1$$

equivalently,

$$\forall A, N \in \mathbb{Z}, \gcd(A, N) = 1 \Rightarrow A^{\varphi(N)} = 1$$

Fact 10.1.4

For $A \in \mathbb{Z}_N^*, E \in \mathbb{Z}$, $A^E \equiv_N A^{E \bmod \varphi(N)}$

Corollary 10.1.1

Then we can think of the exponent as living in the $\mathbb{Z}_{\varphi(N)}$ universe since every other exponent can be reduced to $E \bmod \varphi(N)$

Definition 10.1.10

$A \in \mathbb{Z}_N^*$ is a generator if taking exponents of A generates $\mathbb{Z}_N^*$ i.e.

$\left\{A^E \mid E \in \mathbb{Z}_{\varphi(N)}\right\} = Z_N^*$

10.1.4 Complexity of Modular Operations

Operation	Complexity	Comment
$+_N, -_N$	poly	
$\cdot_N, /_N$	poly	
Check existence of B^{-1}	poly	By checking if $\gcd(B, N) = 1$ with Euclidean algorithm
Compute B^{-1}	poly	with extended Euclidean algorithm

Compute A^E	poly	via fast modular exponentiation
Compute $\log_B A$	not known to be poly	
Compute $\sqrt[E]{A}$	not known to be poly	

Algorithm 10.1.1

Extended Euclidean algorithm[14]

We find $d, k, l \in \mathbb{Z}$ such that $d = \gcd(a, b) = ka + lb$

Algorithm 10.1.2

Reduce finding multiplicative inverse to Extended Euclidean algorithm

Given $1 = \gcd(B, N) = kB + lN$, we know $1 \equiv_N kB$, so setting $B^{-1} = k$ works.

Algorithm 10.1.3

Fast modular exponentiation given $N \in \mathbb{N}$, $A \in \mathbb{Z}_N$, $E \in \mathbb{N}$:

Use repeated squaring to get $A^2 \bmod N$, $A^4 \bmod N$, $A^8 \bmod N$, ... until sufficient.

Multiple together some subset of {A^2 mod N, A^4 mod N, A^8 mod N, ...} so that the exponents sum to E.

10.2 Private Key Encryption

Notation:

- K = key
- C = ciphertext
- M = message

10.2.1 One-Time Pad

A simple encryption scheme that can only be used once (perfectly secure on the first message, but reuse of key is insecure).

Algorithm 10.2.4

Encode and decode with one-time pad.

Encode:

1. Write $M \in \{0,1\}^n$
2. Generate uniform random key $K \in \{0,1\}^n$
3. Take $C = \text{bitwise\_xor}(M, K)$

The ciphertext should be uniformly random in $\{0,1\}^n$.

Decode:

1. Compute $M = \text{bitwise\_xor}(C, K)$

[14] can just look this up

10.2.2 Diffie-Hellman Key Exchange

A protocol to share secret key through public communication built upon the assumption that descrete log in modular universe is hard.

Algorithm 10.2.5

Diffie-Hellman Key Exchange

The idea is that adversaries cannot undo exponentiation, so sending exponents of a number will not leak the exponent.

A

- Choose some large prime p
- Choose some generator $G \in \mathbb{Z}_p^*$ (so that G^{E_1}, G^{E_2} are uniform random)
- Compute $\varphi(p) = (p-1)$
- Randomly pick some $E_1 \in \mathbb{Z}_{\varphi(p)}$ (then exponents can be reduced by mod $\varphi(p)$)
- Compute $G^{E_1} \in \mathbb{Z}_p^*$ with fast exponentiation
- Send out (p, G, G^{E_1})

B

- Receives (p, G, G^{E_1})
- Randomly pick some $E_2 \in \mathbb{Z}_{\varphi(p)}$
- Compute $G^{E_2} \in \mathbb{Z}_p^*$
- Compute $K = G^{E_1 E_2} = \left(G^{E_1}\right)^{E_2} \in \mathbb{Z}_p^*$
- Send back G^{E_2}

A

- Receives G^{E_2}
- Computes $K = G^{E_1 E_2} = \left(G^{E_2}\right)^{E_1} \in \mathbb{Z}_p^*$

Now both A and B have $K = G^{E_1 E_2}$.

Adversary

- Sees (p, G, G^{E_1}, G^{E_2})
- No easy way to compute $E_1, E_2, G^{E_1 E_2}$

10.3 Public Key Encryption

Notation:

- K = secret key
- K_{pub} = public key
- K_{priv} = private key
- C = ciphertext
- M = message

The goal is that public keys can encrypt messages so that they can be decrypted only by someone's private key.

10.3.1 ElGamal

Public-private key scheme using Diffie-Hellman secret key exchange.

Algorithm 10.3.6

ElGamal encryption scheme

A, **B** first perform Diffie-Hellman to get shared secret key $K = G^{E_1 E_2}$.

B (receiver)
- Recall p, G, E_2
- Set ${K_{\text{pub}}}^B = (p, G, E_2)$
- Publish ${K_{\text{pub}}}^B$

A
- Recall p, G, E_1
- Write message $M \in \mathbb{Z}_p^*$
- Encrypt by $C = MG^{E_1 E_2}$
- Send out C

B
- Receives $C = MG^{E_1 E_2}$
- Compute $\left(G^{E_1 E_2}\right)^{-1}$
- Decrypt by $M = C\left(G^{E_1 E_2}\right)^{-1} = MG^{E_1 E_2}\left(G^{E_1 E_2}\right)^{-1}$

10.3.2 RSA

Public-key encryption scheme built upon the assumption that taking roots in modular universe is hard.

Algorithm 10.3.7

RSA encryption scheme

B (receiver)
- Pick distinct large primes P, Q
- Compute $N = PQ$
- Pick $E \in \mathbb{Z}^*_{\varphi(N)}$
- Compute $E^{-1} \in \mathbb{Z}^*_{\varphi(N)}$
- Set $K_{\text{priv}} = E^{-1}$
- Publish $K_{\text{pub}} = (N, E)$

A
- Obtain $K_{\text{pub}} = (N, E)$
- Write message $M \in \mathbb{Z}^*_N$
- Encrypt by $C = M^E \in \mathbb{Z}^*_N$
- Send out C

B
- Receives $C = M^E \in \mathbb{Z}^*_N$
- Compute $C^{E^{-1}} = M^{E(E^{-1})} = M \in \mathbb{Z}^*_N$

Adversary
- Sees N, but can't factor it into P, Q easily, and can't compute $\varphi(N)$ efficiently[15][16]
- Sees E, M^E, but can't take root to get M efficiently

[15] turns out prime factorisation is polytime iff computing $\varphi(N)$ is polytime

[16] it is not proven that there is no way to decrypt without knowing $\varphi(N)$, but no known polytime algorithm either