Theory Of Computation

Muyao Xiao

Nov. 2024

Abstract

The note is taken by studying 18.404J/6.5400 *Theory of Computation* course by professor Michael Sipser of MIT. The course material can be downloaded in MIT OpenCourseWare. Meanwhile, most contents in this note will also be derived from his book *Introduction to the Theory of Computation, third edition.*

The course is divided into 2 parts, computational theory and complexity theory. Computational theory is developed during 1930s - 1950s. It concerns about what is computable. This note will be focused on the first part.

Example. Program verification, mathematical truth

Example (Models of Computation)**.** Finite automata, Turing machines, $\cdots$

Contents

Chapter 1

Introduction, Finite Automata, Regular Expressions

The theory of computation begins with a question: What is a computer. The real computer is too complicated to understand, to start with, we use an idealized computer called **computational model**.

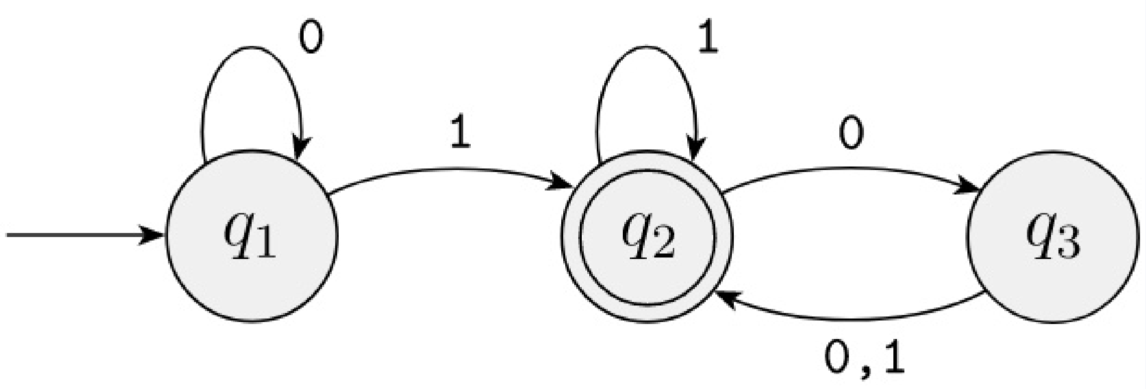
The simplest model among them is **finite state machine** or **finite automaton**.

1.1 Finite Automata

Finite automata are good models for computers with an extremely limited amount of memory.

Finite automata and their probabilistic counterpart **Markov chains** are useful tools when we're attempting to recognize patterns in data. Markov chains have even been used to model and predict price changes in financial markets.

Example (Finite Automata Example). Here's an example of finite automata:



- The figure is called **state diagram** of M_1.
- Three **states**: q_1, q_2 and q_3.
- **Start state**: q_1.
- **Accept state**: q_2.
- The arrows going from one state to another are called **transitions**.

When the automaton receives an input string such as `1101`, it processes that string and produces an output. The output is either **accept** or **reject**:

1. Start in state q_1
2. Read `1`, follow transition from q_1 to q_2

3. Read 1, follow transition from q_2 to q_2

4. Read 0, follow transition from q_2 to q_3

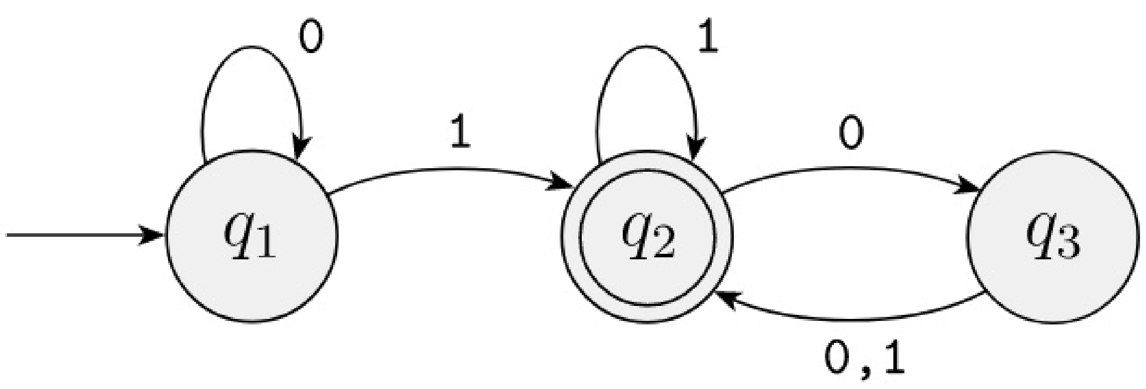
5. Read 1, follow transition from q_3 to q_2

6. *Accept* because M_1 is in an accept state q_2 at the end of the input

Definition 1.1.1 (Formal Definition of A Finite Automaton)**.** A **finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called **state**
2. Σ is a finite set called the **alphabet**
3. $\delta : Q \times \Sigma \Rightarrow Q$ is the **transition function**
4. $q_0 \in Q$ is the **start state**
5. $F \subseteq Q$ is the **set of accept state**

Example (Revisit Finite Automata Example)**.** Let's revisit the finite automata example M_1 and see from the formal definition perspective:



We can describe M_1 formally by writing $M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$
2. $\Sigma = \{0, 1\}$
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

4. q_1 is the start state
5. $F = \{q_2\}$

If A is the set of all strings that machine M accepts, we say that A is the **language of machine M** and write $L(M) = A$. We say that **M recognizes A** or that **M accepts A**. Here because *accept* has different meaning, we use *recognize* for the language.

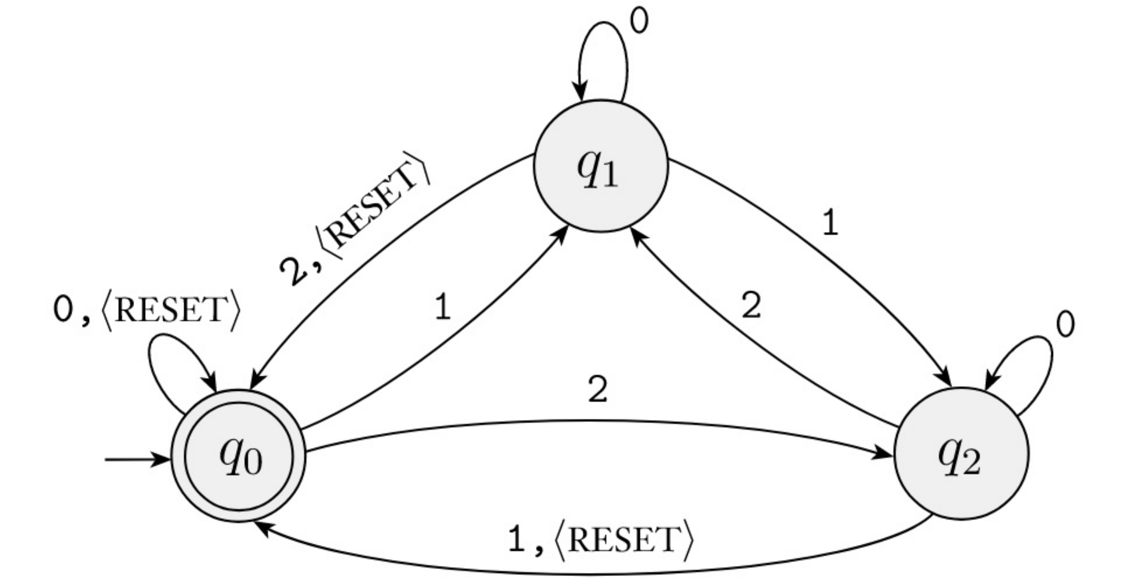
Remark. A machine may accept several strings, but it always recognizes only one language. If the machine accepts no strings, it still recognizes one language – namely, the empty language $\emptyset$.

Example (Revisit Finite Automata Example: Language)**.** In our example, the language set A can be represented as:

A = $\{\omega | \omega$ contains at least one `1` and an even number of `0`s follow the last `1`$\}$.

Then $L(M_1) = A$, or equivalently, M_1 recognizes A.

Example. When describing such a machine:



The alphabet $\Sigma = \{1, 2, 3, < RESET >\}$, we treat $< RESET >$ as a single symbol.

The machine keeps a running count of the sum of the numerical input symbols it reads, modulo 3. Every time it receives $< RESET >$ symbol, it resets the count to 0. It accepts if the sum is 0 modulo 3.

1.2 Formal Definition of Computation

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a FA and let $\omega = \omega_1\omega_2 \cdots \omega_n$ be a string where each ω_i is a member of alphabet Σ. Then M **accepts** ω if a sequence of state $r_0, r_1, \cdots, r_n$ in Q exists with three conditions:

1. $r_0 = q_0$ (machine starts at initial state)
2. $\delta(r_i, \omega_{i+1}) = r_{i+1}$ (machine goes form state to state following the transition function)
3. $r_n \in F$ (machine accepts its input if it ends up in an accept state)

We say that M recognizes language A if $A = \{\omega | M accepts \omega\}$

Note. A is the language, ω is the accepted string. A is the set of all instances of ω.

We say a machine "accepts" a string, and a machine "recognizes" a language.

Definition 1.2.1 (Regular Language)**.** A language is called a **regular language** if some finite automaton recognizes it.

Example. Let B = { ω | ω has even number of 1s }

B is a regular language.

Example. Let C = { ω | ω has equal numbers of 0s and 1s }

C is <u>not</u> a regular language.

1.3 Regular Expressions

1.3.1 Regular Operations

Definition 1.3.1. Let A and B be languages, we define the regular operations **union** , **concatenation** , and **start** as follows:

- Union: $A \cup B = \{x|x \in A||x \in B\}$
- Concatenation: $A \circ B = \{xy|x \in A\&y \in B\}$
- Star: $A^* = \{x_1x_2 \cdots x_k|k \geq 0\&x_i \in A\}$

Notice that ϵ(empty language) always belongs to $A*$.

Example. Σ^*1 is the language end with 1

Remark. Show finite automata equivalent to regular expressions.

1.3.2 Closure Properties

Theorem 1.3.1. The class of regular language is closed under the union operation.
In other words, if A_1 and A_2 are regular languages, so is $A_1 \cup A_2$.

Proof. Let $M_1 = \{Q_1, \Sigma, \delta_1, q_1, F_1\}$ recognize A_1.
Let $M_2 = \{Q_2, \Sigma, \delta_2, q_2, F_2\}$ recognize A_2. (assuming in the same alphabet to make the proof simple)
Construct $M = (Q, \Sigma, \delta, q_0, F)$ recognizing $A_1 \cup A_2$.
M should accept input w if either M_1 or M_2 accepts w.
Component of M:

- $Q = Q_1 \times Q_2$
- $q_0 = (q_1, q_2)$
- $\delta((q, r), a) = (\delta_1(q, a), \delta(r, a))$
- $F = (F_1 \times Q_2) \cup (Q_1 \times F2)$
- not $F = \cancel{F_1 \times F2}$ (this gives intersection!)

■

Example (What is close?). Positive integers close under addition but not close under subtraction.

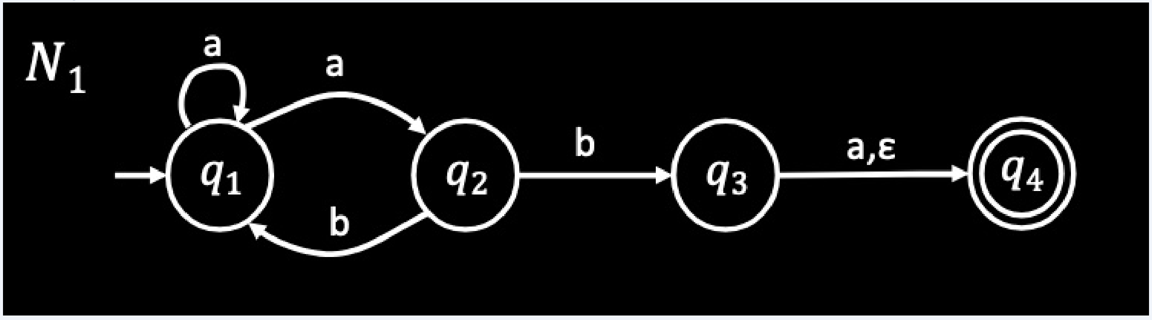
Theorem 1.3.2. The class of regular language is closed under the concatenation operation.
In other words, if A_1 and A_2 are regular languages then so is $A_1 \circ A_2$.

Chapter 2

Nondeterminism, Closure Properties, Regular Expressions –> Finite Automata

2.1 Nondeterminism

Example. This is a nondeterministic finite automaton:



What's the difference with what we saw in the last lecture:

- in q_1, when accepting an a, you can either stay in q_1 or go to q_2
- in q_1, if getting b then there's nowhere to go
- ...

Example inputs

- ab (**accept**)
- aa (**reject**)

New features of nondeterminism:

- multiple paths possible (0, 1 or many at each step)
- ϵ-transition is a "free" move without reading input
- Accept input if some path leads to accept state (acceptance overrules rejection) (if one of possible ways to go accepts, then accepts)

Nondeterminism doesn't correspond to a physical machine we can build, however it is useful mathematically.

2.2 NFA

Definition 2.2.1. A **nondeterministic finite automaton** is a 5-tuple $Q, \Sigma, \delta, q_0, F$, where

1. Q is a finite set of states
2. Σ is a finite alphabet
3. $\delta : Q \times \Sigma_\epsilon \Rightarrow \mathsf{P}(Q)$ is the transition function
4. $q_0 \in Q$ is the start state
5. $F \subseteq Q$ is the set of accept states

In which, Σ_ϵ is a shorthand of $\Sigma \cup \{\epsilon\}$. P(Q) means the power set of Q which can be represented as $P(Q) = R|R \subseteq Q$, which is the set which contains all the subset of Q.

Example. Check the NFA example, we can write transition function such as:

- $\delta(q_1, a) = \{q_1, q_2\}$
- $\delta(q_1, b) = \emptyset$

Ways to think about nondeterminism:

Computational view: Fork new parallel thread and accept if any thread leads to an accept state.

Mathematical view: Tree with branches, accept if any branch leads to an accept state.

Magical: Guess at each nondeterministic step which way to go. Machine always makes the right guess that leads to accepting, if possible.

Theorem 2.2.1. If an NFA recognizes A then A is regular.

Proof. By showing how to convert an NFA to an equivalent DFA.

Let NFA $M = (Q, \Sigma, \delta, q_0, F)$ recognize A, we're going to construct DFA $M' = (Q', \Sigma, \delta', q_0', F')$ recognizing A.

IDEA: DFA M' keeps track of the subset of possible states in NFA M.

Construct of M':

- $Q' = P(Q)$ (States of M' is the power set of Q)
- $\delta'(R, a) = \{q | q \in \delta(r, a) for\ some\ r \in R\}$ $(R \in Q')$
- $q_0' = \{q_0\}$
- $F' = \{R \in Q' | R\ intersects\ F\}$

■

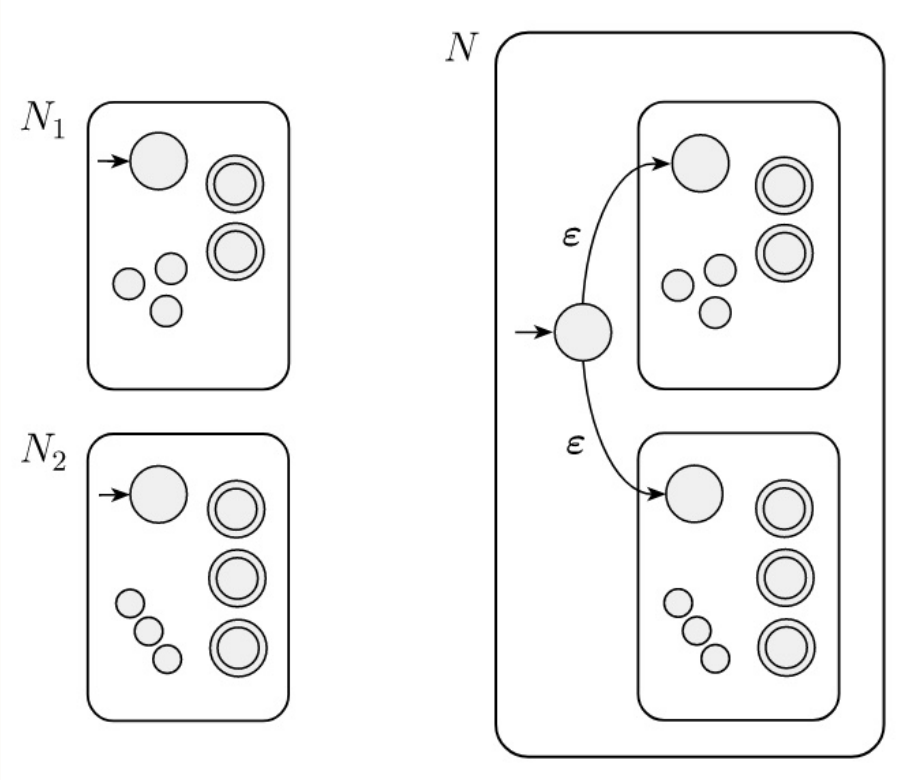
Remark. If M has n states, how many states does M' have by this construction?

The answer is 2^n.

Return to Union Closure Property and Concatenation Closure Property by constructing an NFA to prove.

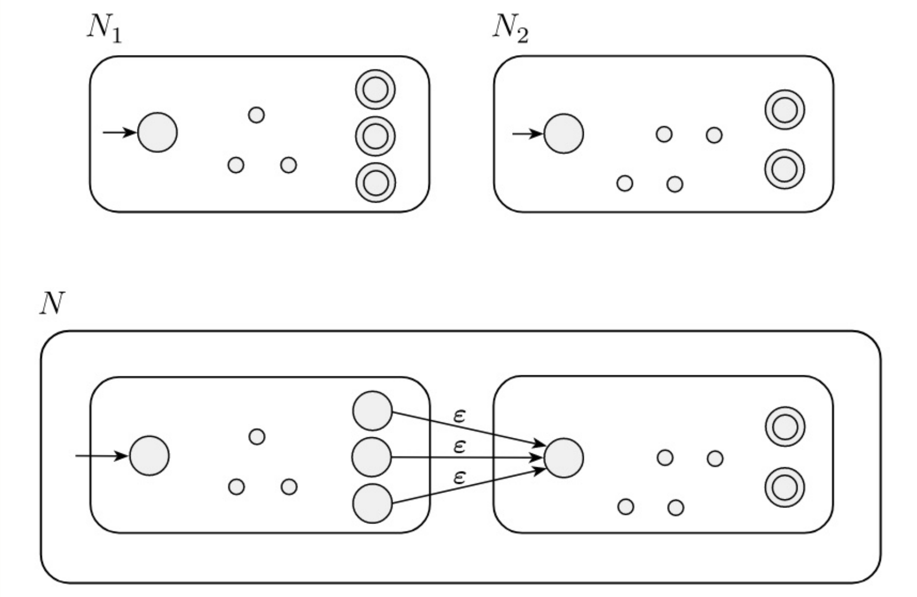
Proof: Closure Properties Union. N1 corresponds to the DFA M1 recognizes A_1, and N2 corresponds to an input of DFA M2 who recognizes A_2.

We can construct an NFA like following:



■

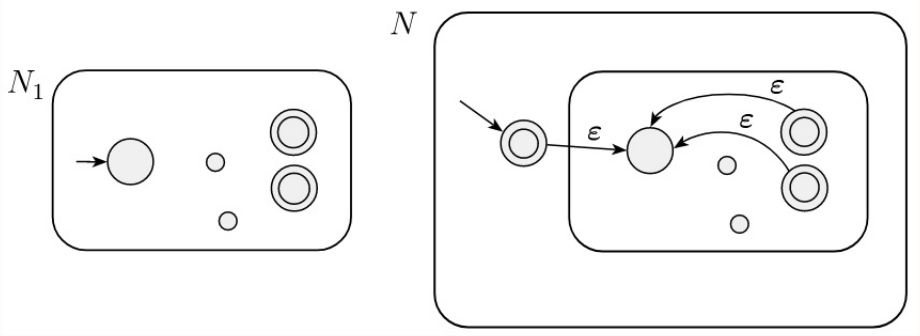
Proof: Closure Properties Concat. Similar with the last one, this one constructs an NFA recognizes $A_1 A_2$:



■

Theorem 2.2.2 (Closure Under Star)**.** If A is a regular language, so is $A*$.

Proof. Given DFA M recognizing A, construct NFA M' recognizing A^*.



■

Remark. M' have n + 1 states in this construction.

Theorem 2.2.3. If R is regular expression and $A = L(R)$ then A is regular.

That is to say: A language is regular and only if some regular expression describes it.

This theorem has two directions:

Lemma 2.2.1. If a language is described by a regular expression, then it is regular

Proof. IDEA: Say that we have a regular expression R describing some language A. We show how to convert R into an NFA recognizing A.

When R is atomic, we can easily construct NFA for:

- R = a for some $a \in \Sigma$
- $R = \epsilon$
- $R = \emptyset$

When R is composite, as we have already constructed for them (closure properties).

Note. This proof works in a recursive way.

■

Lemma 2.2.2. If a language is regular, then it is described by a regular expression.

Proof. This will be proved in next lecture. ■

Note. Before going to prove this theorem, I need to make some clarifications here.

- **Regular language**: if some finite automaton recognizes a language, then it is called regular language.
- We say a machine recognizes a language if the language is the set contains all the string that can run the machine into an accept state.

Also we need the definition of **regular expression** here:

Definition 2.2.2. Say that R is a **regular expression** if R is

1. `a` for some `a` in the alphabet Σ
2. ϵ
3. $\emptyset$
4. $R_1 \cup R_2$ where R_1 and R_2 are regular expressions
5. $R_1 \circ R_2$ where R_1 and R_2 are regular expressions
6. R_1^* where R_1 is a regular expression

Chapter 3

The Regular Pumping Lemma, Finite Automata → Regular Expressions, CFGs

3.1 GNFA

Let's go back to the Lemma in the previous lecture:

Theorem 3.1.1 (DFA -> Regular Expressions). If a language is regular, then it is described by a regular expression.

In another word, if A is regular then $A = L(R)$ for some regular exp R.

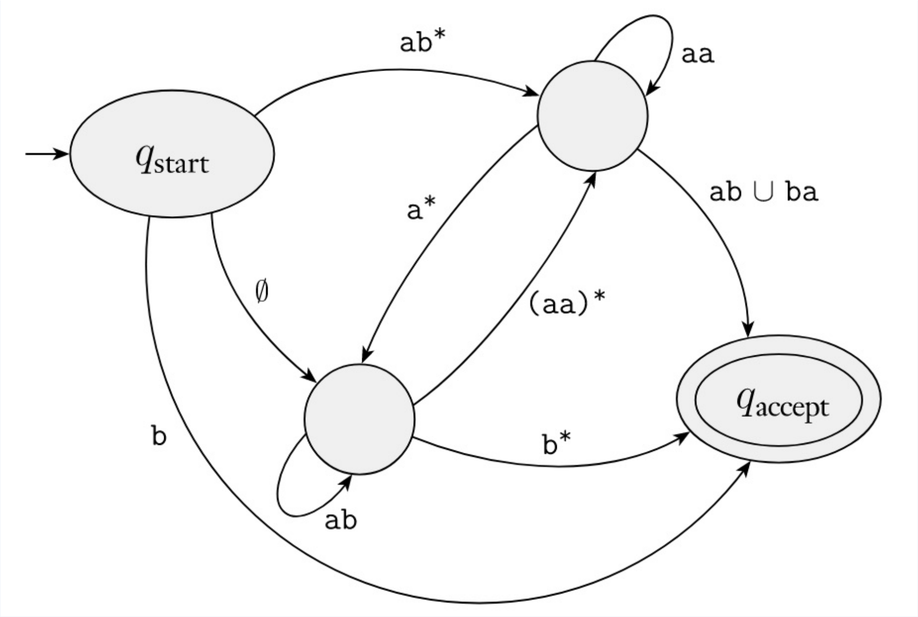
Proof. IDEA: Give conversion DFA $M \to R$ We need new tool: Generalized NFA ∎

Definition 3.1.1 (Generalized NFA). A Generalized nondeterministic Finite Automaton(GNFA) is similar to an NFA, but allows regular expressions as transition labels.

The formal definition is that a **generalized nondeterministic finite automaton** is a 5-tuple, $(Q, \Sigma, \delta, q_{start}, q_{accept})$, where

1. Q is the finite set of states
2. Σ is the input alphabet
3. $\delta : (Q - \{q_{accept}\}) \times (Q - \{q_{start}\}) \to \mathsf{R}$ is the transition function
4. q_{start} is the start state
5. q_{accept} is the accept state

Example (Example of GNFA). Similar with NFA but transitions allow regular expressions



Now we're going to covert GNFA to regular expressions to proof the Lemma.
For convenience, we will assume (we can easily modify the machine to achieve):

- One accept state, separate from the start state.
- One arrow from each state to each state, except
  - only exiting the start state
  - only entering the accept state

Lemma 3.1.1 (GNFA -> Regular Expressions)**.** Every GNFA G has an equivalent regular expression R.

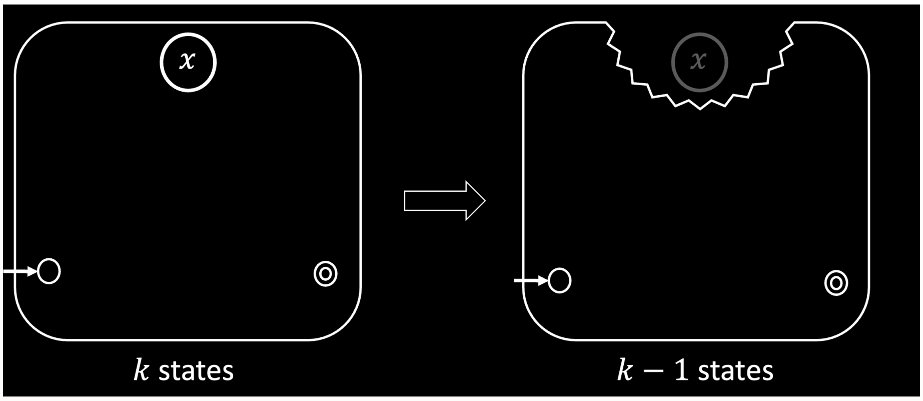
Proof. By induction(recursion) on the number of states k of G.

Basis($k = 2$):G can only looks like:
G = ->start –r–>accept
In this case, R = r.

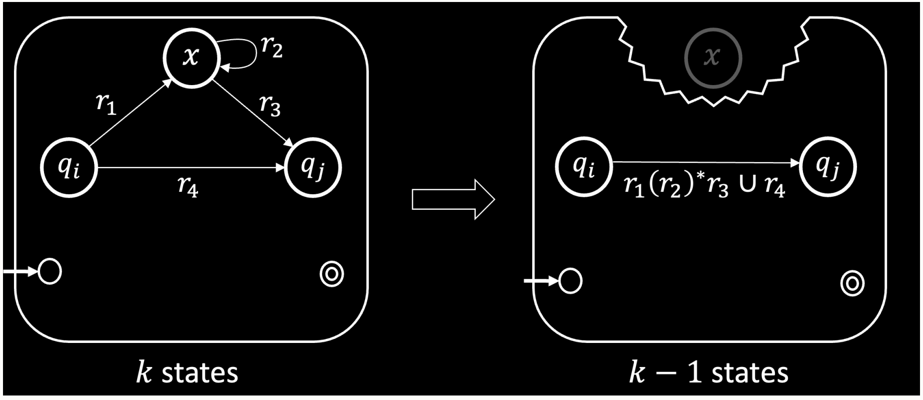
Induction step($k > 2$): Assume Lemma true for $k - 1$ states and prove for k states
IDEA: Convert k-state GNFA to equivalent $(k - 1)$-state GNFA

1. First we pick any state x except the start and accept states.
2. Remove x.
3. Repair the damage by recovering all paths that went through x.
4. Make the indicated change for each pair of states q_i, q_j.

In the following example, this is how we remove a state x:



Then we replace the original paths with new paths:



We're already done, because DFAs are a type of GNFAs. ■

3.2 Non-Regular Languages

How do we show a language is not regular?

- To show a language is regular we give a DFA.
- To show a language is not regular, we must give a proof (It is not enough to say that you couldn't find a DFA for it, therefore the language is not regular).

Example. Here $\Sigma = \{0, 1\}$

1. B has equal numbers of 0 and 1
   Intuition: B is not regular because DFAs cannot count unboundedly.
2. C has equal numbers of `01` and `10` substrings.
   `0101` $\notin$ C
   `0110` $\in$ C
   Intuition: C is not regular because DFAs cannot count unboundedly. (Wrong! Actually, C is regular!)

Moral: You need a proof.

3.3 Pumping Lemma

Pumping lemma is the method for proving non-regularity.

Lemma 3.3.1 (Pumping Lemma)**.** For every regular language A, there's a number p (the "pumping length") such that if $s \in A$ and $|s| \geq p$ then $s = xyz$ where

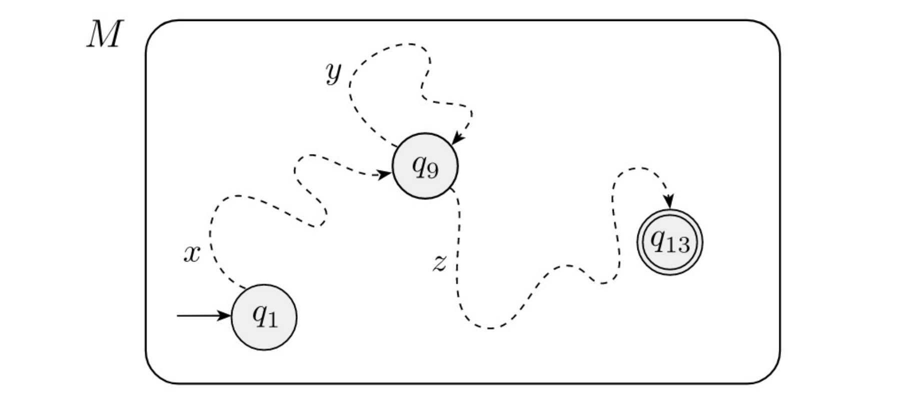
1. $xy^iz \in A$ for all $i \geq 0$ $\quad y^i = yy\cdots y$(i number of y)
2. $y \neq \epsilon$
3. $|xy| \leq p$

> **Remark.** $|s|$ means the length of the string s.

Informally: A is regular -> every long string in A can be pumped and the result stays in A.

Proof. Let DFA M recognize A. Let p be the number of states in M. Pick $s \in A$ where $|s| \geq p$.

When you have a too long string, you inevitably have to revisit the same state again and again, forming the struct as following:



This is also known as The Pigeonhole Principle. ■

Example (Prove Non-regularity). Let $D = \{0^k1^k | k \geq 0\}$

Proof. Show: D is not regular

Proof by Contradiction:

Assume that D is regular, applying pumping lemma, let $s = 0^p1^p \in D$.

Pumping lemma says that can divide $s = xyz$ satisfying the 3 conditions.

But $xyyz$ has excess 0s and thus $xyyz \notin D$ contradicting the pumping lemma. ■

Example. Let $F = \{ww | w \in \Sigma^*\}$, Say $\Sigma^* = \{0, 1\}$

Proof. Assume F is regular.

Try $s = 0^p10^p1 \in F$, show cannot be pumped $s = xyz$ satisfying the 3 conditions. ■

Variant: Combine closure properties with the pumping lemma.

Example. Let $B = \{w | w$ has equal number of 0s and 1s$\}$.

Proof. Assume that B is regular.

We know that 0^*1^* is regular so $B \cap 0^*1^*$ is regular (closure under intersection).

But $D = B \cap 0^*1^*$ and we already showed D is not regular. Contradiction! ■

3.4 Context Free Grammars

It is a stronger computation model.

Chapter 4

Pushdown Automata, CFG<–>PDA

4.1 CFG: Context Free Grammars

Using FA and regular expressions, some language such as $\{0^n1^n|n \geq 0\}$ can not be described.

Context-free grammars is a more powerful method of describing languages. Such grammar can describe certain features that have a recursive structure, which makes them useful in a variety of applications.

An important application of CFG occurs in the specification and compilation of programming languages. A number of methodologies facilitate the construction of a **parser** once a CFG is available. Some tools even automatically generate the parser from the grammar.

Example (CFG example)**.** The following CFG we call G_1:

$$\begin{aligned} A &\rightarrow 0A1 \\ A &\rightarrow B \\ B &\rightarrow \# \end{aligned}$$

Shorthand:

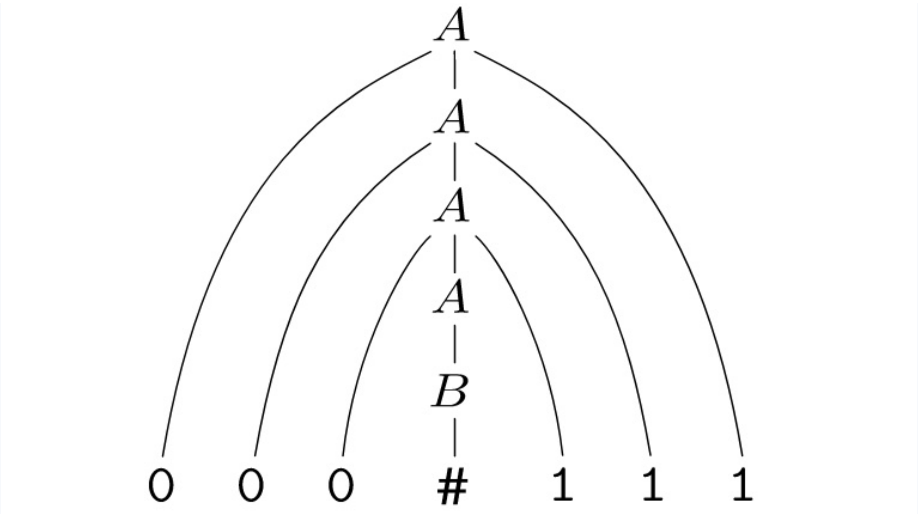
$$\begin{aligned} A &\rightarrow 0A1|B \\ B &\rightarrow \# \end{aligned}$$

- It contains a collection of **substitution rules**, also called **productions**.
- Left side: **variable**, capital letters
- Right side: **terminals**, analogous to the input alphabet and often represented by lowercase letters, numbers or special symbols
- In this example, G_1's variables are A and B, its terminals are 0, 1 and #.

Example of using G_1 to generate string `000#111`:

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

The above information can be shown in a **parse tree**:



Definition 4.1.1 (CFG). A **context-free grammar** is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called the **variables**
2. Σ is a finite set, disjoint from V, called the **terminals**
3. R is a finite set of **rules**, with each rule being a variable and a string of variables and terminals (rule form: $V \to (V \cup \Sigma)^*$)
4. $S \in V$ is the start variable

For $u, v \in (V \cup \Sigma)^*$ write

1. $u \Rightarrow v$ if can go from u to v with one substitution step in G.
2. $u \overset{*}{\Rightarrow} v$ if can go from u to v with some number of substitution steps in G.
   $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k = v$ is called a derivation of v from u.

Definition 4.1.2 (Context Free Language). Given $L(G) = \{w|w \in \Sigma^* \; and \; S \overset{*}{\Rightarrow} w\}$
A is a Context Free Language(CFL) if $A = L(G)$ for some CFG G.

Example. We have 2 set of rules here:
C_1:

$$B \to 0B1|\epsilon$$
$$B1 \to 1B$$
$$0B \to B0$$

C_2:

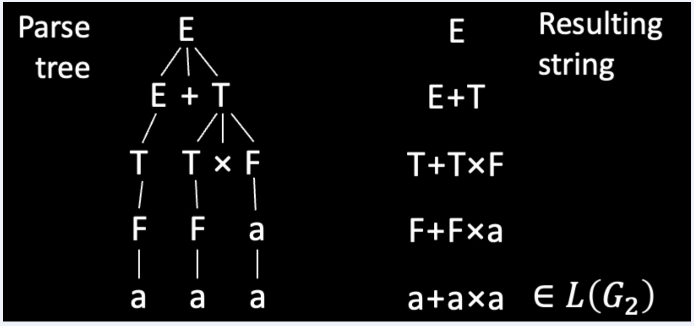
$$S \to 0S|S1$$
$$R \to RR$$

C_1 is not a CFG, because the left side is not pure variable, it has "context".
C_2, on the other hand, even though we can not derive a string with all terminals, but it does not violate the rule, so C_2 is a CFG.

Example (CFG). G_2:

$$E \to E + T | T$$
$$T \to T \times F | F$$
$$F \to (E) | a$$

$V = \{E, T, F\}$
$\Sigma = \{+, \times, (,), a\}$
$R =$ the 6 rules above
$S = E$
Showing the generating tree, it even shows the precedence of the operator × is higher than +:



If a string has 2 different parse trees then it is derived **ambiguously** and we say that the grammar is ambiguous.

Example (Ambiguity). G_2:

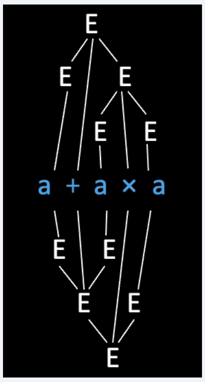
$$E \to E + T | T$$
$$T \to T \times F | F$$
$$F \to (E) | a$$

G_3

$$E \to E + E | E \times E | (E) | a$$

Both of them recognize the same language, $L(G_2) = L(G_3)$, however G_2 is an unambiguous CFG and G_3 is ambiguous.

This tree shows the ambiguity of G_3:



4.2 Pushdown Automata (PDA)

The **Pushdown Automaton** operates like an NFA except can write-add(push) or read-remove(pop) symbols from the top of stack.

Example (PDA). PDA for $D = \{0^k 1^k | k \geq 0\}$

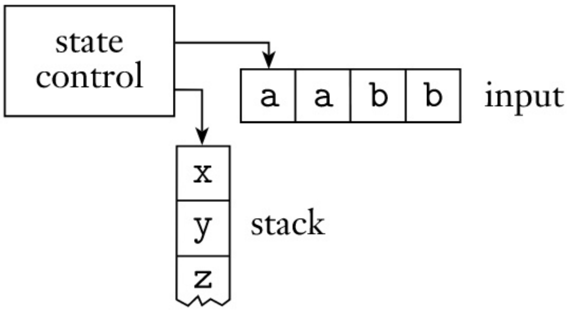


Figure 4.1: Schematic of a pushdown automaton

1. Read 0s from input, push onto stack until read 1.
2. Read 1s from input, while popping 0s from stack
3. Enter accept state if stack is empty (acceptance only at end of input)

Definition 4.2.1 (PDA). A **pushdown automaton** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, F are all finite sets:

1. Σ input alphabet
2. Γ stack alphabet
3. δ: $Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathsf{P}(Q \times \Gamma_\epsilon)$ ($\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$, we allow nondeterminism here)
   Left side is the domain(the set of possible inputs to a function) of the transition function; The current state, and the reading(popping) from the top of the stack and reading from the input (can read ϵ) decide the input of the transition function
   Right side is the writing, the combination of the state and the writing to the top of the stack (can write ϵ) decide the output of the transition
4. q_0: the start state
5. F: the accept states

For a transition function like $\delta(q, a, c) = \{(r_1, d), (r_2, e)\}$, the current state is q, and reading an input a and popping from the stack c, then we might end up going to:

- state r_1 and writing d to the top of stack
- state r_2 and writing e to the top of the stack

One interesting thing about this definition is that we have no primitive to decide whether the stack is empty at the end.

The reason for that is **we don't need to**! Because we can write special symbol to the bottom of the stack at the beginning when the machine starts.

Remark. Should do some example here to understand deeper into the PDA.

Theorem 4.2.1 (Equal of CFL and PDA). A language is context free if and only if some pushdown automaton recognize it.

Proof. We need to prove it in 2 directions:

Lemma 4.2.1 (CFL to PDA)**.** If a language is context free, then some pushdown automaton recognize it.

and:

Lemma 4.2.2 (PDA to CFL)**.** If a pushdown automaton recognize some language, then it is context free.

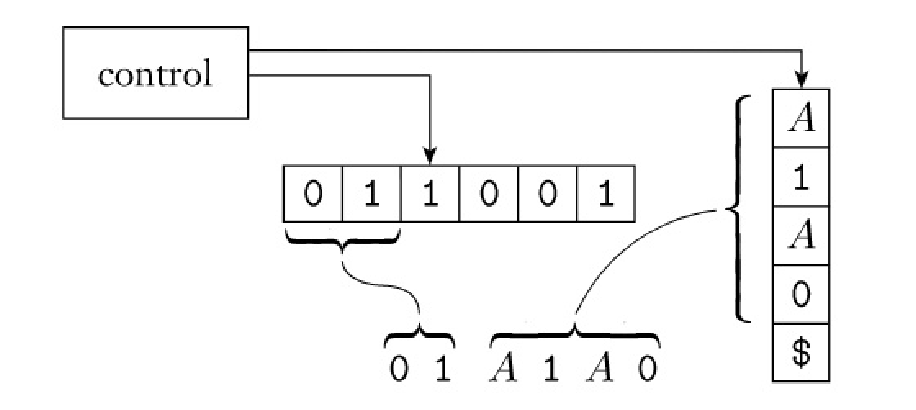
■

CFL to PDA. The idea is to use the stack to do partial substitution, and if the top of the stack matches the input string, we pop it out.

The reason that why we can always do the correct substitution is because of the nondeterminism nature of PDA, so actually we fork a lot of branches and there'll be the correct substitution in it.

Here's the informal description of the PDA P:

1. Place the marker symbol \$ and the start variable on the stack.
2. Repeat the following steps forever:
   (a) If the top of the stack is a variable A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.
   (b) If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If match, repeat; if not, reject on this branch of the nondeterminism.
   (c) If the top of stack is the symbol \$, enter the accept state. Doing so accepts the input if it has all been read.

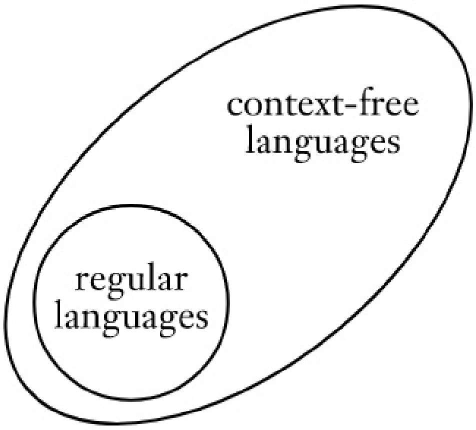


■

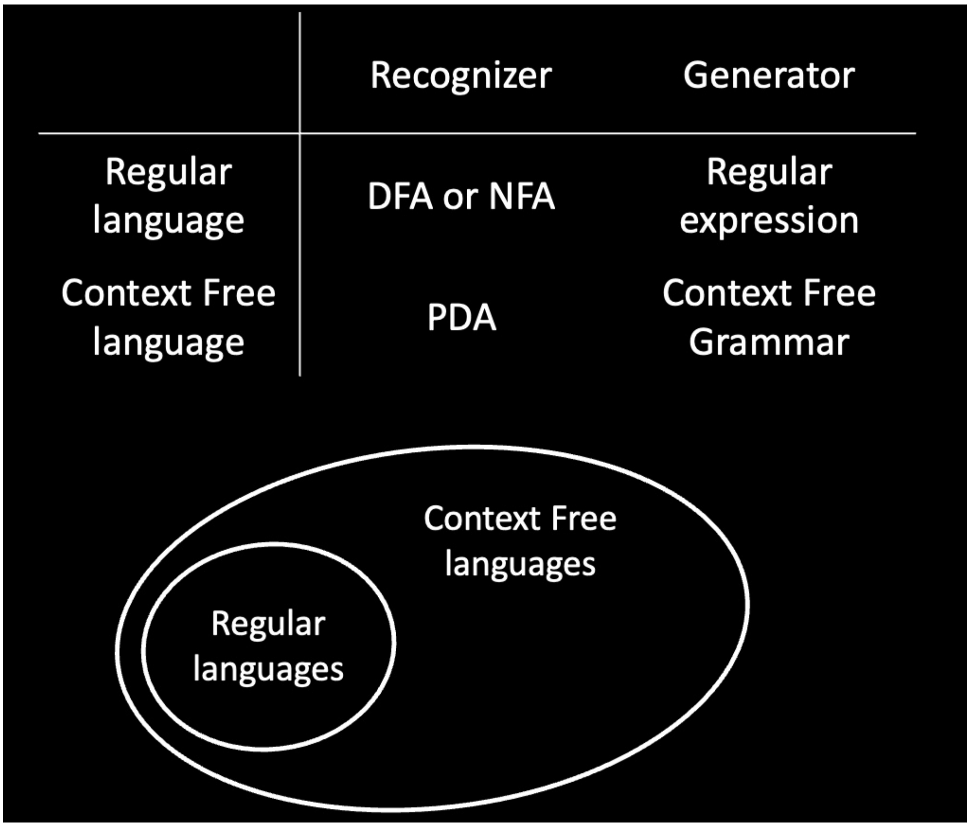
PDA to CFL. I will omit this proof as it's too complicated and not included in the course. But there's a proof for that in the book for LEMMA 2.27. ■

Because every finite automaton is automatically a pushdown automaton that simply ignores its stack, we have:

Corollary 4.2.1. Every regular language is context free.



4.2.1 Recap



	Recognizer	Generator
Regular language	DFA or NFA	Regular expression
Context Free language	PDA	Context Free Grammar

Context Free languages
Regular languages

Figure 4.2: Recap until now

Chapter 5

The CF Pumping Lemma, Turing Machines

5.1 Non-context-free languages

Based on the equivalence of CFGs and PDAs, there are some more corollaries we can see or easy to prove:

Corollary 5.1.1. Every regular language is a CFL

Corollary 5.1.2. If A is a CFL and B is regular then $A \cap B$ is a CFL.

Example. A: $A = \{a^n b^n | n \geq 0\}$ B: $B = \{(ab)^*\}$

Proof sketch: while reading the input, the finite control of the PDA for A simulates the DFA for B.

Need to notice that the class of CFLs is not closed under $\cap$ which is different from regular languages.

Example. I will show an example where the intersection of 2 CFLs is not a CFL.

$L_1 = \{a^n b^n c^m | n, m \geq 0\}$
$L_2 = \{a^m b^n c^n | n, m \geq 0\}$
$L_1 \cap \text{L2} = \{a^n b^n c^n | n \geq 0\}$

Intuitively, to recognize the intersection of L_1 and L_2 we need to compare the count of a, b, c. One stack can only handle the comparison of a and b, but we have no extra stack for comparing c.

But the class of CFLs is closed under $\cup, \circ, *$.

Note. Should try to prove its closed under union, concatenation and star!

5.1.1 CF Pumping Lemma

To prove a language is **not context free**, we need another tool:

Example. Let $B = \{0^k 1^k 2^k | k \geq 0\}$, we will show that B is not a CFL.

The tool we'll use is **Pumping Lemma for CFLs**:

Lemma 5.1.1 (Pumping Lemma for CFLs)**.** If A is a CFL, then there is a number p(the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions

1. for each $i \geq 0$, $uv^ixy^iz \in A$
2. $|vy| > 0$
3. $|vxy| \leq p$

Intuition. Proof IDEA: let s be a *very long* string in A, then the parsing tree for s must be very tall. Then the parse tree must contain some long path from the start variable to the terminal symbol at a leaf. On this long path, because of the pigeonhole principle, some variable R must repeat.

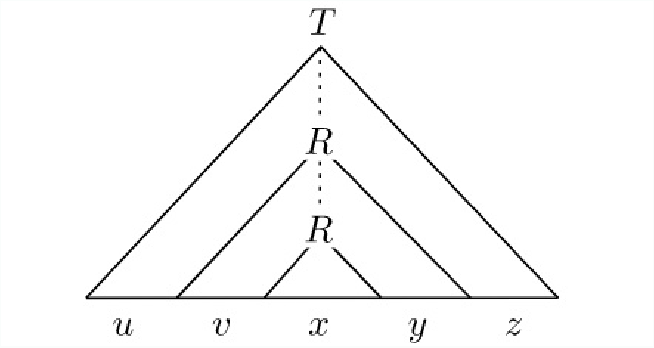


Figure 5.1: when $s = uvxyz$

This repetition allows us to replace the subtree under the second occurrence of R with the subtree under the first occurrence of R and still get a legal parse tree.

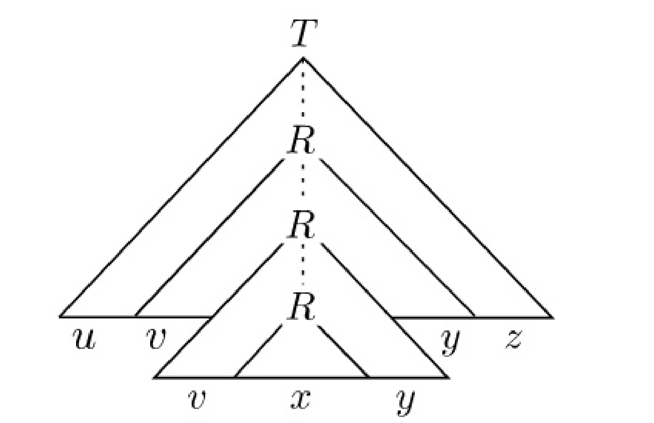


Figure 5.2: when $s = uv^ixy^iz$

Also v and y can repeat 0 time:

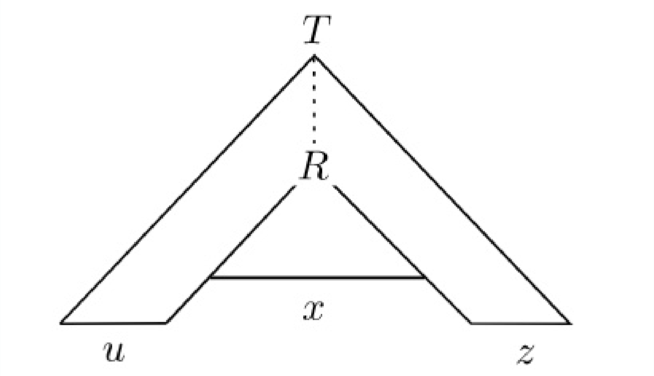


Figure 5.3: when $s = uxz$

■

Let's go back to the example:

Example. $B = \{0^k 1^k 2^k | k \geq 0\}$
Proof by Contradiction: Assume that B is a CFL.
Based on CFL pumping lemma gives p as above, Let $s = 0^p 1^p 2^p \in B$.
Pumping lemma says that can divide $s = uvxyz$ satisfying the 3 conditions.
Condition 3 ($|vxy| \leq p$) implies that vxy can not contain both 0s and 2s.
So uv^2xy^2z has unequal numbers of 0s, 1s and 2s.

Another example:

Example. Let $F = \{ww | w \in \Sigma^*\}$. $\Sigma = \{0, 1\}$.
Show: F is not a CFL.
Suppose F is a CFL, we try to construct a string which violate the CFL pumping lemma.

- Try $s_1 = 0^p 1 0^p 1$, we can choose $x = 1$ of the middle, and v and y the left and right 0, and we can pump to this string. → This is a bad choice to try to find a contradiction
- Try $s_2 = 0^p 1^p 0^p 1^p$, now we can find the contradiction!

5.2 Turing Machines(TMs)

Finite automata are good models for devices that have a small amount of memory. Pushdown automata are good models for devices that have an unlimited memory that is usable only in the last in, first out manner of stack.

We also have shown that some very simple tasks are beyond the capabilities of the models.

Alan Turing propose Turing Machine in 1936, it's similar to a finite machine but with an **unlimited and unstricted memory**. It is a much more accurate model of a general purpose computer. *A Turing machine can do anything that a real computer can do, but even a Turing machine cannot solve certain problems.*

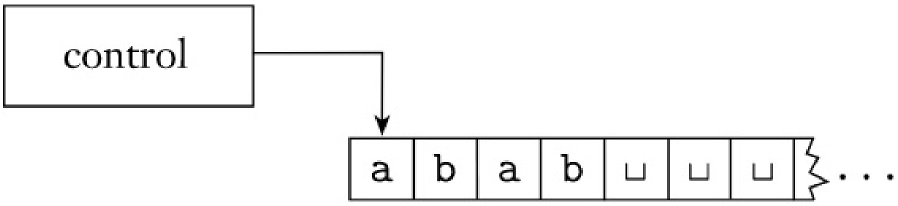


Figure 5.4: Schematic of a Turing machine

Important characteristics of it:

1. Head can read and write
2. Head is two way (can move left or right)
3. Tape is infinite (to the right)
4. Infinitely many blank follow input
5. Can accept or reject any time (not only at end of input)

Example. TM recognizing $B = a^k b^k c^k | k \geq 0$:
Example: `aaabbbccc⊔⊔`

1. Scan right until ␣ while checking if input is in $a^* b^* c^*$, *reject* if not

2. Return head to left
3. Scan right, crossing off single a, b and c: $\not{a}aa\not{b}bb\not{c}cc$
4. If the last one of each symbol, *accept*
5. If the last one of some symbol but not others, *reject*
6. If all symbols remain, return to left and repeat from step 3.

After step 6, it will go back to step 3, and the string will become $\not{a}\not{a}a\not{b}\not{b}b\not{c}\not{c}c$. Then go to step 4, they are all last one of each symbol, so we *accept*.
The effect of "crossing off" is brought by use a tape alphabet like $\Gamma = \{a, b, c, \not{a}, \not{b}, \not{c}, \sqcup\}$.

Definition 5.2.1 (Turing Machine). A Turing Machine (TM) is a 7-tuple $(Q, \Sigma, \Gamma, \sigma, q_0, q_{acc}, q_{rej})$.

- Σ input alphabet
- Γ tape alphabet ($\Sigma \subseteq \Gamma$)
- $Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ (L = left, R = right)

The transition function means that the head in a certain state q and the head is over a tape square containing a symbol a. And if $\delta(q, a) = (r, b, L)$, the machine writes the symbol b replacing a, and goes to state r. The third L means the head moves to left after writing.

On input w a TM M may halt (enter q_{acc} or q_{rej}) or M may run forever ("loop").
So M has 3 possible outcomes for each input w:

1. Accept w (enter q_{acc})
2. Reject w by halting(enter q_{rej})
3. Reject w by looping (running forever)

The above definition of TM is **deterministic**, but we can change it to be **nondeterministic** by change δ to $\delta : Q \times \Gamma \to \mathsf{P}(Q \times \Gamma \times \{L, R\})$

5.2.1 TM recognizers and deciders

Definition 5.2.2. A is Turing-recognizable if $A = L(M)$ for some TM M.

Definition 5.2.3. TM M is a decider if M halts on all inputs. (no looping, meaning eventually the machine will halt to q_{acc} or q_{rej})

Definition 5.2.4. A is Turing-decidable if $A = L(M)$ for some TM decider M.

A Turing recognizable language can cause the machine reject by looping, but a Turing decidable language can always call the TM halt, thus we have the relationship:

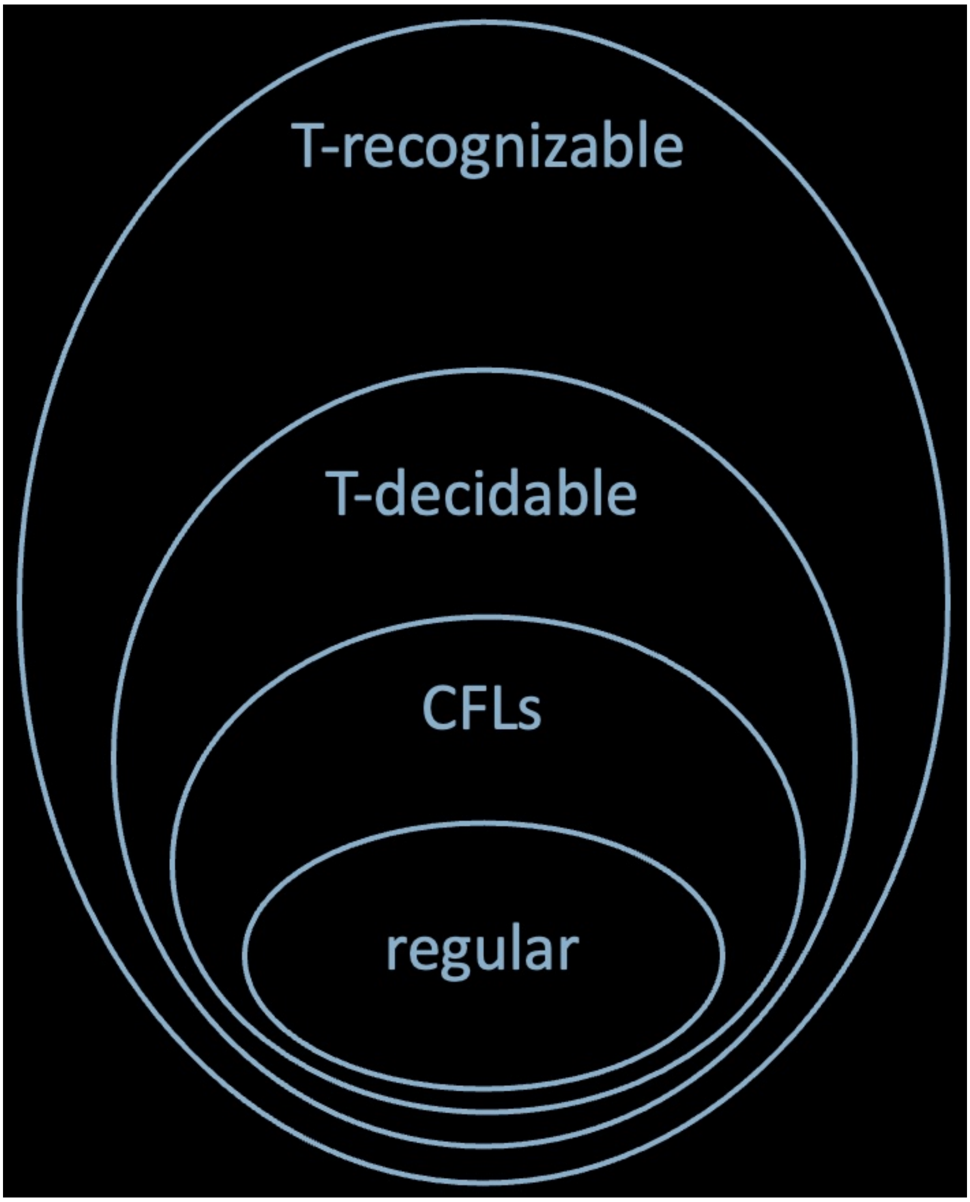


Figure 5.5: Summary of languages

Chapter 6

TM Variants, the Church-Turing Thesis

> **Note.** What kind of models are suitable for general purpose computation?

6.1 Variants of the Turing Machine Model

Turing machine is picked as for the general purpose computation, but choice doesn't matter, all reasonable models are equivalent in power.

6.1.1 Multi-tape TMs

Simple Turing machine has one tape, but we can also have multiple tapes.

One tape is for input, other tapes are work tapes, initially blank. All tapes can be read and write.

Theorem 6.1.1. A is T-recognizable iff some multi-tape TM recognizes A.

Lemma 6.1.1. A is T-recognizable then some multi-tape TM recognizes A.

Proof. Proof. This is trivial. Just use one tape. ■

Lemma 6.1.2. Multi-tape TM recognized A is also single tape recognizable.

Proof. We can simulate multi-tape TM in single-tape Turing Machine:

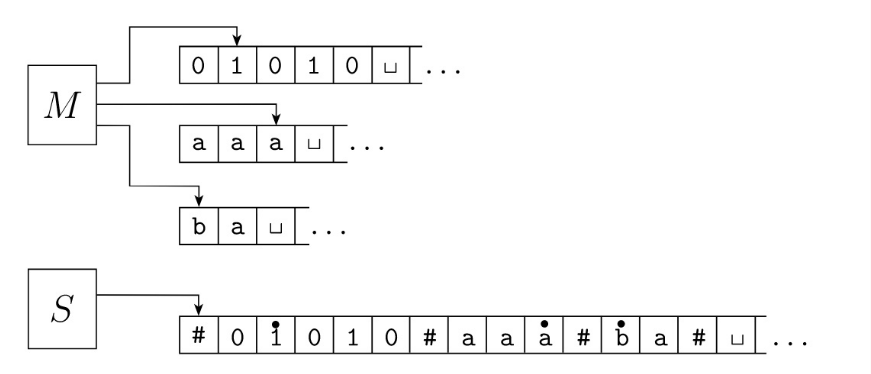


Figure 6.1: S simulates M

S simulates M by storing the contents of multiple tapes on a single tape in "block".
Record head positions with dotted symbols, something like: $b \to \dot{b}$
If M writes to originally blank space, S has to shift room as needed. ■

■

6.1.2 Nondeterministic TMs

A $NondeterministicTM$(NTM) is similar to a Deterministic TM except for its transition function $\delta : Q \times \Gamma \to \mathsf{P}(Q \times \Gamma\{L, R\})$.

Theorem 6.1.2. A is T-recognizable iff some NTM recognizes A.

Proof. (omit the trivial direction)

Lemma 6.1.3. convert NTM to deterministic TM.

The lecture and the textbook use 2 different ways to simulate, I write down the proof of the lecture as it is easier to understand.

Lecture Proof. The lecture simulates this using a single tape, just like the proof:

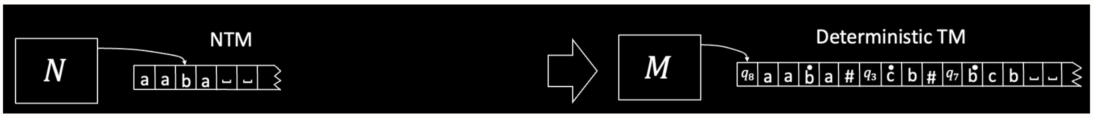


Figure 6.2: M simulates N

- M simulates N by storing each thread in separate "block"
- Also need to store the head location
- Need to store status of each thread
- If a thread forks, then M copies the block
- If a thread accepts then M accepts.

■

■

6.1.3 Enumerators

Definition 6.1.1 (informal). A Turing Enumerator is a deterministic TM with a printer. The Turing machine can use that printer as an output device to print strings. Every time the Turing machine wants to add a string to the list, it sends the string to the printer.

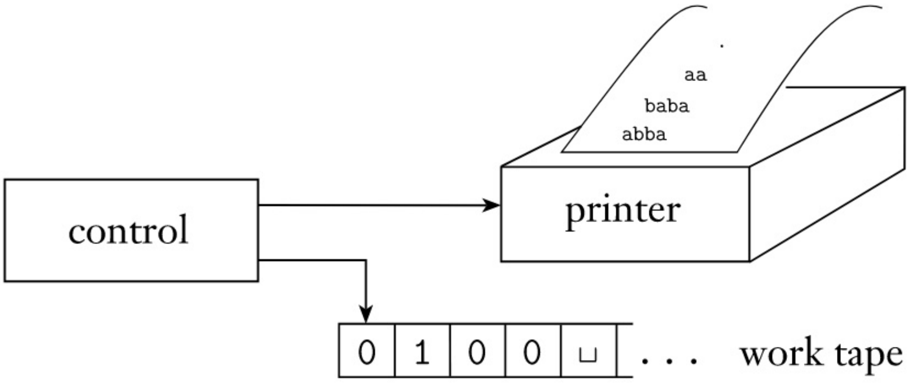


Figure 6.3: Schematic of an enumerator

The enumerator E starts with a blank input on its work tape. The language enumerated by E is the collection of all the strings that it eventually prints out.

Theorem 6.1.3. A is T-recognizable iff $A = L(E)$ for some T-enumerator E.

Lemma 6.1.4. Convert E to equivalent TM M: If we have an enumerator E that enumerates a language A, a TM M recognizes A.

Proof. Proof. The TM M works in the following way:
$M =$" On input w:

1. Run E. Every time that E outputs a string, compare it with w.
2. If w ever appears in the output of E, *accept*."

Clearly, M accepts those strings that appear on E's list. ■

Lemma 6.1.5. If TM M recognizes a language A, we can construct the following enumerator E for A.

Proof. Say that $s_1, s_2, s_3, \cdots$ is a list of all possible strings in Σ^*.
$E =$ "Ignore the input.

1. Repeat the following for $i = 1, 2, 3, \cdots$
2. Run M for i steps on each input, $s_1, s_2, \cdots, s_i$
3. If any computations accept, print out the corresponding s_j"

Intuition. This is basically the E uses the TM it attaches to recognize the language, and print if the attached TM recognizes it.

It can be an infinite loop because all possible strings in Σ^* is infinite if it does not only contain ϵ.

But when really run the TM, we have to run in parallel, as the TM might be looping.

■

■

The essential feature of TM is *unrestricted access to unlimited memory*, it distinguishes TMs from other weak models. Some other models also have the same power as TMs, and they also share the same feature.

6.2 Church-Turing Thesis

Remark. *Intuitive notion of algorithms* equals *Turing machine algorithms*

Church used a notational system called the $\lambda-$calculus to define algorithms. Turing did it with his "machines". The two definitions where shown to be equivalent. The connection between the informal notion of algorithm and the precise definition has come to be called ***Church-Turing thesis***.

It has been used by Yuri Matijasevi to solve Hilbert's 10th problem:

> to provide a general algorithm that, for any given Diophantine equation (a polynomial equation with integer coefficients and a finite number of unknowns), can decide whether the equation has a solution with all unknowns taking integer values.(Wikipeadia: Hilbert's tenth problem)

> **Note.** The idea is to prove the language of Diophantine equations is TM recognizable but not decidable.

6.3 Notation for encodings and TMs

We're going to accept more types (even automaton) as input to Turing machine. But TM only accepts strings, so we need to encode objects into strings.

- If O is some object, we write $\langle O \rangle$ to be an encoding of that object into a string.
- If $O_1, O_2, \cdots, O_k$ is a list of objects then we write $\langle O_1, O_2, \cdots, O_k \rangle$ to be an encoding of them together into a single string.

Chapter 7

Decision Problems for Automata and Grammars

We demonstrate certain problems can be solved algorithmically and others that cannot. Our objective is to explore the limits of algorithm solvability.

The importance of studying unsolvability:

1. Knowing a problem is algorithmically unsolvable *is* useful because you realize that the problem must be simplified or altered before you can find an algorithm solution.
2. Even you know the problem is solvable, a glimpse of the unsolvable can stimulate your imagination and help you gain an important perspective on computation.

> **Remark** (Computational Problem)**.** In theoretical computer science, a computational problem is one that asks for a solution in terms of an algorithm.(Wikipedia)

> **Example.** Some types of computational problem: decision problems, search problems, counting problems, optimization problems.
>
> Questions like "what is the meaning of life?" and "do I look good in this outfit" are not computational problems.
>
> (Unit 2: Computational Problems and Algorithms)

7.1 Acceptance Problem for DFAs

> **Remark.** Sometimes distinguish a machine that is looping from one that is merely taking a long time is difficult. For this reason, we prefer Turing machines that halt on all inputs, such machines never loop.
>
> Those machines are called **deciders**.

The **acceptance problem** for DFAs of testing whether a particular DFA accepts a given string can be expressed as a language, A_{DFA}. The language contains the encodings of all DFAs together with strings that the DFAs accept.

Why we are representing a pair like $\langle B, w\rangle$ as a language? Because this is convenient as we have already set up terminology for dealing with languages.

> **Theorem 7.1.1.** Let $A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a DFA and } B \text{ accepts w}\}$
>
> A_{DFA} is decidable

This problem of "testing whether a DFA B accepts an input w" is the same with "whether a combination <B, w> is a member of language A_{DFA}".

In this way, we formulate the computational problem into "testing the membership in the language".

If the language is decidable, meaning we can use a TM to decide this problem in finite time (by the definition).

This theorem tells that the computational problem "testing whether a given finite automaton accepts a given string" is decidable.

Proof. M ="On input $\langle B, w\rangle$, where B is a DFA and w is a string:

1. Simulate B on input w
2. If the simulation ends in an accept state, *accept*, If it ends in a nonaccepting state, *reject*.

" ■

7.2 Acceptance Problem for NFAs

Theorem 7.2.1. Let $A_{NFA} = \{\langle B, w\rangle \mid B$ is a NFA and B accepts w$\}$

Proof. Given TM D_{A-NFA} that decides A_{NFA}.
D_{A-NFA} = "On input $\langle B, w\rangle$

1. Convert NFA B to equivalent DFA B'
2. Run TM D_{A-DFA} on input $\langle B', w\rangle$
3. *Accept* if D_{A-DFA} accepts, *Reject* if not.

" ■

7.3 Emptiness Problem for DFAs

Theorem 7.3.1. Let $E_{DFA} = \{\ \langle B\rangle \mid B$ is a DFA and $L(B) = \emptyset\}$.
E_{DFA} is decidable.

The problem is like asking us is it possible to write an algorithm to decide if B is a super dumb DFA and does not accept any string.

Proof. Given TM D_{E-DFA} that decides E_{DFA}.
D_{E-DFA} = "On input $\langle B\rangle$

1. Mark start state
2. Repeat until no new state is marked:
   Mark every state that has an incoming arrow from a previously marked state.
3. *Accept* if no accept state is marked. *Reject* if some accept state is marked.

" ■

7.4 Equivalence Problem for DFAs

Theorem 7.4.1. Let $EQ_{DFA} = \{\ \langle A, B\rangle \mid$ A and B are DFAs and $L(A) = L(B)\}$
EQ_{DFA} is decidable.

This problem mean whether we have an algorithm can decide 2 DFAs can accept same set of strings.

Proof. Given TM D_{EQ-DFA} that decides EQ_{DFA}.
D_{EQ-DFA} = "On input $\langle A, B\rangle$ [IDEA: Make DFA C that accepts w where A and B disagree]

1. Construct DFA C where $L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$ ($L(C)$ is symmetric difference, we try to prove it is an empty set)
2. Run D_{E-DFA} on $\langle C\rangle$ (we can do that because of the empty decider)

3. *Accept* if D_{E-DFA} accepts, *Reject* if not.

" ■

7.5 Acceptance Problem for CFGs

Theorem 7.5.1. Let $A_{CFG} = \{\langle G, w\rangle | \ G$ is a CFG and $w \in L(G)\}$
$A_{CFG} is decidable$

This is the same to ask does G generate w.

Proof. The intuition is that we can check all the strings this CFG has generated and check if w is one of them. But the problem is that we need to know that this process has a bound.

To prove this we need another theorem that has been proved:

Theorem 7.5.2 (Chomsky Normal Form(CNF)). CNF only allows rules:

$$A \to BC$$
$$B \to b$$

Lemma 7.5.1. Can convert every CFG into CNF.

Lemma 7.5.2. If H is in CNF and $w \in L(H)$ then every derivation of w has $2|w| - 1$ steps.

The second lemma is important as it assures us that for any w we can using finite steps to generate it if it can be generated using the CFG.

Give TM D_{A-CFG} that decides A_{CFG}.
D_{A-CFG} = "On input $\langle G, w\rangle$

1. Convert G into CNF
2. Try all derivations of length $2|w| - 1$
3. Accept if any generate w, Reject if not

■

Corollary 7.5.1. Every CFL is decidable.

7.6 Emptiness Problem for CFGs

Theorem 7.6.1. Let E_{CFG} = { $\langle G\rangle$ | G is a CFG and $L(G) = \emptyset$}
E_{CFG} is decidable.

Proof. D_{E-CFG} = "On input $\langle G\rangle$ [IDEA: work backwards from terminals]:

1. Mark all occurrences of terminals in G
2. Repeat until no new variables are marked

   Mark all occurrences of variable A if
   $A \to B_1 B_2 \cdots B_k$ is a rule and all B_i were already marked.
3. Reject if the start variable is marked. Accept if not

■

Example (How the mark process work)**.** Suppose we have:

$$\begin{aligned} S &\to RTa \\ R &\to Tb \\ T &\to a \end{aligned}$$

Step1: we can only mark T as we know that T generates only terminals

$$\begin{aligned} S &\to RTa \\ R &\to Tb \\ T &\to a \end{aligned}$$

Step2: because of step 1, we can also mark R, because we know T is marked.

$$\begin{aligned} S &\to RTa \\ R &\to Tb \\ T &\to a \end{aligned}$$

And so on...

7.7 Equivalence Problem for CFGs

Theorem 7.7.1. Let $EQ_{CFG} = \{ \langle G, H \rangle \mid$ G, H are CFGs and $L(G) = L(H)\}$
EQ_{CFG} is **NOT** decidable.

We'll do the proof in next lecture.

Theorem 7.7.2. Let $AMBIG_{CFG} = \{ \langle G \rangle |$ G is an ambiguous CFG$\}$
$AMBIG_{CFG}$ is **NOT** decidable.

Homework.

Remark. Why we can not follow the same technique we used to show EQ_{DFA} is decidable?
Because CFLs are not closed under complementation and intersection. We can't make those symmetric difference.

7.8 Acceptance Problem for TMs

Theorem 7.8.1. Let $A_{TM} = \{ \langle M, w \rangle$ M is a TM and M accepts w$\}$
A_{TM} is **NOT** decidable.

Will be proved in next lecture

Theorem 7.8.2. A_{TM} is T-recognizable.

Proof. TM U recognizes A_TM
$U =$ "On input $\langle M, w \rangle$

1. Simulate M on input w

2. Accept if M halts and accepts

3. Reject if M halts and rejects

4. ~~Reject if M never halts~~ (not a legal TM action, because we cannot tell).

" ■

Chapter 8

Undecidability

8.1 The Diagonalization Method

Cantor (1̃890s) had the following idea:

Definition 8.1.1 (Same size of sets). Say that set A and B have the same size if there is a one-to-one("injective") and **onto**("surjective") function: $f : A \to B$.

Example (Countable Sets). Let $N = \{1, 2, 3, \cdots\}$ and let $Z = \{\cdots, -2, -1, 0, 1, 2, \cdots\}$. Show N and Z have the same size.

Example (Countable Sets). Let $Q^+ = \{m/n | m, n \in N\}$, show N and Q^+ has the same size.

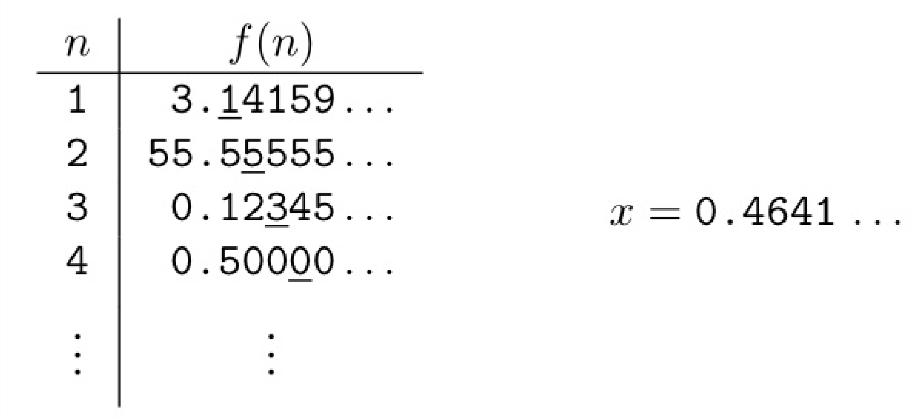
Definition 8.1.2 (Countable). A set A is countable if either it is finite or it has the same size as $\mathbb{N}$.

Theorem 8.1.1 ($\mathbb{R}$ is Uncountable). Let $\mathbb{R}$ = all real numbers

Proof. Proof by contradiction via diagonalization: Assume R is countable.

So there is a 1-1 correspondence $f : N \to R$

Demonstrate a number $x \in R$ that is missing from the list. It has i-th digit different from the i-th number in the list.



n	$f(n)$
1	$3.\underline{1}4159\ldots$
2	$55.5\underline{5}555\ldots$
3	$0.12\underline{3}45\ldots$
4	$0.500\underline{0}0\ldots$
$\vdots$	$\vdots$

$x = 0.4641\ldots$

Figure 8.1: Diagonalization

■

Let L = all languages, we have some corollaries:

Corollary 8.1.1. L is uncountable.

Proof. There's a 1-1 correspondence form L to R so they are the same size.

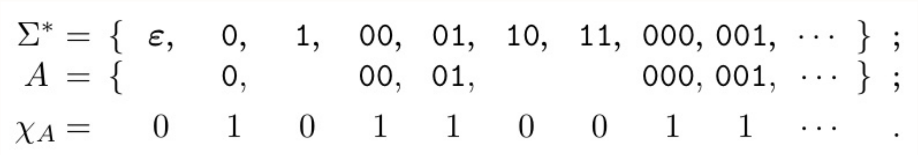
> **Remark** (Observation). $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \cdots\}$ is countable.
> For each length, there are only finite numbers of strings.

> **Remark** (Observation). The set of all Turing machines is countable because each Turing machine M has an encoding into a string $\langle M \rangle$.
>
> **Why each M can encode?**: Because TM is a finite object with a well-defined structure.
>
> > A Turing machine M can be described as a 7-tuple $Q, F, q_0, \Sigma, \Gamma, \delta, blank$. This means that if someone gives you this 7-tuple, then the TM is well-defined, and you can precisely define how it behaves, etc. (CSStackExchange)
>
> **Why this mean M is countable?**: This is my thought, because each TM can be encoded into a string. And because the encoding alphabet of the string is limited, for the same reason why Σ^* is countable, this is also countable.

Theorem 8.1.2. The set of all infinite binary sequences is uncountable.

Proof. This is easy to be proved by using diagonalization. ■

Let $\mathcal{B}$ be the set of all infinite binary sequences. We can show that the set of all languages $\mathcal{L}$ has the same size of $\mathcal{B}$:



$$
\begin{array}{rlllllllllll}
\Sigma^* = \{ & \varepsilon, & 0, & 1, & 00, & 01, & 10, & 11, & 000, & 001, & \cdots & \} ; \\
A = \{ & & 0, & & 00, & 01, & & & 000, & 001, & \cdots & \} ; \\
\chi_A = & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \cdots & .
\end{array}
$$

For each kind of language A, according to whether a string exists in it, we can generate a bitmap $\mathcal{X}_A$ according to that. ■

Corollary 8.1.2. Some language is not decidable

Proof. There are more languages than Turing Machine! ■

> **Remark.** Consider Hilbert's 1st question.

8.2 Undecidable Language

Theorem 8.2.1. $A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$
A_{TM} is not decidable.

Proof. Assume some TM H decides A_{TM}.
So H on $\langle M, w \rangle =$

$$
\begin{array}{l}
Accept\ if\ M\ accepts\ w \\
Reject\ if\ not
\end{array}
$$

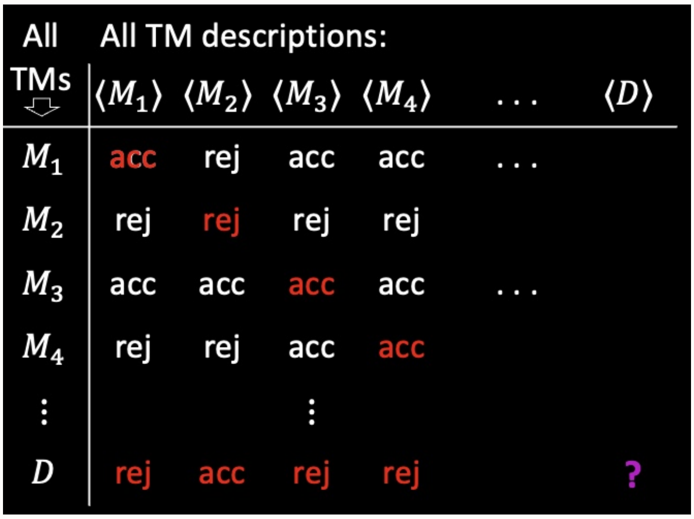
Use H to construct TM D
D = "On input $\langle M \rangle$

1. Simulate H on input $\langle M \langle M \rangle \rangle$

2. *Accept* if H rejects, *Reject* if H accepts."

D accepts $\langle M \rangle$ iff M doesn't accept $\langle M \rangle$.
D accepts $\langle D \rangle$ iff D doesn't accept $\langle D \rangle$.
Contradiction.

Where is the diagonalization?



All TMs	All TM descriptions: $\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	...	$\langle D \rangle$
M_1	acc	rej	acc	acc	...	
M_2	rej	rej	rej	rej		
M_3	acc	acc	acc	acc	...	
M_4	rej	rej	acc	acc		
⋮			⋮			
D	rej	acc	rej	rej		?

Figure 8.2

■

This implies that there is no TM that can always decide whether an arbitrary TM M will accept an input w.

Theorem 8.2.2. If A and $\overline{A}$ are T-recognizable then A is decidable.

This theorem connects decidability and recognizability.

Proof. Let TM M_1 and M_2 recognize A and $\overline{A}$.
Construct TM T deciding A.
T = "On input w

1. Run M_1 and M_2 on w in parallel until one accepts.

2. If M_1 accepts then *accept*, if M_2 accepts then reject. "

■

Corollary 8.2.1. $\overline{A_{TM}}$ is T-unrecognizable.

Proof. A_{TM} is T-recognizable but also undecidable (as we have proved here). If $\overline{A_{TM}}$ is also recognizable, then A_{TM} should be decidable. ■

Here is the relationship:

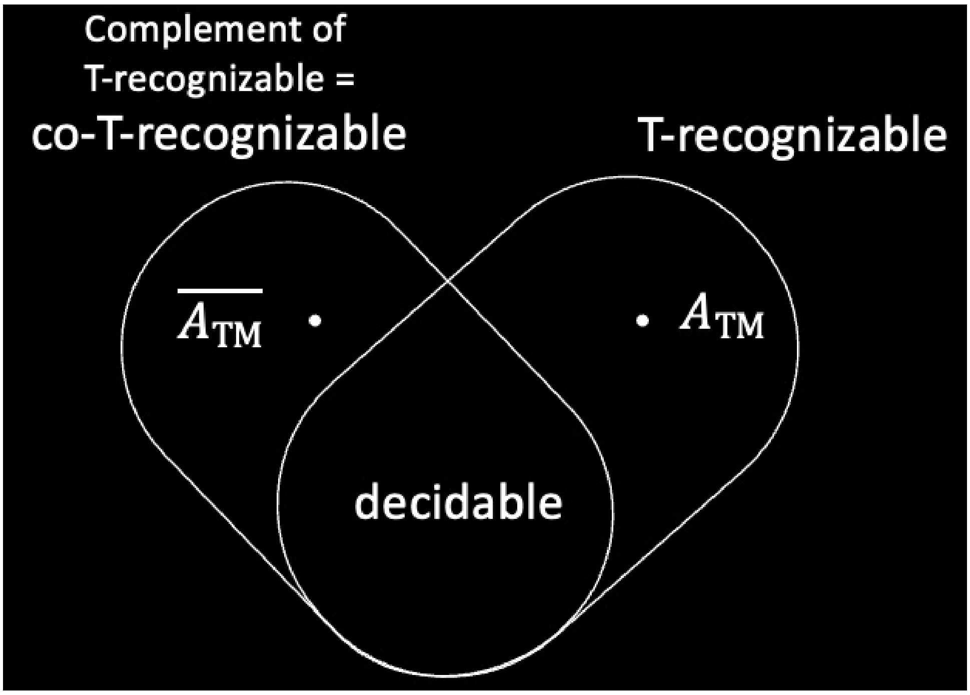


Figure 8.3

8.3 The Reducibility Method

Use our knowledge that A_{TM} is undecidable to show other problems are undecidable.

Definition 8.3.1. $HALT_{TM} = \{ \langle M, w \rangle \mid M$ halts on input $w \}$

Theorem 8.3.1. $HALT_{TM}$ is undecidable.

Proof. Assume that $HALT_{TM}$ is decidable and show that A_{TM} is decidable (false!)
A_{TM} is undecidable has been proved in here.
Let TM R decide $HALT_{TM}$.
Construct TM S deciding A_{TM}.
$S =$ "On input $\langle M, w \rangle$

1. Use R to test if M on w halts, if not, reject.
2. Simulate M on w until it halts (as guaranteed by R)
3. If M has accepted then *accept*, if M has rejected then *reject*."

TM S decides A_{TM}, a contradiction. Therefore $HALT_{TM}$ is undecidable. ■

Chapter 9

Reducibility

9.1 The Reducibility Method

If we know that some problem (say A_{TM}) is undecidable, we can use that to show other problems are undecidable.

Remark. Review the proof of HALT problem.

Definition 9.1.1 (Reducibility). If we have 2 languages (or problems) A and B, then A is reducible to B means that we can use B to solve A.

Example. Measuring the area of a rectangle is reducible to measuring the lengths of its sides.

Example. We showed that A_{NFA} is reducible to A_{DFA}.

If A is reducible to B then solving B gives a solution to A:

- then B is easy → A is easy.
- $\boxed{\text{then } A \text{ is hard} \rightarrow B \text{ is hard.}}$ (this is the form we will use)

9.2 General Reducibility

Theorem 9.2.1. Let $E_{TM} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset \}$
E_{TM} is undecidable.

Proof. Proof by contradiction. Show that A_{TM} is reducible to E_{TM}.
Assume that E_{TM} is decidable and show that A_{TM} is decidable.
Let TM R decide E_{TM}
Construct TM S deciding A_{TM}

Remark. A naive idea will be constructing S to run R on $\langle M \rangle$. But here is the problem, if R accepts $\langle M \rangle$, meaning M accepts no string, that is $L(M) = \emptyset$, this is good. But if R reject $\langle M \rangle$, we know M must accept some language, but we don't know what language.
That's why we need a modified M.

$S =$ "On input $\langle M, w \rangle$

1. We can use M and w to construct a TM M_w.

   $M_w =$ "On input x

   (a) If $x \neq w$, reject

(b) If $x \neq w$, run M on input w and accept if M does."

2. Run R on input $\langle M_w \rangle$

3. If R accepts, *reject*; if R rejects, *accept*."

■

9.3 Mapping Reducibility

Definition 9.3.1. A function $f : \Sigma^* \to \Sigma^*$ is a computable function if there is a TM F where F on input w halts with $f(w)$ on its tape for all strings w.

> **Note** (rephrase). $f : \Sigma^* \to \Sigma^*$ is **computable (or Turing computable)** if there is some TM M and for all w, M will take w as input, and write $f(w)$ on the tape.

Example. Arithmetic operations on integers are computable functions.
For example, we can make a machine that takes input $\langle m, n \rangle$ and returns $m + n$.

Definition 9.3.2. A is mapping-reducible to $B(A \leq_m B)$ **if there is** a computable function f where $w \in A$ iff $f(w) \in B$.

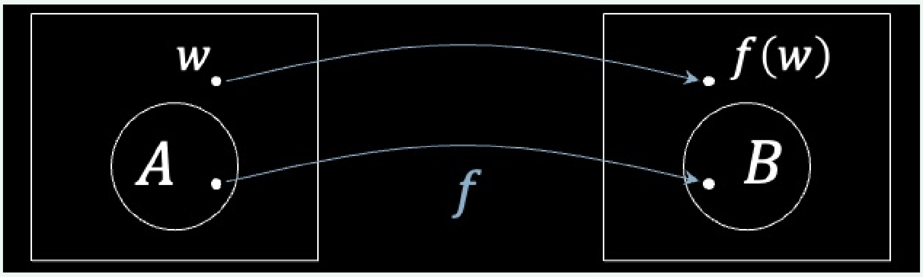


Figure 9.1: Function f reducing A to B

Example. $A_{TM} \leq_m \overline{E_{TM}}$
The computable reduction function f is $f(\langle M, w \rangle) = \langle M_w \rangle$

> **Remark.** Recall TM M_w = "On input x
>
> 1. if $x \neq w$, reject
> 2. else run M on w
> 3. Accept if M accepts"

Because $\langle M, w \rangle \in A_{TM}$ iff $\langle M_w \rangle \in \overline{E_{TM}}$ (M accepts w iff $L(\langle M_w \rangle) \neq \emptyset$)

9.3.1 Mapping Reductions - properties

Theorem 9.3.1. If $A \leq_m B$ and B is decidable then so is A

The theorem means there is a computable function (reduction function) which takes any input and produces an output where the input in A iff the output is in B.

Proof. Say TM R decides B.
Construct TM S deciding A: S = "On input w

1. Compute $f(w)$
2. Run R on $f(w)$ to test if $f(w) \in B$

   > **Remark.** Because R decides B, then R is a decider. A TM is a decider means this TM will halt on every input, no matter if the input is in B.

3. If R halts then output the same result."

■

> **Remark.** According to this theorem, we can conclude that for this example, if $\overline{E_{TM}}$ is decidable, then A_{TM} is decidable. Because we all know that A_{TM} is not decidable, so $\overline{E_{TM}}$ is not decidable.

Corollary 9.3.1. If $A \leq_m B$ and A is undecidable then so is B

Theorem 9.3.2. If $A \leq_m B$ and B is T-recognizable then so is A.

Proof. Same as above ■

Corollary 9.3.2. If $A \leq_m B$ and A is T-unrecognizable then so is B.

Theorem 9.3.3. $A \leq_m B \Leftrightarrow \overline{A} \leq_m \overline{B}$

Proof. This is because $A \leq_m B$ implies if there is an x in A, iff $f(x)$ is in B.
So if x not in A, only if $f(x)$ is not in B. Which is the complement form. ■

Corollary 9.3.3. For the same reason, we know that $A_{TM} \not\leq_m E_{TM}$, so we can say $\overline{A_{TM}} \not\leq \overline{E_{TM}}$.

9.3.2 Mapping vs General Reducibility

They are not the same!

Mapping Reducibility of A to B: Translate A-questions to B-questions

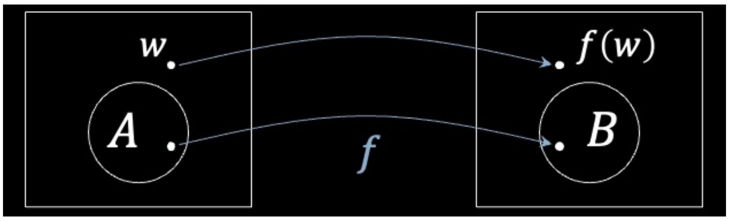


Figure 9.2: Showing Mapping Reducibility

- A special type of reducibility.
- Useful to prove T-unrecognizability.

(General) Reducibility of A to B: Use B solver to solve A

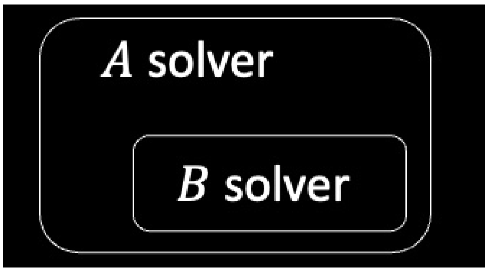


Figure 9.3: Showing General Reducibility

- May be conceptually simpler
- Useful to prove Undecidability

Noteworthy difference:

1. A is reducible to $\overline{A}$
2. A may not be mapping reducible to $\overline{A}$

> **Example.** $\overline{A_{TM}} \not\leq_m A_{TM}$

9.3.3 Reducibility - Templates

To prove B is undecidable:

- Show undecidable A is reducible to B (often A is A_{TM})
- Template: Assume TM R decides B, Construct TM S deciding A. Contradiction.

To prove B is T-unrecognizable:

- Show T-unrecognizable A is mapping reducible to B. (often A is $\overline{A_{TM}}$).
- Template: give reduction function f

Theorem 9.3.4. Recall $E_{TM} = \{ \langle M \rangle | \; M$ is a TM and $L(M) = \emptyset\}$
E_{TM} is T-unrecognizable

Proof. (We have already shown it is undecidable)
Show $\overline{A_{TM}} \leq_m E_{TM}$
Reduction function $f(\langle M, w\rangle) = \langle M_w \rangle$.

> **Remark.** M_w is the same machine we have constructed before.

■

Theorem 9.3.5. $EQ_{TM} = \{ \langle M_1, M_2 \rangle | \; M_1$ and M_2 are TMs and $L(M_1) = L(M_2)$ \}
Both EQ_{TM} and $\overline{EQ_{TM}}$ are T-unrecognizable

Proof.

1. $\overline{A_{TM}} \leq_m EQ_{TM}$
2. $\overline{A_{TM}} \leq_m \overline{EQ_{TM}}$
For any w let T_w = "On input x

1. ignore x
2. Simulate M on w"

1. $f(\langle M, w\rangle) = \langle T_w, T_{reject}\rangle$ (T_{reject} is a TM that always rejects)
2. $f(\langle M, w\rangle) = \langle T_w, T_{accept}\rangle$ (T_{accept} is a TM that always accepts)

■

9.4 Problems

Problem 9.4.1 (Rice's theorem). Let P be any nontrivial property of the language of a Turing machine. Prove that the problem of determining whether a given Turing machine's language has property P is undecidable.

In more formal terms, let P be a language consisting of Turing machine descriptions where P fulfills two conditions. First, P is nontrivial – it contains some, but not all, TM descriptions. Second, P is a property of the TM's language – whenever $L(M_1) = L(M_2)$, we have $\langle M_1\rangle \in P \Leftrightarrow \langle M_2\rangle \in P$. Here, M_1 and M_2 are any TMs. Prove that P is an undecidable language.

Chapter 10

The Computation History Method

Remark (Remember)**.** To prove some language B is undecidable, show that A_{TM} (or any known decidable language) is reducible to B.

10.1 Computation History

Definition 10.1.1 (TM configuration)**.** A configuration of a TM is a triple (q, p, t) where

- $q =$ the state,
- $p =$ the head position,
- $t =$ tape contents

representing a snapshot of the TM at a point in time.

Remark (Encode a configuration)**.** Encode configuration (p, q, t) as string $t_1 q t_2$ where $t = t_1 t_2$ and the head position is on the first symbol of t_2.

Example. Configuration: $(q_3, 6, aaaaaabbbbb)$
Encoding as string: $aaaaaq_3abbbbb$

Definition 10.1.2 (TM Computation History)**.** An (accepting) computation history for TM M on input w is a sequence of configurations $C_1, C_2, \cdots, C_{accept}$ that M enters until it accepts.

Intuition. It's like the log of the execution of the machine.

Remark (Encode a computation history)**.** Encode a computation history as $C_1 \# C_2 \# \cdots \# C_{accept}$ where each configuration is encoded as a string.

10.2 Linearly Bounded Automata

Definition 10.2.1 (Linealy Bounded Automata(LBA))**.** A linearly bounded automata (LBA) is a 1-tape TM that cannot move its head off the input portion of the tape.

Remark. Tape size adjusts to the length of input.

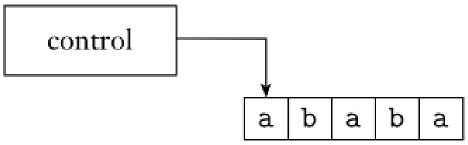


Figure 10.1: Schematic of an LBA

Theorem 10.2.1. Let $A_{LBA} = \{\langle B, w\rangle | LBA\ B\ accepts\ w\}$
A_{LBA} is decidable.

Proof. (idea) If B on w runs for long, it must be cycling.

> **Remark.** For inputs of length n, an LBA can have only $|Q| \times n \times |\Gamma|^n$ different configurations.
> Therefore, if an LBA runs for longer, it must repeat some configurations and thus will never halt.

■

Theorem 10.2.2. Let $E_{LBA} = \{\ \langle B\rangle\ |\ B$ is an LBA and $L(B) = \emptyset\ \}$
E_{LBA} is undecidable.

Proof. Show A_{TM} is reducible to E_{LBA}.
Assume R decides E_{LBA}, we can construct a TM S deciding A_{TM} (which is contradiction):
$S =$ "on input $\langle M, w\rangle$

1. Construct LBA $B_{M,w}$ which tests whether its input x is an accepting computation history for M on w, and only accepts x if it is.
2. Use R to determine whether $L(B_{M,w}) = \emptyset$.
3. Accept if no. Reject if yes."

Why we can construct an LBA $B_{M,w}$ like that? Because for the LBA, input x is the content on the tape, the LBA can move its header on the tape and follow this history and verify if it is a valid history of input w. ■

10.3 History Method for proving undecidability

Example (Hilbert's 10^{th} Problem). Recall $D = \{\ \langle p\rangle\ |$ polynomial $p(x_1, x_2, \cdots, x_k) = 0$ has integer solution$\}$

Theorem 10.3.1. Hilbert's 10^{th} problem: Is D decidable? No.

Proof. Show A_{TM} is reducible to D. ■

Do toy problem instead which has a similar proof method: PCP. The method is The Computation History Method.

10.4 Post Correspondence Problem (PCP)

Problem 10.4.1 (PCP problem). Give a collection of pairs of strings as dominoes:

$$P = \{\begin{bmatrix} t_1 \\ b_1 \end{bmatrix}, \begin{bmatrix} t_2 \\ b_2 \end{bmatrix}, \cdots, \begin{bmatrix} t_k \\ b_k \end{bmatrix}\}$$

A match is a finite sequence of dominoes in P (repeats allowed) where the concatenation of the $t's$ = the concatenation of the $b's$.

Example. Match $= \begin{bmatrix} t_{i_1} \\ b_{i_1} \end{bmatrix} \begin{bmatrix} t_{i_2} \\ b_{i_2} \end{bmatrix} \cdots \begin{bmatrix} t_{i_3} \\ b_{i_3} \end{bmatrix}$ where $t_{i_1} t_{i_2} \cdots t_{i_l} = b_{i_1} b_{i_2} \cdots b_{i_l}$

Example.

$$P = \{ \begin{bmatrix} ab \\ aba \end{bmatrix} \begin{bmatrix} aa \\ aba \end{bmatrix} \begin{bmatrix} ba \\ aa \end{bmatrix} \begin{bmatrix} abab \\ b \end{bmatrix} \}$$

Match:

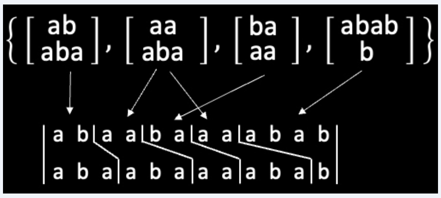


Figure 10.2: A match

Problem: Given P, is there a match?

Theorem 10.4.1 (PCP problem)**.** The above problem is undecidable. (there's no problem to solve the problem)

Formalize the problem:

Let $PCP = \{ \langle P \rangle | \ P \text{ has a match} \}$

PCP problem. IDEA: Show A_{TM} is reducible to PCP.

The main technique is reduction from A_{TM} via accepting computation histories.

How to do it? We try to make a connection that for any TM M and input w, we can construct an instance P:

$\boxed{\text{P has a match} \Rightarrow \text{An accepting computation history for } M \text{ on } w.}$

We will give some assumptions to make the proof simpler, and we can eliminate those assumptions later.

Problem 10.4.2 (Modified PCP (MPCP))**.** We modify the PCP to require that a match starts with the first domino, $\begin{bmatrix} t_1 \\ b_1 \end{bmatrix}$.

$MPCP = \{ \langle P \rangle \mid P$ is an instance of the PCP with a match that starts with the first domino$\}$

Proof. We are given M and w, we're trying to make a set of dominoes, where a match will be a computation history for M on w.

How we make our dominoes?

Part 1: Starting domino:

$$\begin{bmatrix} u_1 \\ v_1 \end{bmatrix} = \begin{bmatrix} \# \\ \# q_0 w_1 \cdots w_n \# \end{bmatrix}$$

■

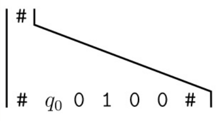


Figure 10.3: With Part 1 domino

Cont. **Part 2**: For each $a, b \in \Gamma$(alphabet in tape) and $q, r \in Q$(states) where our transition function be $\delta(q, a) = (r, b, R)$(If the TM in state q, and the head is a, it moves to state r, write b into it and its head move one step to the right). This information can be captured by domino:

$$\begin{bmatrix} q & a \\ b & r \end{bmatrix}$$

Part 3: For every $a \in \Gamma$, we put this domino:

$$\begin{bmatrix} a \\ a \end{bmatrix}$$

Part 4: We also need to have $\#$, it is not in Γ, so I make it different part:

$$\begin{bmatrix} \# \\ \# \end{bmatrix}$$

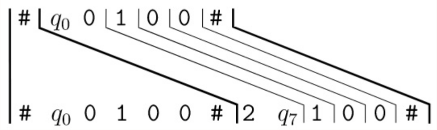


Figure 10.4: With Part 2, 3, 4 dominoes

When we're adding more states, the bottom will have q_{accept}, it is done for the perspective of the M; but it is not a match yet! Notice that the bottom is like always a configuration ahead of the top.

Part 5: To make the match, we add dominoes like these:

$$\begin{bmatrix} aq_{accept} \\ q_{accept} \end{bmatrix} \quad \begin{bmatrix} q_{accept}a \\ q_{accept} \end{bmatrix}$$

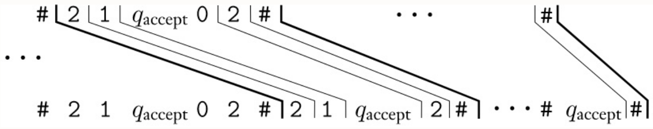

# 2 1 qaccept 0 2 # ... #
...
# 2 1 qaccept 0 2 # 2 1 qaccept 2 # ... # qaccept #

Figure 10.5: With Part 5 dominoes

Part 6: To end, we add:

$$\begin{bmatrix} q_{accept}\#\# \\ \# \end{bmatrix}$$

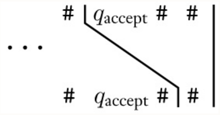


Figure 10.6: With Part 6 domino

■

Remark. Why we can have consecutive #s? Because there are ϵ.

Now we assume that TM R decides PCP.
We construct TM S deciding A_{TM}:
$S =$ "on input $\langle M, w\rangle$

Cont. Construct PCP instance $P_{M,w}$(what we have done!) where a match corresponds to a computation history for M and w.

2. Use R to determine whether $P_{M,w}$ has a match.

3. Accept if yes, Reject if no."

■

■

Theorem 10.4.2. Let ALL_{CFG} ={ $\langle G\rangle \mid G$ is a CFG and $L(G) = \Sigma^*$ }
ALL_{CFG} is undecidable.

Proof. Show A_{TM} is reducible to ALL_{PDA} via the computation history method.
Assume TM R decides ALL_{PDA} and construct TM S deciding A_{TM}.
$S =$ "On input $\langle M, w\rangle$

1. Construct PDA $B_{M,w}$ which tests whether its input x is an accepting computation history for M and w, and only accepts x if it is NOT

2. Use R to determine whether $L(B_{M,w}) = \Sigma^*$

3. Accept if no, Reject if yes."

■

Chapter 11

The Recursion Theorem and Logic

11.1 Self-reproduction Paradox

Suppose a Factory makes cars, meaning complexity of factory is greater than complexity of cars. But can a factory makes factories?

Similarly, can a program print itself?(YES!)

11.1.1 A self-reproducing TM

Theorem 11.1.1. There is a TM $SELF$ which (on any input) halts with $\langle SELF \rangle$ on the tape.

Lemma 11.1.1. There is a computable function $q : \Sigma^* \to \Sigma^*$ such that $q(w) = \langle P_w \rangle$ for every w, where P_w is the TM $P_w =$ "Print w on the tape and halt".

($q(w)$ is the description of a TM P_w who prints w and halts.)

Note. Computable function f means if there is a TM M for all input w, it halts with $f(w)$ on the tape.

Proof. Straightforward, we can apparently make such kind of function. ■

Proof. $SELF$ has 2 parts, A and B.

$A = P_{\langle B \rangle}$, which makes A can print a $\langle B \rangle$ on the tape:

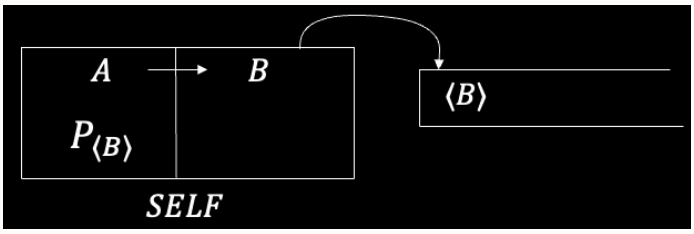


Figure 11.1: A print $\langle B \rangle$ on the tape

Intuition. Can we also do the same in B and makes it print A?

No! Because this is circular reasoning, A contains the serialization form of B, if B contains the serialization form of A, then the machine will be infinite large.

$B =$

1. "Compute $q(\textit{tape contents})$ to get A (The tape content happens to be $\langle B \rangle$, but it does not care)

2. Combine with B to get $AB = SELF$

3. Halt with $\langle SELF \rangle$ on tape."

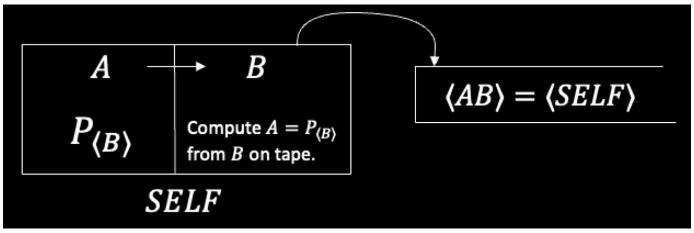


Figure 11.2: Print $\langle AB \rangle$ on the tape

■

Example (English Implementation). Write "Hello World":
$>$ `Hello World`
Write the following twice, the second time in quotes "Hello World":
$>$ `Hello World "Hello World"`
Write the following twice, the second time in quotes
"Write the following twice, the second time in quotes":
$>$ `Write the following twice, the second time in quotes`
$>$ `"Write the following twice, the second time in quotes"`

The upper phrase of the English example is called Action part, the B in the TM is the Action part.

11.2 The Recursion Theorem

Theorem 11.2.1 (Compute your own description). Let T be a TM that computes a function $t : \Sigma^* \times \Sigma^* \to \Sigma^*$. There is a TM R that computes a function $r : \Sigma^* \to \Sigma^*$, where for every w,

$$r(w) = t(\langle R \rangle, w)$$

Remark. r works exactly like t except r provides the description of R.

Proof. R has 3 parts: A, B and T.
T is given.
$A = P_{\langle BT \rangle}$

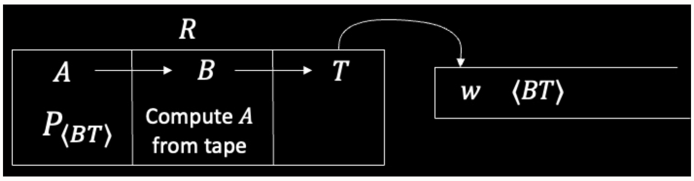


Figure 11.3: Print $\langle BT \rangle$ on the tape

$B =$

1. "Compute $q(tape\, contents\, after\; w)$ to get A

2. Combine with BT to get $ABT = R$

3. Pass control to T on input $\langle w, R\rangle$."

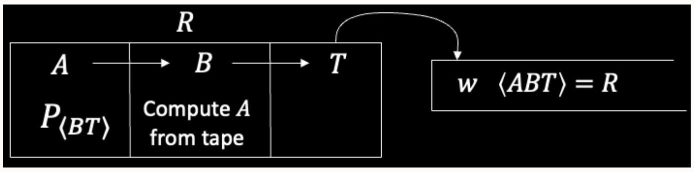


Figure 11.4: Print $\langle ABT\rangle$ on the tape

■

Note. The recursion theorem provides the ability to implement the self-referential *this* into any programming language. With it, any program has the ability to refer to its own description, which has certain applications.

11.2.1 A_{TM} is undecidable - new proof

A_{TM} is not decidable is a very important conclusion, we used it to prove halting theorem.

Theorem 11.2.2. A_{TM} is not decidable.

Proof. Assume some TM H decides A_{TM}.

Consider the following TM R: $R =$ "On input w

1. Get own description $\langle R\rangle$ (This is the content of the recursion theorem!)
2. Use H on input $\langle R, w\rangle$ to determine whether R accepts w (This is the definition of A_{TM}, our assumption is that R can decide whether any TM can accept any input)
3. Do the opposite of what H says"

Suppose H reject $\langle R, w\rangle$, R will accept the input w.
We can rephrase this into: if R reject w, R will accept w: contradiction! ■

11.2.2 Fixed-point Theorem

A **fixed point** of a function is a value that isn't changed by the application of the function. In this case, we consider functions as computable transformations of TM descriptions. We show that for any such transformation, some TM exists whose behavior is unchanged.

Theorem 11.2.3. For any computable function $f : \Sigma^* \to \Sigma^*$, there is a TM R such that $L(R) = L(S)$ where $f(\langle R\rangle) = \langle S\rangle$.

In other words, consider f to be a program transformation function. Then for some program R, its behavior is unchanged by f.

In this theorem, f plays the role of transformation, R is the fixed point.

Proof. Let R be the following TM.

$R =$ "On input w

1. Get own description $\langle R\rangle$ (recursion theorem)
2. Compute $f(\langle R\rangle)$ and call the result $\langle S\rangle$
3. Simulate S on w."

Here we can see $\langle F\rangle$ and $f(\langle F\rangle) = \langle G\rangle$ describe equivalent TM.

Note. The step 2, why we can assume $f(\langle R\rangle)$ is a valid description of another?
By the definition of total computable function.

Note. I am not very satisfied with this proof, I still feel a gap why we can say the simulated is equivalent with the original.

■

11.2.3 MIN_{TM} is T-unrecognizable

Definition 11.2.1. M is a minimal TM if $|\langle M'\rangle| < |\langle M\rangle| \to L(M') \neq L(M)$.

Remark. There is a shortest encoding form of TM, there's no shorter TM can do the same thing.

Theorem 11.2.4. Let $MIN_{TM} = \{ \langle M\rangle | M$ is a minimal TM$\}$, MIN_{TM} is T-unrecognizable.

Proof. Assume some TM E enumerates MIN_{TM}.
Consider the following TM R: $R =$ "On input w

1. Get own description $\langle R\rangle$
2. Run E until some TM B appears, where $|\langle R\rangle| < |\langle B\rangle|$
3. Simulate B on w."

Thus $L(R) = L(B)$ and $|\langle R\rangle| < |\langle B\rangle|$ so B is not minimal.
But we assume E enumerates MIN_{TM}, so contradiction.

Note. Why there must be TM B has encoding form larger than E?

■

Other applications:

1. Computer viruses
2. A true but unprovable Mathematical statement due to Kurt Godel:
   "This statement is unprovable."

11.3 Intro to Mathematical Logic

Definition 11.3.1 (Goal). A mathematical study of mathematical reasoning itself. (Formally defines the language of mathematics, mathematical truth, and provability.)

Theorem 11.3.1 (Godel's First Incompleteness Theorem). In any reasonable form system, some true statements are not provable.

Proof. We use 2 properties of formal proofs:

1. Soundness: If ϕ has a proof π then ϕ is true.
2. Checkability: The language $\{\langle \pi, \phi\rangle | \pi$ is a proof of statement $\phi\}$ is decidable.

Checkability implies the set of provable statements $\{ \langle \phi\rangle | \phi$ has a proof$\}$ is T-recognizable.

Similarly, if we can always prove $\langle M, w \rangle \in \overline{A_{TM}}$ when it is true, then $\overline{A_{TM}}$ is T-recognizable. (false!)

Therefore, some true statements of the form $\langle M, w \rangle \in \overline{A_{TM}}$ are unprovable.

Next, we use the recursion theorem to give a specific example of a true but unprovable statement.

■

Chapter 12

Time Complexity

12.1 Intro to Complexity Theory

- Computability theory (1930s - 1950s):

  Is A decidable?

- Complexity theory (1960s - present):

  Is A decidable with restricted resources? (time/memory/...)

In complexity theory, we concern only about decidable language, but the question is how?

Example. Let $A = a^k b^k | k \geq 0$
Q: How many steps are needed to decide A?
Depend on input.
We give an upper bound for all inputs of length n. (worst case complexity)

Theorem 12.1.1. A 1-tape TM M can decide A where, on inputs of length n, M uses at most cn^2 steps, for some fixed constant c.

Remark (Terminology). M uses $O(n^2)$ steps.

Proof. $M =$ "On input w

1. Scan input to check if $w \in a^*b^*$, reject if not
2. repeat until all crossed off

   Scan tape, crossing off one a and one b Reject if only a's or only b's remain
3. Accept if all crossed off."

Note. This algorithm has been discussed when introducing TM.

In step 2, we have $O(n)$ iterations and each iteration has $O(n)$ steps.

Note. Can we do better than $O(n^2)$ using 1-tape TM?
We can do $O(nlogn)$.

■

Theorem 12.1.2. A 1-tape TM M can decide A by using $O(nlogn)$ steps.

Proof. $M =$ "On input w

1. (The same as last proof, check alphabet)
2. Repeat until all crossed off

   Scan tape, crossing off every other a and b

   Reject if even/odd parities disagree.
3. Accept if all crossed off"

> **Note.** If we can keep record of parities, but why we cannot record the count?
> We can, but keeping count also takes steps. We can store the parities on the finite memory.

■

Theorem 12.1.3. A 1-tape TM M cannot decide A by using $o(nlongn)$ steps.
Regarding little o, here are the definitions of big-O and little-o:

Definition 12.1.1 (Big-O). $f(n)$ is $O(g(n))$ if $f(n) \leq cg(n)$ for some fixed c independent of n.
$g(n)$ is an **(asymptotic) upper bound** for $f(n)$.

Definition 12.1.2 (Little-o). $f(n)$ is $o(g(n))$ if $f(n) \leq \epsilon$ for all $\epsilon > 0$ and large n.

> **Remark.** We use small-o notation to say one function is asymptotically less than another.
> The difference between big-O and little-o is analogous to the difference between $\leq$ and $<$.

Proof is not required for this course.

Theorem 12.1.4. A multi-tape TM M can decide $A = \{a^k b^k | k \geq 0\}$ using $O(n)$ steps.

Proof. We can copy a's to the second tape.
I'll omit this proof. ■

12.1.1 Model Dependence

Number of steps to decide $A = a^k b^k | k \geq 0$ depends on the model.

- **1-tape TM** : $O(nlogn)$
- **Multi-tape TM**: $O(n)$

Computability theory: model independence (Church-Truing Thesis).
Therefore the choice of model does not matter. Mathematically nice.
Complexity Theory: model dependence.
But dependence is low (polynomial) for reasonable deterministic models.
We will focus on questions that do not depend on model choice.

> **Note.** What does this "polynomial" mean?

We will continue using 1-tape TM.

12.2 TIME Complexity Class

Definition 12.2.1 (runs in time). Let $t : \mathbb{N} \to \mathbb{N}$. Say TM M runs in time $t(n)$ if M always halts within $t(n)$ steps on all inputs of length n.

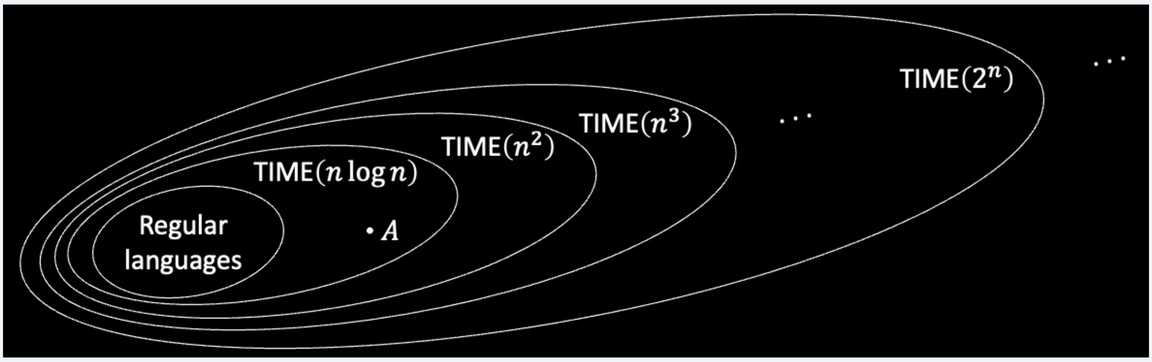
Definition 12.2.2 (TIME(t(n))). $TIME(t(n)) = \{B|$ some deterministic 1-tape TM M decides B and M runs in time $O(t(n))\}$

This is a class of languages.

Example. $A = \{a^k b^k | k \geq 0\} \in TIME(nlogn)$

> **Remark.** All the regular languages only scan, so they are under TIME(n).

Images of different class of languages regarding TIME:



Note. Is there gap between $TIME(nlogn)$ and $TIME(n)$?

This will be answered later in **Time Hierarchy Theorem**.

Problem 12.2.1. Let $B = \{ww^R | w \in \{a, b\}^*\}$, what's the smallest function t that $B \in TIME(t(n))$?

> **Remark.** ww^R means the right part is the reverse of the left part.

Correct answer is $O(n^2)$.

12.3 Multi-tape vs 1-tape

Theorem 12.3.1. Let $t(n) \geq n$.

If a multi-tape TM decides B in time $t(n)$, then $B \in TIME(t^2(n))$.

Proof. Analysis conversion of multi-tape to 1-tape TMs.

To simulate 1-step of M's computation, S uses $O(t(n))$ steps. (S stores multiple tapes in 1-tape and it has to move back and forth).

So total simulation time is $O(t(n) \times t(n)) = O(t^2(n))$ ■

Definition 12.3.1 (polynomially related). Two models of computation are polynomially related if each can simulate the other with a polynomial overhead:

So $t(n)$ times → $t^k(n)$ time on the other model, for some k.

All reasonable deterministic models are polynomially related.

- 1-tape TMs
- multi-tape TMs

- multi-dimensional TMs
- random access machine (RAM)
- cellular automata

12.4 The Class P

Definition 12.4.1 (Defnition of P). **P** is the class of languages that are decidable in polynomial time on a deterministic string-tape TM:
$P = \bigcup_k TIME(n^k)$ = Polynomial time decidable languages

Why **P** is important:

- Invariant for all reasonable deterministic models.
- Corresponds roughly to realistically **solvable** problems. (For example, not safe for cryptography)

Example. $PATH = \{\langle G, s, t\rangle | G$ is a directed graph with a path form s to t $\}$

Theorem 12.4.1. $PATH \in P$

Proof. M making all the nodes until nothing new can be marked.
Accept if t is marked, reject if not. ■

Example. $HAMPATH = \{\langle G, s, t\rangle | G$ is a directed graph with a path from s to t and the path goes through every node of G$\}$
(This path is also called **Hamiltonian Path**).

Problem 12.4.1. $HAMPATH \in P$?

Remark. This is a famous unsolved problem, equal to $N = NP$

12.5 Other examples

Theorem 12.5.1 (Thoream 7.16 in the book). Every context-free language is a member of P.

Proof. This is proved in next lecture using dynamic programming.
Previously, we have proved that every `CFL` is decidable using Chomsky Normal Form, but the algorithm required exponential time. ■

Chapter 13

P and NP, SAT, Poly-time Reducibility

13.1 The Class of NP

In a nondeterministic TM (NTM) decider, all branches halt on all inputs.

Definition 13.1.1. An NTM runs in time $t(n)$ if all branches halt within $t(n)$ steps on all inputs of length n.

> **Intuition.** This is very strong constraint, this is because it should be able to *verify* each branch in polynomial time.

Definition 13.1.2. $NTIME(t(n)) = \{B | \text{some 1-tape NTM decides } B \text{ and runs in time } O(t(n))\}$

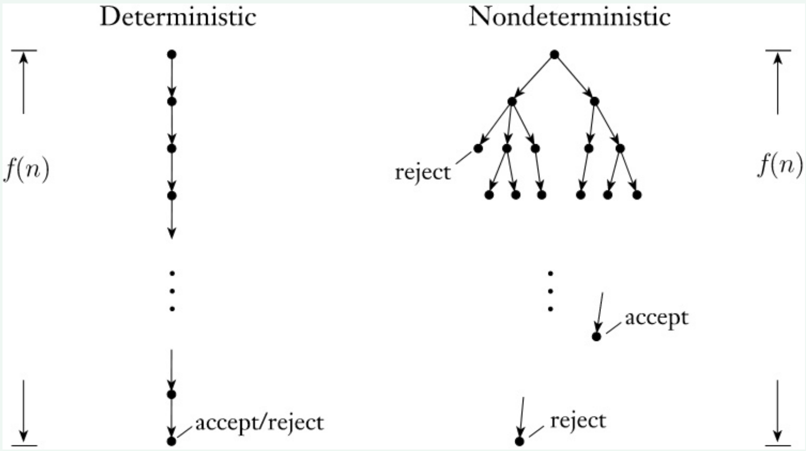


Figure 13.1: Comparing deterministic and nondeterministic time

Definition 13.1.3. $NP = \bigcup_k NTIME(n^k)$ = nondeterministic polynomial time decidable languages

- Invariant for all reasonable nondeterministic models.
- Corresponds roughly to easily verifiable problems.

13.2 Example of NP

Theorem 13.2.1. $HAMPATH \in NP$

Proof. We don't know if it's P, but we can solve it in NP.

"On input $\langle G, s, t\rangle$ (Say G has m nodes)

1. Nondeterministically write a sequence $v_1, v_2, \cdots, v_m$ of m nodes
2. Accept if $v_1 = s$

   $v_m = t$

   each $v_i, v_i + 1$ is an edge

   and no v_i repeats
3. Reject if any condition fails."

Remark. The mindset for this is that in step 1, we generate all possible sequence, this is neither P nor NP, because it is not a decision problem.

Step 2, each branch/path/permutation/sequence can be verified in P time, and this can be checked by a nondeterministic machine. That's the reason we say HAMPATH belongs to class NP languages.

■

Definition 13.2.1 (COMPOSITES). $COMPOSITES = \{x|x \text{ is not prime and } x \text{ is written in binary}\}$
$= \{x|x = yz \text{ for int } y, z > 1,\ x \text{ in binary}\}$

Theorem 13.2.2. $COMPOSITES \in NP$

Proof. "On input x

1. Nondeterministically write y where $1 < y < x$
2. Accept if y divides x with remainder 0.

   Reject if not."

Note (2002). $COMPOSITES \in P$ has been proved in 2002:

AKS primality test

■

13.3 P vs NP

Intuition. NP = All languages where you can verify membership quickly (this is called **short certificates**)

P = All languages where can test membership quickly

We know that $P \subseteq NP$, but $P = NP$ (Cook 1971)?

Problem 13.3.1. Let $\overline{HAMPATH}$ be the complement to $HAMPATH$.

$\langle G, s, t\rangle \in \overline{HAMPATH}$ if G does not have a HAMPATH path from s to t.

Is $\overline{HAMPATH} \in NP$?

The reasonable answer is "We Don't Know".

Note that we cannot invert the accept/reject output of the NTM for it, this is not how nondeterministic works!!!

13.3.1 Dynamic Programming

Theorem 13.3.1 (Recall). $A_{CFG} = \{\langle G, w\rangle | G \text{ is a CFG and } w \in L(G)\ \}$
A_{CFG} is decidable.

Proof. (Using Chomsky Normal Form(CNF)). This gave an NP type algorithm. ■

Theorem 13.3.2. $A_{CFG} \in P$

Proof. Recursive algorithm C tests if G generates w, starting at any specified variable R.
C = "On input $\langle G, w, R\rangle$

```
1. For each way to divide $w = xy$ and for each rule $R \to ST$
2. Use $C$ to test $\langle G, x, S\rangle$ and $\langle G, y, T\rangle$
3. Accept if both accept
4. Reject if none of above accepted."
```

Then decide A_{CFG} by starting from G's start variable.
C is a correct algorithm, but it takes non-polynomial time.

> **Remark** (Fix). Use recursion + memory called $DynamicProgramming(DP)$

Here's a fixed version:
D = "On input $\langle G, w, R\rangle$

```
1. If previously solved $\langle G, w, R\rangle$ then answer same, else continue.
2. For each way to divide $w = xy$ and for each rule $R \to ST$
3. Use $D$ to test $\langle G, x, S\rangle$ and $\langle G, y, T\rangle$
4. Accept if both accept
5. Reject if none of above accepted."
```

Then decide A_{CFG} by starting from G's start variable. ■

Bottom-Up DP is also briefly discussed, which is the iterative way of DP, as we often see in algorithm courses.

13.4 Satisfiability Problem

Definition 13.4.1. A **Boolean formula** ϕ has Boolean variables (TURE/FALSE values) and Boolean operations AND($\wedge$), OR($\vee$), and NOT($\neg$).

Definition 13.4.2. ϕ is **satisfiable** if ϕ evaluates to TRUE for some assignment to its variables.

Example. Let $\phi = (x \vee y) \wedge (\bar{x} \vee \bar{y})$ (Notation: $\bar{x}$ means $\neg y$).

Definition 13.4.3. $SAT = \{\ \langle\phi\rangle | \phi \text{ is satisfiable Boolean formula}\}$

Definition 13.4.4 (Cook, Levin 1971). $SAT \in P \Rightarrow P = NP$

Proof. polynomial time (mapping) reducibility (will be proved in later lecture). ■

Remark. $SAT \in NP$ (Obviously)

13.4.1 Polynomial Time Reducibility

Definition 13.4.5. A is polynomial time reducible to B ($A \leq_P B$) if $A \leq_m B$ by a reduction function $f : \Sigma^* \mapsto \Sigma^*$ that is computable in polynomial time.

For every w:

$$w \in A \Leftrightarrow f(w) \in B$$

The function f is called **polynomial time reduction** of A to B.

Theorem 13.4.1. If $A \leq_P B$ and $B \in P$ then $A \in P$.

Proof. $A \leq_P B$: $\forall w \in A$, we can find a function f making $f(w) \in B$.

$B \in P$: $\forall w \in B$, there exists a TM M which can compute w in polynomial time.

To prove $A \in P \Rightarrow$ there's a TM which can compute all strings in A at polynomial time.

N = "On input w,

1. compute $f(w)$.
2. using TM M to decide $f(w)$, return the result."

If $w \in A$, then $f(w)$ in B. The above 2 steps are both polynomial, so N can decide w in polynomial time, meaning $A \in P$. ■

Note (Analog with A_{TM}). All T-recognizable languages are polynomial reducible to A_{TM}.

Chapter 14

NP-Completeness

14.1 SAT Cont.

Definition 14.1.1. A Boolean formula ϕ is in Conjunctive Normal Form (CNF) if it has the form

$$\phi = (x \vee \bar{y} \vee z) \wedge (\bar{x} \vee \bar{s} \vee z \vee u) \wedge \cdots \wedge (\bar{z} \vee \bar{u})$$

Literal : a variable or a negated variable
Clause: an OR of literals (the thing in a brace)
CNF: an AND of clauses(all of them).
3CNF: a CNF with exactly 3 literals in each clause
$3SAT = \{ \langle\phi\rangle | \phi$ is a staisfiable 3CNF formula$\}$

Definition 14.1.2. A $k - clique$ in a graph is a subset of k nodes all directly connected by edges.

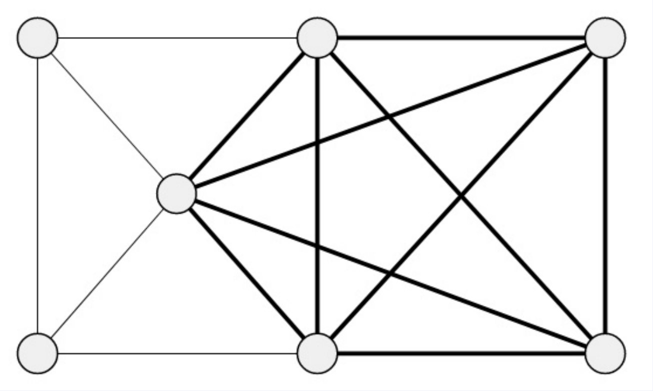


Figure 14.1: A graph with a 5-CLIQUE

Example.

$CLIQUE = \{\langle G, k\rangle \mid$ graph G contains k-clique$\}$

We will show $3SAT \leq_P CLIQUE$

Theorem 14.1.1. $3SAT \leq_P CLIQUE$

Proof. IDEA: ϕ is satisfiable $\Leftrightarrow G$ has a $k - clique$
Construction to make the transformation from $3SAT$ to $k - CLIQUE$:

Remark (Transformation). Here is the mapping from CNF to the graph:

- $3SAT\ literal \mapsto$ vertex (node) in $k - CLIQUE$
- $3SAT\ \#clause \mapsto k$ in $k - CLIQUE$
- $3SAT$ be TRUE at same time $\mapsto$ edges in the graph

Principle in making edges:

1. can't make edges within a clause
2. can't make inconsistent edges (for example we can't connect a with $\bar{a}$)

This is what the transformation looks like, forbidden edges are connected in red line:

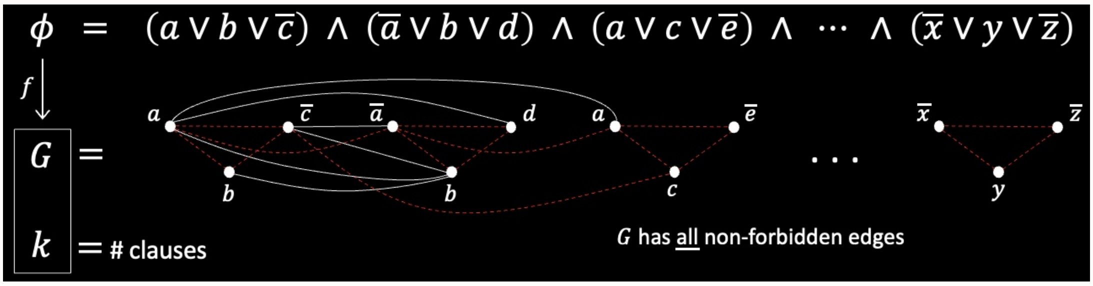

$\phi = (a \vee b \vee \bar{c}) \wedge (\bar{a} \vee b \vee d) \wedge (a \vee c \vee \bar{e}) \wedge \cdots \wedge (\bar{x} \vee y \vee \bar{z})$
f
G =
k = # clauses
a
b
$\bar{c}$
$\bar{a}$
b
d
a
c
$\bar{e}$
. . .
$\bar{x}$
y
$\bar{z}$
G has all non-forbidden edges

Figure 14.2: transformation from CNF to graph

Lemma 14.1.1 (Forward). ϕ is satisfiable $\Rightarrow G$ has a $k - clique$.

Proof. Take any satisfying assignment to ϕ. Pick 1 true literal in each clause.
The corresponding nodes in G are a $k - clique$ because they don't have forbidden edges. ■

Lemma 14.1.2 (Backward). G has a $k - clique \Rightarrow phi$ is satisfiable.

Proof. Take any $k - clique$ in G, it must have 1 node in each clause.
Set each corresponding literal TRUE. That gives a satisfying assignment to ϕ. ■

Notice that the reduction (transformation) f is computable in polynomial time. ■

Corollary 14.1.1. $CLIQUE \in P \Rightarrow 3SAT \in P$

Remark. Does the theorem require 3 literals per clause?
The answer is no, it works for any size clause.

14.2 NP-Completeness

Definition 14.2.1. Language B is NP-Complete if

1. $B \in NP$
2. For all $A \in NP$, $A \leq_P B$

Definition 14.2.2 (NP-hard)**.** A problem is **NP-hard** if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.

Notice that the 2nd constraint for defining B is NP-complete is actually saying B is NP-hard.

So we can say a language is NP-complete if it is in NP, and it is NP-hard.

If B is NP-complete and $B \in P$ then P = NP.

Theorem 14.2.1 (Cook-Levin Theorem)**.** SAT is NP-complete

Remark. Proof. Next lecture. Assume it is true for now. ■

Note. $\forall A \in NP$:

$$A \leq_P SAT \leq_P 3SAT \leq_P CLIQUE/HAMPATH$$

Note. To show some language C is NP-complete, show $3SAT \leq_P C$.

Importance of NP-complete:

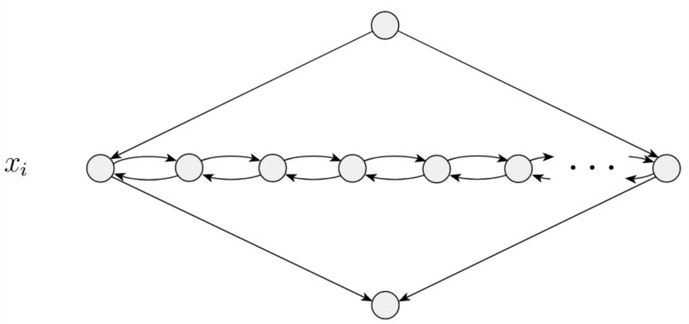
1. Showing B is NP-complete is evidence of computational intractability. (Show a problem is NP-complete is a powerful evidence that it is not P)
2. Gives a good candidate for proving $P \neq NP$.

Theorem 14.2.2. $HAMPATH$ is NP-complete.

Proof. Show $3SAT \leq_P HAMPATH$ (assume 3SAT is NP-Complete)

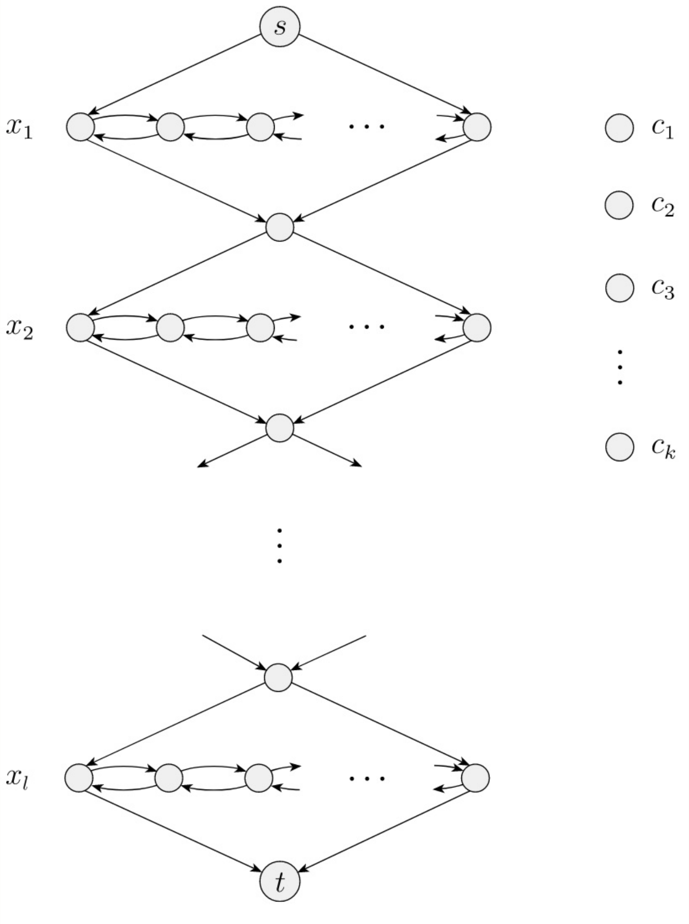
The simulation is so hard to describe, but this is a famous question, so I will just describe the setup here:

Transformation of each variable/literal x_i: it is represented with a diamond-shaped structure that contains a horizontal row of nodes:



Transformation of each clause c_j: as a node.

Global structure:



The core is the representation of a clause c_j contains an x_i or $\bar{x_i}$:

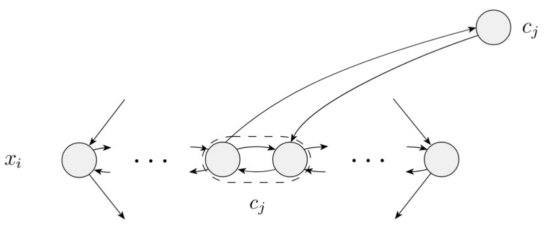


Figure 14.3: the additional edges when c_j contains x_i

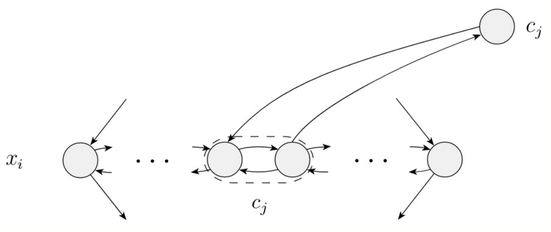


Figure 14.4: the additional edges when c_j contains $\bar{x_i}$

(The complete proof can be easily got from internet.) ■

Remark. Would this construction show the undirected Hamilton path problem is NP-complete? No, the construction depends on G being directed.

14.3 Summary

How to do polynomial reduction.

Chapter 15

Cook-Levin Theorem

15.1 Cook-Levin Theorem

Theorem 15.1.1. SAT is NP-complete

Proof. There are 2 lemma need to be proved:

Lemma 15.1.1 (Lemma1: NP)**.** $SAT \in NP$

Proof. This is already done ∎

Lemma 15.1.2 (Lemma2: NP-hard)**.** If $\forall A \in NP$, we have $A \leq_P SAT$.

Proof. $A \in NP$: $A \in NP$ be decided by NTM M in time n^k.

We want to prove there's a polynomial reduction function $f : \Sigma^* \mapsto formula$ mapping A to SAT, so:

- $f(w) = \langle \phi_{M,w} \rangle$
- $w \in A \Leftrightarrow \phi_{M,w} issatisfiable$

IDEA: the reduction for A takes a string w and produces a CNF ϕ that simulates the NP machine for A on input w.

- If the machine accepts, ϕ is satisfiable.
- If the machine doesn't accept, no assignment satisfies ϕ.

∎

Cont. What we do is to transform the problem into a history of M accepting w:

Definition 15.1.1 (tableau)**.** An (accepting) tableau for NTM M on w is an $n^k \times n^k$ table representing the computation history for M on w on an accepting branch of the nondeterministic computation.

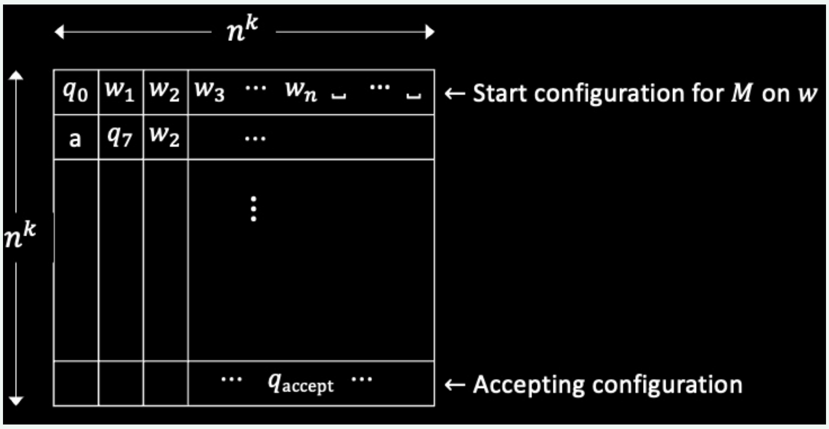

n^k
q_0 w_1 w_2 w_3 ⋯ w_n ␣ ⋯ ␣
← Start configuration for M on w
a q_7 w_2 ⋯
⋮
n^k
⋯ q_{accept} ⋯
← Accepting configuration

Figure 15.1: Example of tableau

Notice that for NTM, this is only one of the branch which ends up in an accept state.

Now we need to construct $\phi_{M,w}$ making it represent that "M accepts w", which is equal to that a tableau exists for M on w exists.

The formula $\phi_{M,w}$ can be represented as following:

$$\phi_{M,w} = \phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{accept}$$

now we will explain the 4 parts one by one.

1. ϕ_{cell}: each cell of the tableau can be a state, an alphabet or an empty space, that is to say:

In every cell at least one $Q \cup \Gamma$ and at most one.

Let $C = Q \cup \Gamma$, we have:

$$\phi_{cell} = \bigwedge_{1\leq i,j\leq n^k} \left(\bigvee_{\sigma\in C} x_{i,j,\sigma} \wedge \bigwedge_{\sigma,\tau\in C\ \sigma\neq\tau} \overline{x_{i,j,\sigma} \wedge x_{i,j,\tau}} \right)$$

2. ϕ_{start}: the start configuration should be serialized as $q_0 w_1 \cdots w_n \cdots \sqcup$:

$$\phi_{start} = x_{1,1,q_0} \wedge x_{1,2,w_1} \wedge x_{1,3,w_2} \wedge \cdots \wedge x_{1,n^k,\sqcup}$$

3. ϕ_{accept}: one of the cell is q_{accept}:

$$\phi_{accept} = \bigvee_{1\leq j\leq n^k}$$

4. ϕ_{move}: this guarantees that the computation history proceeds legally. This is achieved by recognize all the legal patterns.

To explain this, I will show some illegal examples:

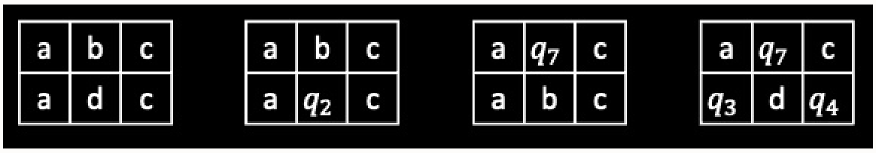
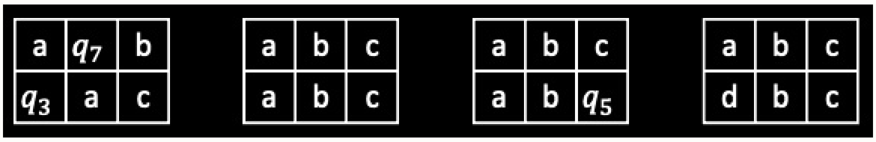


Figure 15.2: illegal patterns

■

Cont. And some legal patterns:



a	q_7	b
q_3	a	c

a	b	c
a	b	c

a	b	c
a	b	q_5

a	b	c
d	b	c

Figure 15.3: legal patterns

So we need to make the window be one of the legal patterns, and this is required by all the windows:

$$\phi_{move} = \bigwedge_{1<i,j<n^k} \Big(\bigvee_{legal_pattern} (x_{i,j-1,r} \wedge x_{i,j,r} \wedge x_{i,j+1,r} \wedge x_{i+1,j-1,r} \wedge x_{i+1,j,r} \wedge x_{i+1,j+1,r})\Big)$$

The size of $\phi_{M,w}$ is roughly the size of the tableau for M on w, so the size is $O(n^k \times n^k) = O(n^{2k})$

■

■

15.2 SAT3 is NP-complete

Theorem 15.2.1. $3SAT$ is NP-complete

Proof. Because SAT is NP-complete has already been proved, we just need to prove $SAT \leq_P 3SAT$ here.

A CNF may look like this:

$$\phi = ((a \wedge b) \vee c) \wedge (\bar{a} \vee c)$$

The key is first to use transform any conjecture into its result by list all its result:
For $a \wedge b$, we can transform it into:

$$((a \wedge b) \mapsto z_1) \wedge ((\bar{a} \wedge b) \mapsto z_1) \wedge ((a \wedge \bar{b}) \mapsto z_1) \wedge ((\bar{a} \wedge \bar{b}) \mapsto z_1) \tag{1}$$

We can transform (1) according to $(A \mapsto B) \mapsto (\bar{A} \vee B)$:

$$(\overline{(a \wedge b)} \vee z_1) \wedge (\overline{(\bar{a} \wedge b)} \vee z_1) \wedge (\overline{(a \wedge \bar{b})} \vee z_1) \wedge (\overline{(\bar{a} \wedge \bar{b})} \vee z_1) \tag{2}$$

According to de morgan's law:

$$(a \vee b \vee z_1) \wedge (\bar{a} \vee b \vee z_1) \wedge (a \vee \bar{b} \vee z_1) \wedge (\bar{a} \vee \bar{b} \vee z_1)$$

This is the same with 3SAT form, and we can do this for all conjectures. ■