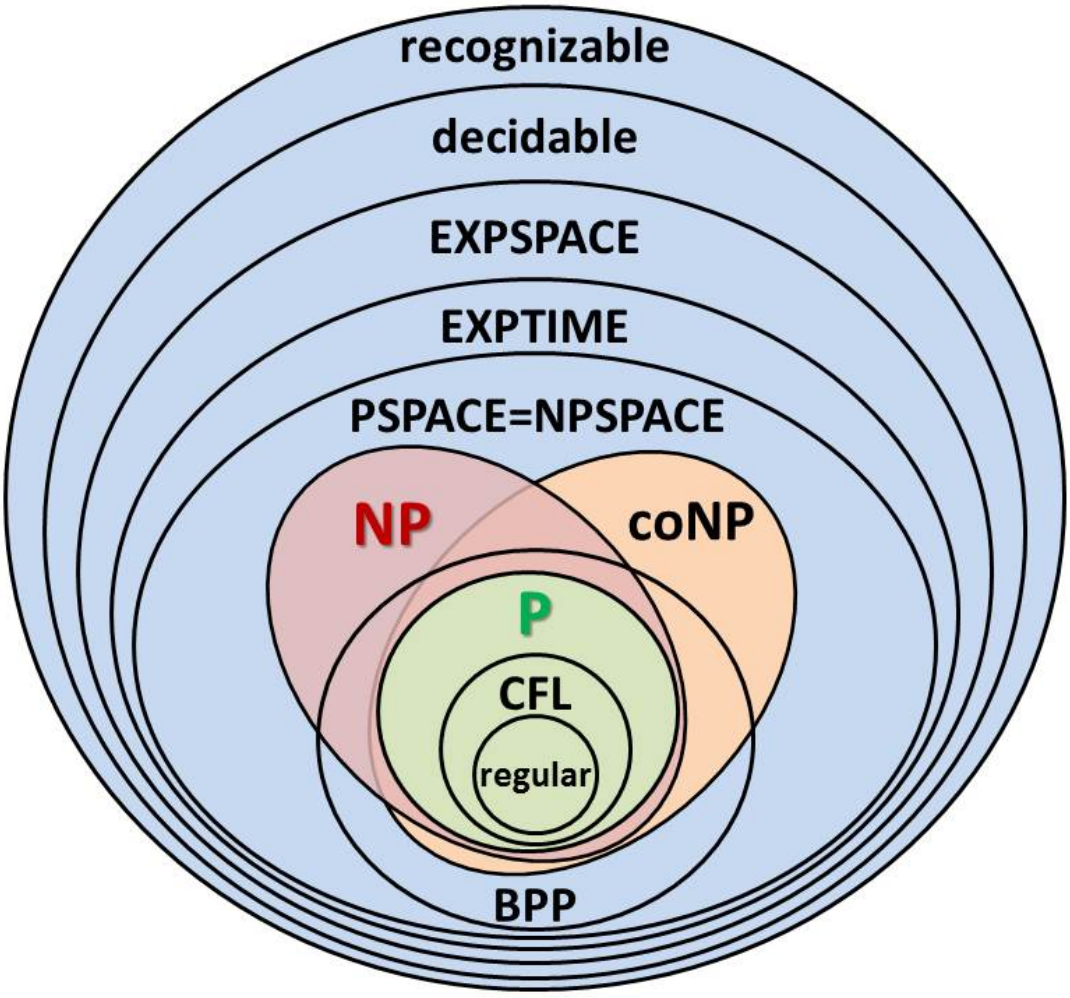
Introduction to Computation Theory

Vinsong

December 9, 2025

Abstract

The lecture note of 2025 Fall Introduction to Computation Theory by professor 林智仁.



Contents

Chapter 0

Basic Knowledge

Lecture 1

2025-09-01

0.1 Mathematical Notions

0.1.1 Set & its operation

Definition 0.1.1 (Set). Omitted

Definition (Sequence & Tuple). Here are some definitions of basic containers

Definition 0.1.2 (Sequence). Sequence is the objects in order, which have two properties:

- Order:
$$(1,2,3) \neq (2,1,3)$$
- Repetition:
$$\text{Sequence} : (1,2,3) \neq (1,1,2,3)$$
$$\text{Set} : \{1,2,3\} = \{1,1,2,3\}$$

Definition 0.1.3 (Tuple). Finite sequence, $(1,2,3)$ is a 3-tuple

Definition 0.1.4 (Cartesian Product). Here is the Cartesian Product between two sets. We define

$$A = \{1,2\},\ B = \{x,y\}$$

then,

$$A \times B = \{(1,x),(1,y),(2,x),(2,y)\}$$

0.1.2 Function & Relation

Definition 0.1.5 (Function). Function is a machine with single output.

Definition (Equivalence Relations). Here are the properties of Equivalence Relations.

Definition 0.1.6 (reflexive).

$$\forall x,\ xRx$$

Definition 0.1.7 (symmetric).

$$\forall x, y,\ xRy \iff yRx$$

Definition 0.1.8 (transitive).

$$xRy,\ yRz \implies xRz$$

Example.

$$i \equiv_7 j, \quad \text{if} \quad 0 = i - j \mod 7$$

- Reflexive

$$i - i = 0 \mod 7$$

- Symmetric

$$i - j = 7a,\ j - i = -7a$$

- Transitive

$$i - j = 7a,\ j - k = 7b \implies i - k = 7(a + b)$$

0.1.3 String & Languages

Definition (String & Languages). Here is the definition of Language.

Example (Alphabet).

$$\{0, 1\}$$

Example (String).

$$01000$$

Definition 0.1.9 (Language). Set of Strings

$$L(A)$$

is the language of A

0.2 Definitions, Theorems, and Proofs

- **Definition**: Introduce new concept.
- **Statement**: A sentence that is either true or flase.
- **Theorem**: A statement that is true.
  - **Lemma**: A "helping" theorem.
  - **Corllary**: A theorem that follows easily from another theorem.

0.2.1 Proof by Construction

Proposition 0.2.1. Sum of degrees of every graph is even

Proof. Each edge contributes 2 nodes, so

$$\sum_{v \in V} \deg(v) = 2 \times |E|$$

Hence, the sum of degrees of every graph is even. ■

Note. The implication is the definition of graphs.

0.2.2 Proof by Contradiction

Assume the statement is false, then deduce a contradiction.

0.2.3 Proof by Induction

- Basis: Prove for $n = 0$ or $n = 1$ or some trivial case.
- Inductive Step: Assume true for $n = k$ (Induction Hypothesis), prove for $n = k + 1$.

Chapter 1

Regular Languages

1.1 Deterministic Finite Automata (DFA)

- Automaton: single
- Automata: plural

Definition 1.1.1 (Deterministic Finite Automata (DFA)). We define a DFA as a 5-tuple

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

where

- Q: Set of states (Finite)
- Σ: Alphabet (i.e. set of input characters) (Finite)
- δ: $Q \times \Sigma \to Q$: Transition Function
- $q_0 \in Q$: Start state
- $F \subset Q$: Set of accept states

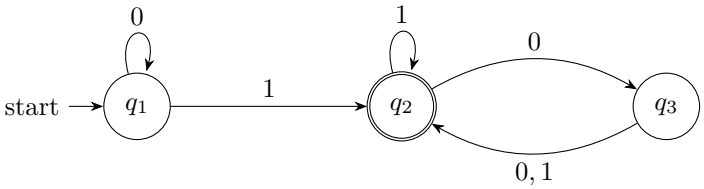


Figure 1.1: A state diagram

If we call this machine M, then we have.

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

For the example given above,

$$Q = \{q_1, q_2, q_3\}$$
$$\Sigma = \{0, 1\}$$
$$q_0 = q_1$$
$$F = \{q_2\}$$

The δ function:

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

Definition 1.1.2. The language that recognize by a Machine M is denoted as

$$L(M) = A$$

We say A is recognizeed (accepted) by M.

1.1.1 Definition of Computation

Let,

- $M = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$ be a finite automaton.
- $w = w_1, \cdots, w_n$ be a string over Σ.

Theorem 1.1.1. M accepts w if $\exists$ states $r_0 \cdots r_n$ such that

(1) $r_0 = q_0$

(2) $r_{i+1} = \delta(r_i, w_{i+1}), \quad i = [0, n-1]$

(3) $r_n \in F$

Definition 1.1.3 (Regular Language)**.** A language is regular if recognized by some automata.

1.1.2 Regular Operations

Definition. Assume A, B are given languages,

Definition 1.1.4 (Union)**.**

$$A \cup B = \{w \mid w \in A \vee w \in B\}$$

Definition 1.1.5 (Concatenation)**.**

$$A \circ B = \{w_1 w_2 \mid w_1 \in A, w_2 \in B\}$$

Definition 1.1.6 (Kleene Star).

$$A^* = \{w_1 \cdots w_k \mid k \geq 0, w_i \in A\}$$

which can also be defined as

$$\bigcup_{i=1}^{\infty} A_i = \{\varepsilon\} \cup A \cup A^2 \cup A^3 \cup \cdots, \quad A^0 = \{\varepsilon\},\ A^n = \{wv \mid w \in A^{n-1}, v \in A\}$$

Definition 1.1.7 (closed). We say an operation R is closed if the following property holds if

$$x \in A, y \in A, \text{ then } xRy \in A$$

Theorem 1.1.2. Regular languages are closed under the union, concatenation, and Kleene star.

Proof. We define two machines as follows

$$M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$$
$$M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$$

if we union them, we can define a new machine

$$M_1 \cup M_2 = \begin{cases} M = (Q, \Sigma, \delta, q_0, F) \\ Q = \{(r_1, r_2) \mid r_1 \in Q_1, r_2 \in Q_2\} \\ \delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a)) \\ q_0 = (q_1, q_2) \\ F = \{(r_1, r_2) \mid r_1 \in F_1 \text{ or } r_2 \in F_2\} \end{cases}$$

Hence, regular languages are closed under union. ■

Lecture 2

1.2 Nondeterministic Finite Automata (NFA)

2025-09-08

First, we see a NFA that accept strings with 1 in 3rd position from the end,

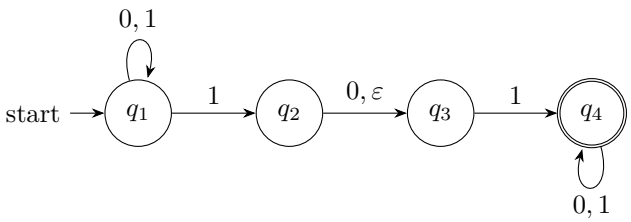


Figure 1.2: NFA machine

- δ is not a function, i.e. $\delta(q_1, 1) = q_1$ or q_2
- ε between q_2, q_3 means q_2 can move to q_3 without any input

We can transport NFA to DFA by some method, for example, for the above NFA we can have:

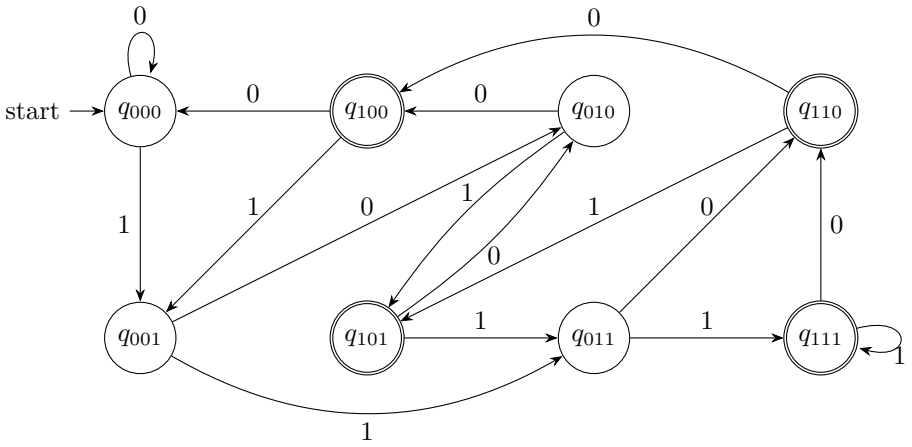


Figure 1.3: NFA machine transport to DFA

We can record it in three bits, it will be complicated.

Definition 1.2.1 (power set)**.**

$$P(Q) = \{X | X \in Q\}$$

which contain all the $2^{|Q|}$ combinations.

Definition 1.2.2 (Nondeterministic Finite Automata (NFA))**.** We define a NFA as a 5-tuple

$$M = (Q, \Sigma_\varepsilon, \delta, q_0, F)$$

where

- Q: Set of states (Finite)
- $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$
- δ: $Q \times \Sigma_\varepsilon \to P(Q)$
- $q_0 \in Q$
- $F \subset Q$

Theorem 1.2.1. We have w

$$w = y_1 \cdots y_m \quad \text{where } y_i \in \Sigma_\varepsilon$$

A sequence $r_0 \cdots r_m$ such that

(1) $r_0 = q_0$

(2) $r_{i+1} = \delta(r_i, y_{i+1}), \quad i = [0, n-1]$

(3) $r_n \in F$

Note. So m may not be the original length (as y_i may be ε)

1.2.1 Equivalence of DFA and NFA

From DFA $\Rightarrow$ NFA. Formally DFA is not an NFA due to Σ and Σ_ε. but we can easily handle this by adding

$$q_i, \varepsilon \to \emptyset$$

For NFA $\Rightarrow$ DFA, we have the example on the slides on a graph.

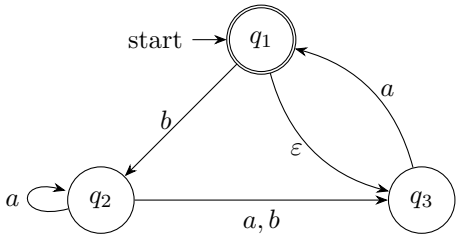


Figure 1.4: NFA example

$\Downarrow$

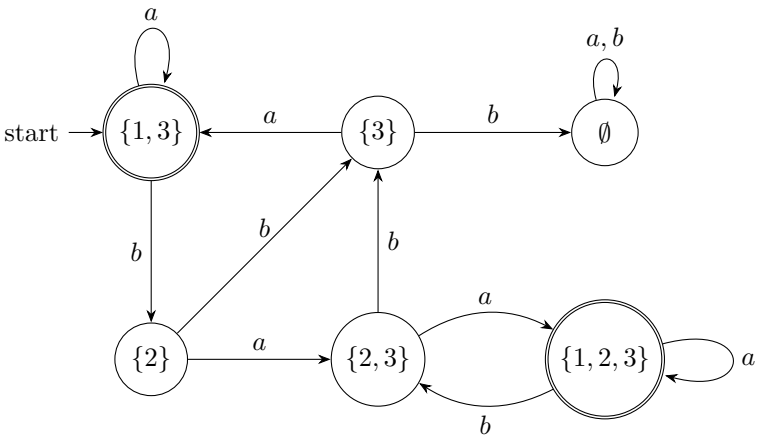


Figure 1.5: DFA convertion example

- Remove the states that are not reachable.
- Remove the states that not handle the ε transition. For example, the start state

$$\{q_1\} \text{ wrong} \quad \to \quad \{q_1, q_3\} \text{ correct}$$

Definition 1.2.3.

$$E(\{q_0\}) = \{q_0\} \cup \{\text{states reached by } \varepsilon \text{ from } q_0\}$$

Then we can redefine the procedure formally.

Theorem 1.2.2. Given a NFA

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

We can convert it to a DFA

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

where

- $Q' = P(Q)$
- $q_0' \in P(Q) = E(\{q_0\})$
- $F' = \{R \mid R \in Q', R \cap F \neq \emptyset\}$
- δ':

$$\delta'(R, a) = \bigcup_{r \in R} E(\delta(r, a))$$

1.2.2 Closure under requlars operations

We give two NFAs N_1, N_2,

$$N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$$
$$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$$

note that $\varepsilon \notin \Sigma$, and the graph of them are:

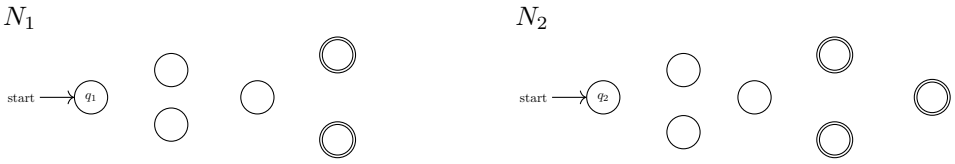


Figure 1.6: N_1, N_2

- **Union**: We can contrruct the $N_1 \cup N_2$ in

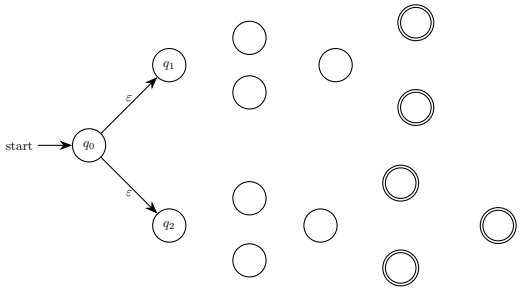


Figure 1.7: $N_1 \cup N_2$

Proposition 1.2.1 (Construction of Union). New NFA is

$$N_1 \cup N_2 = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$$

where

- $Q = Q_1 \cup Q_2 \cup \{q_0\}$
- δ :

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \\ \delta_2(q,a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0, a = \varepsilon \\ \emptyset & q = q_0, a \neq \varepsilon \end{cases}$$

- $F = F_1 \cup F_2$

- **Concatenation**: We can construct the $N_1 \circ N_2$ in

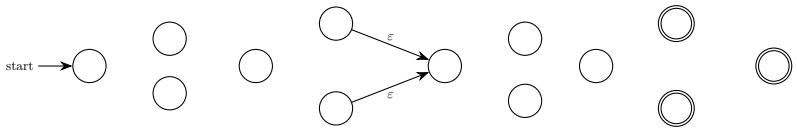


Figure 1.8: $N_1 \circ N_2$

Proposition 1.2.2 (Construction of Concatenation)**.** New NFA is

$$N_1 \circ N_2 = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$$

where

- $Q = Q_1 \cup Q_2$
- δ :

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1\ F_1 \\ \delta_2(q,a) & q \in Q_2 \\ \delta_1(q,\varepsilon) \cup \{q_2\} & q \in F_1, a = \varepsilon \\ \delta_1(q,\varepsilon) & q \in F_1, a \neq \varepsilon \end{cases}$$

- $q_0 = q_1$
- $F = F_2$

- **Kleene star**: N_1^* can also accept $\{\emptyset\}$, then we can construct the N_1^* in

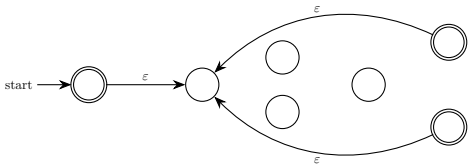


Figure 1.9: N_1^*

Proposition 1.2.3 (Construction of Kleene Star)**.** New NFA is

$$N_1^* = (Q_1,\ \Sigma,\ \delta_1,\ q_0,\ F_1)$$

where

- $Q = Q_1 \cup \{q_0\}$

- δ :

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1\ F_1 \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1, a = \varepsilon \\ \delta_1(q,\varepsilon) & q \in F_1, a \neq \varepsilon \\ \{q_1\} & q = q_0, a = \varepsilon \\ \emptyset & q = q_0, a \neq \varepsilon \end{cases}$$

- $F = F_1 \cup \{q_0\}$

Note. Some operations are also closed under regular languages,

- **Intersection:**

$$A_1 \cap A_2$$

Use the product automaton (the same construction as for Union). A string is accepted if and only if the state is in the accept states of both N_1 and N_2 at the same time.

- **Set Difference:**

$$A_1 - A_2$$

Use the product automaton as well. A string is accepted if the state is in the accept states of N_1 but *not* in the accept states of N_2.

- **Complement:**

$$A_1^c = \Sigma^* - A_1$$

Since Σ^* is regular and the class of regular languages is closed under set difference, A_1^c is also regular.

Lecture 3

1.3 Regular expressions

2025-09-15

A regualar expression is a tool to describe a language.

Definition 1.3.1 (Regular expressions). R is a regular expressions if it is one of the following expressions:

(1) a, where $a \in \Sigma$

(2) ε ($\varepsilon \notin \Sigma$)

(3) $\emptyset$

(4) $R_1 \cup R_2$, where R_1, R_2 are regular expressions

(5) $R_1 \circ R_2$, where R_1, R_2 are regular expressions

(6) R_1^*, where R_1 is a regular expression

If their is no parentheses, we follow the order of:

`Kleene star` → `Concatenation` → `Union`

Remark.

$$R^+ = RR^*, \quad R^+ \cup \{\varepsilon\} = R^*$$

For $\emptyset$ and ε, we have

- ε: empty string
- $\emptyset$: empty language (language without any string)

$$(0 \cup \varepsilon)1^* = 01^* \cup 1^*$$
$$(0 \cup \emptyset)1^* = 01^*$$
$$\emptyset 1^* = 1^*\emptyset = \emptyset$$

Example. Here are some examples,

- Strings that start and end with the same symbol:

$$0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1$$

- $(\Sigma\Sigma)^*$: strings with even length
- $R \cup \emptyset = R$
- $R \circ \varepsilon = R$
- $\emptyset^* = \{\varepsilon\}$

Floating point numbers can also be represented by regular expressions. For example,

$$(+ \cup - \cup \varepsilon)(DD^* \cup DD^*.D^* \cup D^*.DD^*), \text{ where } D = \{0, \ldots, 9\}$$

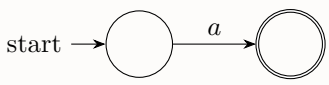
Example.

$$72 \in DD^*$$
$$2.1 \in DD^*.D^*$$
$$7. \in DD^*.D^*$$
$$.01 \in D^*.DD^*$$

Lemma 1.3.1. Language by a regular expression $\Longrightarrow$ Regular (described by an automaton)

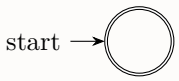
Proof. The proof is by induction,

- $R = a \in \Sigma$ can be recognize by



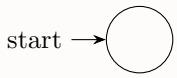
$$N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$$
$$\delta(q_1, a) = \{q_2\}$$
$$\delta(r, b) = \emptyset, r \neq q_1 \text{ or } b \neq a$$

- $R = \varepsilon$



$$N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$$
$$\delta(q_1, a) = \emptyset, \forall a$$

- $R = \emptyset$



$$N = (\{q\}, \Sigma, \delta, q, \emptyset)$$
$$\delta(r, a) = \emptyset, \forall r, a$$

- $R = R_1 \cup R_2,\ R = R_1 \circ R_2,\ R = R_1^*$ have proof by NFA.

■

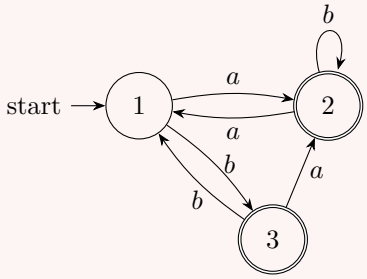
1.3.1 Convert a DFA to a regular expression

The idea is:

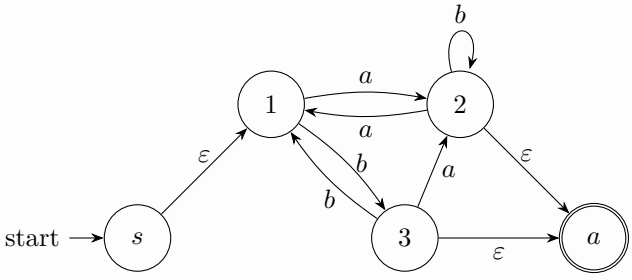
1° DFA $\longrightarrow$ GNFA

2° Remove states from GNFA until only the start and accept states.

Question. Convert the following DFA into regular expression.



Answer. First, convert to GNFA:



Next, is to remove the states one by one. We skip, so we can get the answer:

$$(a(aa \cup b)^* ab \cup b)((ba \cup a)(aa \cup b)^* ab \cup bb)^*((ba \cup a)(aa \cup b)^* \cup \varepsilon) \cup a(aa \cup b)^*$$

which is very complicated. ⊛

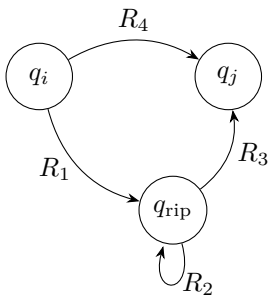
Definition 1.3.2 (Generalized NFA(GNFA)). We define a GNFA as a 5-tuple

$$G = (Q, \Sigma, \delta, q_{start}, q_{accept})$$

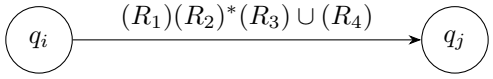
where

- F is not a se, but a single accept state q_{accept}
- δ function is:
$$(Q - \{q_{accept}\}) \times (Q - \{q_{start}\}) \to R$$
where R is all regular expressions over Σ.
- Two new states:
$$q_{start} \to q_0 \text{ with } \varepsilon$$
$$\text{any } q \in F \to q_{accept} \text{ with } \varepsilon$$

Consider q_{rip} is the state being removed



The new regular expression between q_i and q_j is



We can wrote the whole process into a algorithm.

Algorithm 1.1: CONVERT(G) —State-Elimination from GNFA to RE

Input: $G = (Q, \Sigma, \delta, q_s, q_a)$ a GNFA

Output: A regular expression R for the language of G

$k \leftarrow |Q|$;
; `// number of states`
if $k = 2$ **then**
  return $\delta(q_s, q_a)$; `// the (single) edge label from` q_s `to` q_a
Choose any $q_{\text{rip}} \in Q \setminus \{q_s, q_a\}$;
$Q' \leftarrow Q \setminus \{q_{\text{rip}}\}$;
Initialize δ' as the restriction of δ to $Q' \times Q'$;
foreach $q_i \in Q' \setminus \{q_a\}$ **do**
  foreach $q_j \in Q' \setminus \{q_s\}$ **do**
    $R_1 \leftarrow \delta(q_i, q_{\text{rip}})$;
    $R_2 \leftarrow \delta(q_{\text{rip}}, q_{\text{rip}})$;
    $R_3 \leftarrow \delta(q_{\text{rip}}, q_j)$;
    $R_4 \leftarrow \delta(q_i, q_j)$;
    $\delta'(q_i, q_j) \leftarrow R_4 \cup \left(R_1 R_2^* R_3\right)$;
$G' \leftarrow (Q', \Sigma, \delta', q_s, q_a)$;
return CONVERT(G');

Lecture 4

1.4 Pumping lemma

2025-09-22

1.4.1 Non regular language

Some languages cannot be recognized by DFA such as,

$$\{0^n 1^n \mid n \geq 0\}$$

We might remember #0 first, but # of possible n's is ∞, so we have some method to prove that the language is non-regular.

Theorem 1.4.1 (pumping lemma). If A is regular, $\exists p$ such that $\forall s \in A, |s| \geq p$,

$$\exists x, y, z, \text{ such that } s = xyz \text{ and}$$

1° $\forall i \geq 0, xy^i z \in A$

2° $|y| > 0$

3° $|xy| \leq p$

Proof. Skip, which is on the slides. ■

1.4.2 Example for Pumping Lemma

Question. Show that the language $L = \{0^n 1^n \mid n \geq 0\}$ is not regular using the pumping lemma.

Answer. Now consider the string

$$s = 0^p 1^p$$

We know that $|s| \geq p$. By the lemma, s can be split into xyz such that

$$xy^i z \in B, \forall i \geq 0, \quad |y| > 0, \quad \text{and } |xy| \leq p$$

1° If $y = 0 \cdots 0$, then

$$xy = 0 \cdots 0 \quad \text{and} \quad z = 0 \cdots 0\, 1 \cdots 1.$$

Thus,

$$xy^2 z : \#0 > \#1.$$

Hence $xy^2 z \notin B$, a contradiction.

2° If $y = 1 \cdots 1$, then similarly

$$xy^2 z \notin B \quad \text{as} \quad \#0 < \#1.$$

3° If $y = 0 \cdots 0\, 1 \cdots 1$, then

$$xy^2 z \notin B \quad \text{since it is not of the form } 0^* 1^*.$$

Note. Just pick one is sufficient to show the answer.

⊛

Question. Show that the language $C = \{w \mid \#0 = \#1\}$ is not regular using the pumping lemma.

Answer. We can use the situation in the pevious example, consider

$$s = 0^p 1^p$$

We can't proof the third condition due to $C = \{w \mid \#0 = \#1\}$ which just require the $\#0 = \#1$. Then we can use the third condition

$$|xy| \leq p$$

which means y are strict into the first 0^p we can only consider the first case.

$$|xy| \leq p \Rightarrow y = 0 \cdots 0 \text{ in } s = 0^p 1^p$$

Then,

$$xy^2 z \notin C$$

⊛

Lemma 1.4.1. When using pumping lemma, we usually use contradiction, so we use

$$\forall p\ \exists s \in A,\ |s| \geq p,\ \Big[\forall x, y, z\ \Big((s = xyz \wedge |y| > 0 \wedge |xy| \leq p)\ \rightarrow\ \exists i \geq 0,\ xy^i z \notin A\Big)\Big].$$

Use the claim and the first, second condition to get the negation of the third condition.

Question. $D = \{1^{n^2} \mid n \geq 0\}$ is not regular

Answer. We pick

$$s = 1^{p^2} \in D$$

Then, if $s = xyz, |xy| \leq p, |y| > 0$, we can get

$$p^2 < |xy^2z| \leq p^2 + p \leq (p+1)^2$$

hence, $xy^2z \notin D$. ⊛

Chapter 2

Context-Free Languages

Lecture 5

2.1 Context-Free Grammars (CFG)

2025-10-20

Which is more powerful, and can be used in compilers. A **Grammar** is a collection of substitution rules that describe the structure of a language.

> **Example.** Consider a grammar G_1:
>
> $$A \to 0A1$$
> $$A \to B$$
> $$B \to \#$$

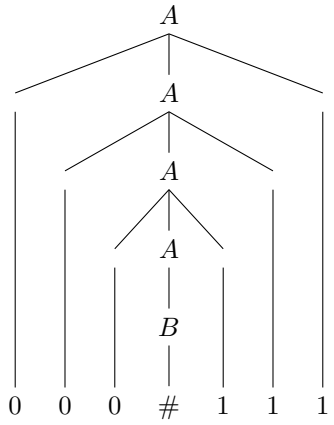
Here are the jargon terms:

- Each of one is called a **substitution rule**.
- **Variables** (non-terminals): A, B (Capital letters)
- **Terminals**: $0, 1, \#$ (Lowercase letters, numbers, symbols)
- **Start variable**: A (the variable we start with)

The process of generating strings is called **derivation**. G_1 generates 000#111 by

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

We can show the derivation using a **parse tree**:



2.1.1 Definition of CFG

The language of grammar G is denoted by $L(G)$, for the language we discuss here,

$$L(G_1) = \{0^n \# 1^n \mid n \geq 0\}$$

Now we give the formal definition of CFG.

Definition 2.1.1 (Context-Free Grammar)**.** We defined a CFG as a 4-tuple

$$G = (V, \Sigma, R, S)$$

where

- V: Variables (Finite)
- Σ: Terminals (Finite)
- R: Rules:
$$\text{Variables} \to \text{Strings of Variables and Terminals (including } \varepsilon)$$
- $S \in V$: Start variable

For instance, for G_1,

$$G_1 = (\{A, B\}, \{0, 1, \#\}, R, A)$$

where R is:

$$A \to 0A1 \mid B, \quad B \to \#$$

Notation. If u, v, w are strings and rule $A \to w$ is applied, then we say

$$uAv \text{ yields } uwv$$

denoted as

$$uAv \Rightarrow uwv$$

Notation. If

$$u = v \text{ or } u \Rightarrow u_1 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$$

then we write

$$v \overset{*}{\Rightarrow} u$$

Definition 2.1.2 (Language of a CFG)**.** The language generated by a CFG G with start variable S is

$$L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$$

2.1.2 Examples of CFGs

Question. Consider the grammar $G_2 = (\{S\}, \{a, b\}, R, S)$:

$$S \to aSb \mid SS \mid \varepsilon$$

What is $L(G_2)$?

Answer. If we let a, b be the left and right parentheses respectively, then $L(G_2)$ is the set of all balanced parentheses. ⊛

Example. Consider the grammar $G_3 = (V, \Sigma, R, S)$ where

- $V = \{\langle \text{expr} \rangle, \langle \text{term} \rangle, \langle \text{factor} \rangle\}$
- $\Sigma = \{+, \times, (,), a\}$
- R:

$$\begin{aligned} \langle \text{expr} \rangle &\to \langle \text{term} \rangle + \langle \text{expr} \rangle \mid \langle \text{term} \rangle \\ \langle \text{term} \rangle &\to \langle \text{factor} \rangle \times \langle \text{term} \rangle \mid \langle \text{factor} \rangle \\ \langle \text{factor} \rangle &\to (\langle \text{expr} \rangle) \mid a \end{aligned}$$

Consider the string $a + a \times a$:

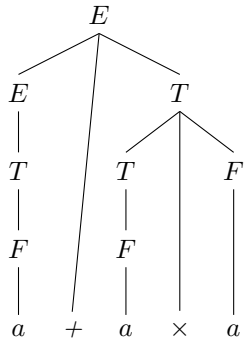


Figure 2.1: Parse tree of $a + a \times a$

Consider the string $(a + a) \times a$:

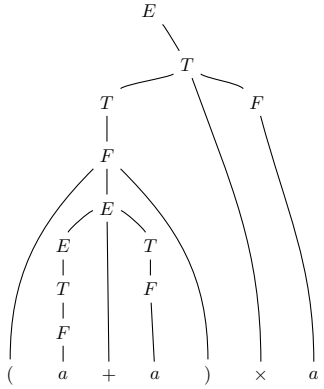


Figure 2.2: Parse tree of $(a + a) \times a$

Note. The example above shows that CFGs can express operator precedence and associativity.

2.1.3 Design of CFGs

We can design CFGs in many methods. Here are some common patterns:

- Combining smaller parts:

  Example. $L(G) = \{a^n b^n \mid n \geq 0\} \cup \{b^n a^n \mid n \geq 0\}$

  We can let the rule R be:

$$S_1 \to aS_1b \mid \varepsilon$$
$$S_2 \to bS_2a \mid \varepsilon$$
$$S \to S_1 \mid S_2$$

- From DFA:

  Lemma 2.1.1. For any regular language A, there exists a CFG G such that $L(G) = A$. The rules of CFG can be

$$R_i \to aR_j \quad \text{for each transition } \delta(q_i, a) = q_j$$
$$R_i \to \varepsilon \quad \text{if } q_i \in F$$

  The difference is that CFG allows the format

$$R_i \to aR_jb$$

  But DFA only allows

$$R_i \to aR_j$$

  where we treat R_i as the state and let $\delta(R_i, a) = R_j$.

2.1.4 Parse Trees and Ambiguity

If we let the rules of G_3 be

$$\langle \text{expr} \rangle \to \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid \langle \text{expr} \rangle \times \langle \text{expr} \rangle \mid (\langle \text{expr} \rangle) \mid a$$

We can see the following two parse trees for $a + a \times a$:

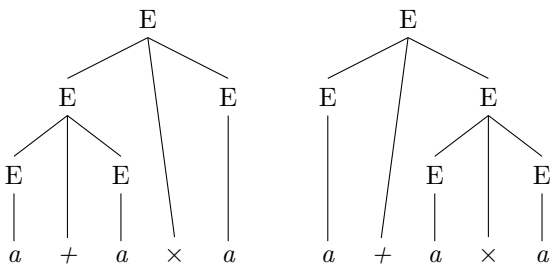


Figure 2.3: Two different parse trees for $a + a \times a$ under ambiguous grammar

This is called **ambiguity**. A CFG is **ambiguous** if there exists some string with two or more different parse trees. The above G_3 is **unambiguous**, G_3' with new rules is **ambiguous**.

However, an unambiguous grammar may also generate same parse tree but different derivations. Consider G_3:

- We can do derivation

$$\begin{aligned}\langle \text{expr} \rangle &\Rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \\ &\Rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \times \langle \text{factor} \rangle\end{aligned}$$

- We can also do derivation

$$\begin{aligned}\langle \text{expr} \rangle &\Rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \\ &\Rightarrow \langle \text{term} \rangle + \langle \text{term} \rangle\end{aligned}$$

which is not considered ambiguous. So we have the following definition:

Definition 2.1.3 (leftmost derivation)**.** A **leftmost derivation** is a derivation where at each step, the leftmost variable is replaced.

Then we can have the formal definition of ambiguity:

Definition 2.1.4 (Ambiguous)**.** A is **ambiguous** if $w \in A$ and there exists two or more different leftmost derivations for w.

Definition 2.1.5 (Inherent Ambiguity)**.** A language is **inherently ambiguous** if it only has ambiguous grammars.

Example. Consider the language

$$L = \{a^i b^j c^k \mid i = j \text{ or } j = k\}$$

We can consider the string $a^2b^2c^2$. It can be generated by two different leftmost derivations. First we consider

$$S \Rightarrow S_1 \mid S_2$$

- Using $i = j$:

$$\begin{aligned}S_1 &\to AC \\ A &\to aAb \mid \varepsilon \\ C &\to c\,C \mid \varepsilon\end{aligned}$$

  the derivation is

$$S_1 \Rightarrow AC \Rightarrow aAbC \Rightarrow aaAbbC \Rightarrow aabbC \Rightarrow aabbcC \Rightarrow aabbcc$$

- Using $j = k$:

$$\begin{aligned}S_2 &\to A'C' \\ A' &\to a\,A' \mid \varepsilon \\ C' &\to b\,C'\,c \mid \varepsilon\end{aligned}$$

  the derivation is

$$S_2 \Rightarrow A'C' \Rightarrow aA'C' \Rightarrow aaA'bC'c \Rightarrow aabbC'cc \Rightarrow aabbcc$$

Lecture 6

2.2 Chomsky Normal Form

2025-10-27

We want to simplify the structure of context-free grammars. One useful normal form is the Chomsky Normal Form (CNF).

Definition 2.2.1 (Chomsky Normal Form). A context-free grammar is in **Chomsky Normal Form** if all its production rules are of the form:

- $A \to BC$, where A, B, C are non-terminal symbols and B, C are not the start symbol.
- $A \to a$, where $a \in \Sigma$ ($\varepsilon \notin \Sigma$)
- $S \to \varepsilon$ is allowed, where S is the start symbol.

Example. Convert the following CFG to CNF:

$$S \to ASA \mid aB$$
$$A \to B \mid S$$
$$B \to b \mid \varepsilon$$

First, we add S_0 as the new start symbol:

$$S_0 \to S \quad S \to ASA \mid aB \quad A \to B \mid S \quad B \to b \mid \varepsilon$$

Next, we remove the ε-productions $B \to \varepsilon$:

$$S_0 \to S \quad S \to ASA \mid aB \mid a \quad A \to B \mid \varepsilon \mid S \quad B \to b$$

Next, we remove the ε-productions $A \to \varepsilon$:

$$S_0 \to S \quad S \to ASA \mid aB \mid a \mid AS \mid SA \mid S \quad A \to B \mid S \quad B \to b$$

Next, we remove single production $S \to S$:

$$S_0 \to S \quad S \to ASA \mid aB \mid a \mid AS \mid SA \quad A \to B \mid S \quad B \to b$$

Next, we remove single production $S_0 \to S$:

$$S_0 \to ASA \mid aB \mid a \mid AS \mid SA \quad S \to ASA \mid aB \mid a \mid AS \mid SA \quad A \to B \mid S \quad B \to b$$

Next, we remove single production $A \to B$, $A \to S$:

$$S_0 \to ASA \mid aB \mid a \mid AS \mid SA \quad S \to ASA \mid aB \mid a \mid AS \mid SA \quad A \to b \mid ASA \mid aB \mid a \mid AS \mid SA \quad B \to b$$

Finally, we convert to CNF by introducing new variables for terminals and breaking down long productions:

$$S_0 \to AA_1 \mid UB \mid a \mid AS \mid SA$$
$$S \to AA_1 \mid UB \mid a \mid AS \mid SA$$
$$A \to b \mid AA_1 \mid UB \mid a \mid AS \mid SA$$
$$A_1 \to SA$$
$$B \to b$$
$$U \to a$$

2.2.1 Procedure of Converting CFG to CNF

To convert any CFG to CNF, we can follow these steps:

1° **Add** a new start symbol S_0 with the production

$$S_0 \to S$$

2° **Remove** all ε-productions, except for the start symbol, i.e. $A \to \varepsilon$ $(A \neq S_0)$, for any

$$\cdots \to uAv$$

add the production

$$\cdots \to uv$$

3° **Remove** single productions of $A \to B$ where $A, B \in V/\{S\}$.

$$A \to B,\ B \to \gamma \quad \Rightarrow \quad A \to \gamma$$

> **Remark.** $A \to \gamma$ can't be a unit rule previously removed.

4° **Convert** remaining productions to CNF:

$$A \to u_1 u_2 \cdots u_k \quad u_i \in V \cup \Sigma$$

and

$$\text{if } k = 1, \text{ then } u_i \in \Sigma$$

Convert as follows:

$$A \to u_1 A_1$$
$$A_1 \to u_2 A_2$$
$$\vdots$$

Replaced every terminal $u_i \in \Sigma$ with a new variable U_i:

$$U_i \to u_i \quad u_i \in \Sigma$$

2.2.2 Infinite Loop in Converting

> **Example.** Consider the grammar:
>
> $$S \to B \mid \varepsilon$$
> $$B \to S \mid \varepsilon$$

We first add a new start symbol:

$$S_0 \to S \quad S \to B \mid \varepsilon \quad B \to S \mid \varepsilon$$

Next, we remove the ε-productions:

$$S_0 \to S \mid \varepsilon \quad S \to B \quad B \to S \mid \varepsilon$$

Next, we remove the ε-productions again:

$$S_0 \to S \mid \varepsilon \quad S \to B \mid \varepsilon \quad B \to S$$

This process will continue indefinitely. The reason is $S \to \varepsilon$ has been handled. So there is no need to add $S \to \varepsilon$.

2.3 Pushdown Automata

We now introduce the machine that recognizes context-free languages (CFL), called Pushdown Automata (PDA). PDA is a machine with a stack, which is a way to store previous states.

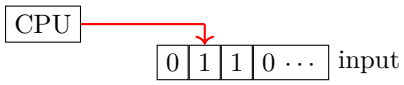


Figure 2.4: DFA or NFA

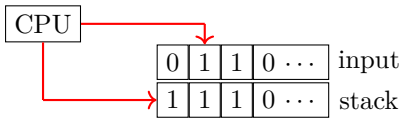


Figure 2.5: Pushdown Automata (PDA)

Example. Consider the language $A = \{0^n 1^n \mid n \geq 0\}$. We can design a PDA to recognize A:

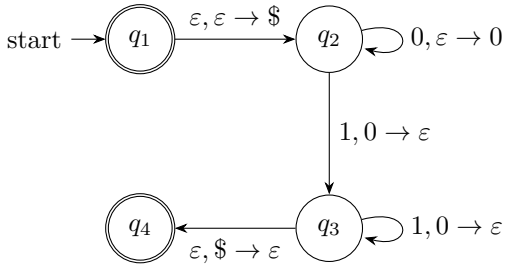


Figure 2.6: PDA for $A = \{0^n 1^n \mid n \geq 0\}$

$ is a special bottom stack symbol to indicate the initial state of the stack. The PDA works as follows:

- $q_2 \to q_2$, put 0 into stack
- $q_2 \to q_3$ and $q_3 \to q_3$, read 1 and `pop` 0 up

If the input is 0011 which is same as $\varepsilon 0011 \varepsilon$, the process is as follows:

$$
\begin{aligned}
&q_1, \emptyset, \varepsilon \\
&q_2, \{\$\}, 0 \\
&q_2, \{0, \$\}, 0 \\
&q_2, \{0, 0, \$\}, 1 \\
&q_3, \{0, \$\}, 1 \\
&q_3, \{\$\}, \varepsilon \\
&q_4, \{\}
\end{aligned}
$$

Notation. $\{\}$: contents of the stack before processing the input character.

2.3.1 Formal definition of PDA

Definition 2.3.1 (Pushdown Automata). A **pushdown automaton** (PDA) is a 6-tuple

$$(Q, \Sigma, \Gamma, \delta, q_0, F)$$

, where

- Q: States
- Σ: Input alphabet
- Γ: Stack alphabet
- δ: Transition function
$$Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$$
- $q_0 \in Q$: Start state
- $F \subset Q$: Set of accepting states

The definition of the above PDA for $A = \{0^n 1^n \mid n \geq 0\}$ is as follows:

- $Q = \{q_1, q_2, q_3, q_4\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{0, \$\}$
- $q_0 = q_1$
- $F = \{q_1, q_4\}$

For the the transition function, we care about three things:

- Current state
- Current input
- Top of the stack

The transition function δ works as follows:

	0			1			ε		
	0	\$	ε	0	\$	ε	0	\$	ε
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$	$\{(q_3, \varepsilon)\}$					
q_3				$\{(q_3, \varepsilon)\}$			$\{(q_4, \varepsilon)\}$		
q_4									

For example, we say the transition of $q_2 \to q_3$ to be

$$\delta(q_2, 1, 0) = \{(q_3, \varepsilon)\}$$

2.3.2 Nondeterministic situation

Example. Design a PDA for the language $B = \{a^i b^j c^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } j = k\}$.

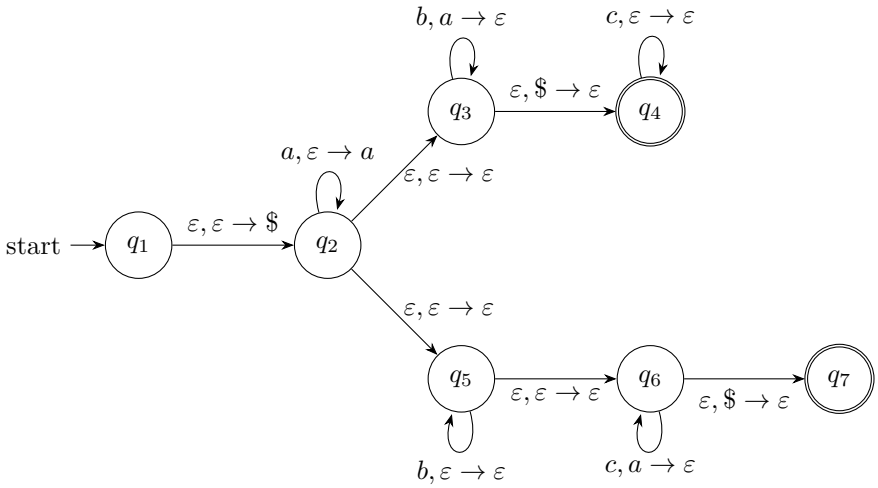
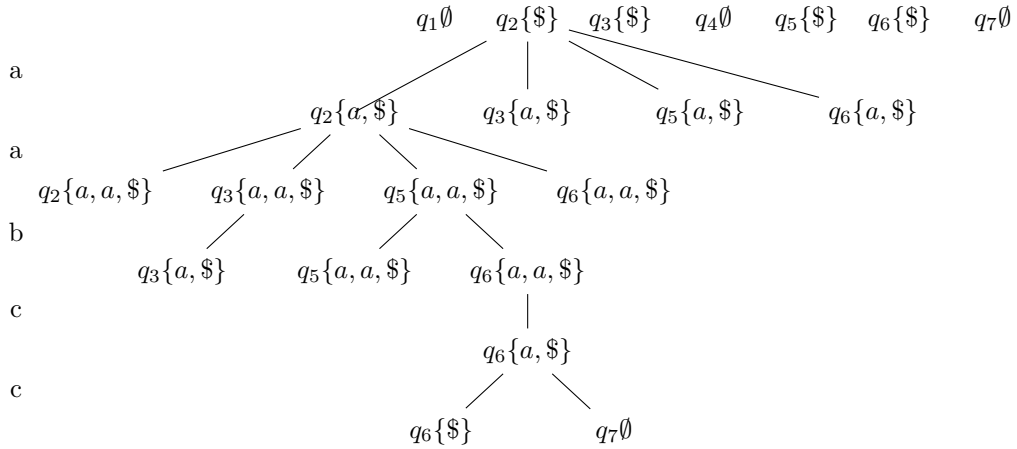


Figure 2.7: Nondeterministic PDA

We input a^2bc^2, to illustrate the process, we can build the following computation tree:



Example. Design a PDA for the language $C = \{ww^R \mid w \in \{0, 1\}^*\}$.

Idea. Symbols pushed to stack, nondeterministically guess middle is reached

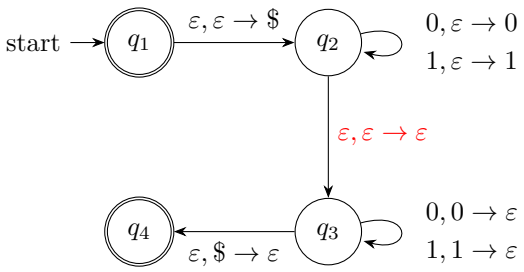


Figure 2.8: PDA for $C = \{ww^R \mid w \in \{0, 1\}^*\}$

2.3.3 Converting CFL to PDA

Example. Convert the CFG G to PDA that recognizes $L(G)$:

$$S \to aTb \mid b$$
$$T \to Ta \mid \varepsilon$$

Idea. For rule substitution, we replace the left-hand side variable with the right-hand side string i.e.

$$A \to \gamma \quad \Rightarrow \quad \texttt{pop}\ A \text{ from stack, } \texttt{push}\ \gamma \text{ to stack}$$

if there are multiple productions for A, we `push` them in a reversed way.

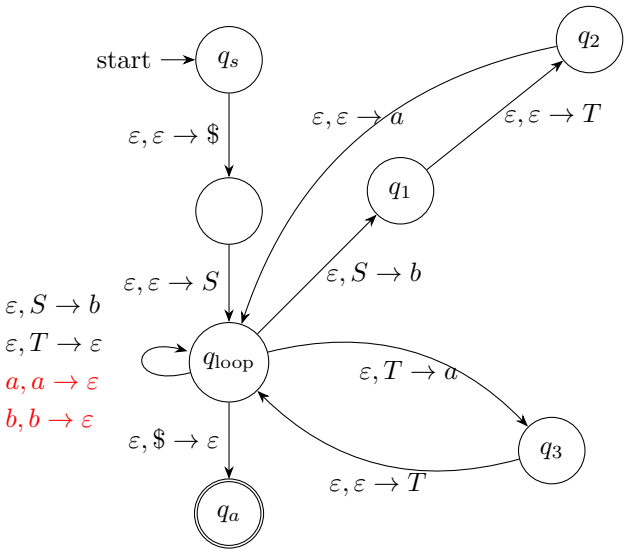


Figure 2.9: PDA for CFG G

Remark. There are two transitions we must add to process the "input":

$$a, a \to \varepsilon$$
$$b, b \to \varepsilon$$

The procedure of converting CFG to PDA is as follows:

$$\begin{aligned}
&q_{\text{start}} \xrightarrow{\varepsilon} q_{\text{loop}}, \{S, \$\} \xrightarrow{\varepsilon} q_1, \{b, \$\} \xrightarrow{\varepsilon} q_2, \{T, b, \$\} \\
&\xrightarrow{\varepsilon} q_{\text{loop}}, \{a, T, b, \$\} \xrightarrow{a} q_{\text{loop}}, \{T, b, \$\} \\
&\xrightarrow{\varepsilon} q_3, \{a, b, \$\} \xrightarrow{\varepsilon} q_{\text{loop}}, \{T, a, b, \$\} \\
&\xrightarrow{\varepsilon} q_3, \{a, a, b, \$\} \xrightarrow{\varepsilon} q_{\text{loop}}, \{T, a, a, b, \$\} \\
&\xrightarrow{\varepsilon} q_3, \{a, a, a, b, \$\} \xrightarrow{\varepsilon} q_{\text{loop}}, \{T, a, a, a, b, \$\} \\
&\xrightarrow{\varepsilon} q_{\text{loop}}, \{a, a, a, b, \$\} \xrightarrow{a} q_{\text{loop}}, \{a, a, b, \$\} \\
&\xrightarrow{a} q_{\text{loop}}, \{a, b, \$\} \xrightarrow{a} q_{\text{loop}}, \{b, \$\} \\
&\xrightarrow{b} q_{\text{loop}}, \{\$\} \xrightarrow{\varepsilon} q_{accept}
\end{aligned}$$

Proposition 2.3.1. Even with a non-deterministic setting, we ensure that only strings generated by this CFG can be accepted by the PDA

- A string is accepted only if all characters are processed (this is part of the PDA definition!)
- We have \$ to ensure that the stack is empty in the end

2.3.4 Converting PDA to CFL

Lemma 2.3.1. Language recognized by PDA $\implies$ context free

Note. We need PDA to satisfy

1° Single start state

2° Stack empty before accepting

3° Each transition `push` or `pop`, but not both

Idea. For each pair of states $p, q \in Q$ of a PDA P, we have A_{pq} and

$$A_{pq} \text{ generates } x \Rightarrow P \text{ from } p \text{ with empty stack to } q \text{ with empty stack, reading } x$$

First, we discuss how to handle transitions

$$\forall p, q, r \in Q,\ A_{pq} \to A_{pr}A_{rq}$$

We let the

- x-axis: input string
- y-axis: stack height

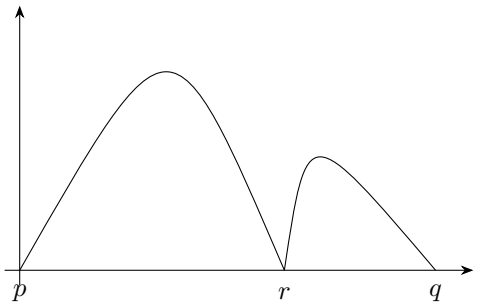


Figure 2.10: PDA transition $A_{pq} \to A_{pr}A_{rq}$

If we can go

$$\text{from } p \text{ to } r \text{ without changing stack}$$

and

$$\text{from } r \text{ to } q \text{ without changing stack}$$

then we can do

$$\text{from } p \text{ to } q \text{ without changing stack}$$

Next, we have

$$\forall p, q, r, s \in Q,\ a, b \in \Sigma_\varepsilon,\ t \in \Gamma$$

If,

$$(r, t) \in \delta(p, a, \varepsilon) \text{ and } (q, \varepsilon) \in \delta(s, b, t)$$

we discuss how to handle transitions

$$A_{pq} \to aA_{rs}b$$

Then we have

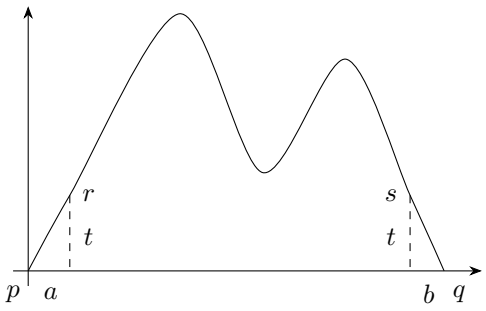


Figure 2.11: PDA transition $A_{pq} \to aA_{rs}b$

Finally, we have the following base case:

$$\forall p \in Q,\ A_{pp} \to \varepsilon$$

To follow the condition (1°), we give a new example

Example. Consider the language $L = \{0^n 1^n \mid n \geq 1\}$.

Now q_1 is not an accept state

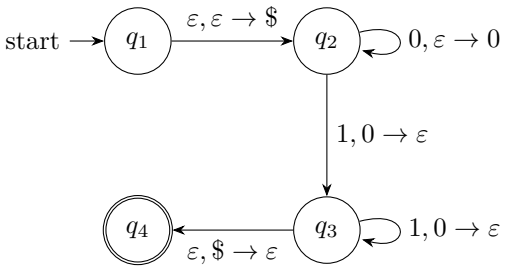


Figure 2.12: PDA for $A = \{0^n 1^n \mid n \geq 1\}$

Consider two elements in Γ

$$t_0 = \$, \quad t_1 = 0$$

- $t = \$$

p	r	s	q	t	a	b
1	2	3	4	$	ε	ε

then we can get the rule

$$A_{14} \to A_{23}$$

- $t = 0$

p	r	s	q	t	a	b
2	2	2	3	0	0	1
2	2	3	3	0	0	1

then we can get the rules

$$A_{23} \to 0A_{22}1$$
$$A_{23} \to 0A_{23}1$$

Other rules: 64 rules

$$A_{11} \to A_{11}A_{11}$$
$$A_{11} \to A_{12}A_{21}$$
$$A_{11} \to A_{13}A_{31}$$
$$A_{11} \to A_{14}A_{41}$$
$$\vdots$$

and

$$A_{11} \to \varepsilon$$
$$A_{22} \to \varepsilon$$
$$A_{33} \to \varepsilon$$
$$A_{44} \to \varepsilon$$

2.3.5 Procedure of converting PDA to CFL

Proposition 2.3.2. Given a PDA

$$P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$$

We construct a CFG with variables

$$\text{var}(G) = \{A_{pq} \mid p, q \in Q\}$$

and start variable

$$S = A_{q_0 q_{accept}}$$

With rules

1° Single start state

2° Stack empty before accepting

3° Each transition `push` or `pop`, but not both

A new start $q_s \to q_{s'}$ with $\varepsilon, \varepsilon \to \$$, and for any $q \in F$, we have $\varepsilon, a \to \varepsilon$ back to q, $\forall a \in \Sigma$. Then from any $q \in F$, we do $\varepsilon, \$ \to \varepsilon$ to q_a

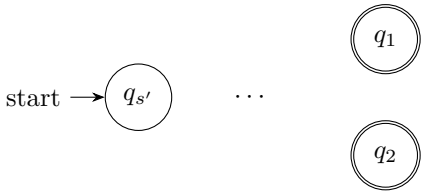
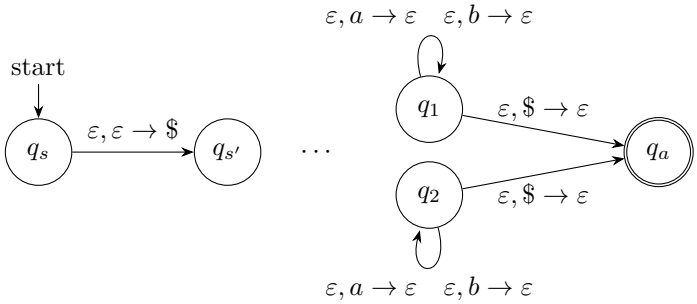


Figure 2.13: PDA with single accept state and empty stack before accepting

The new one will become



These is not enough to ensure condition (3°), we can do some modifications:

- To have each transition either `push` or `pop` (but not both), replace

  $$q_1 \xrightarrow{a,\, a \to b} q_2$$

  with the pair

  $$q_1 \xrightarrow{a,\, a \to \varepsilon} q_3, \qquad q_3 \xrightarrow{\varepsilon,\, \varepsilon \to b} q_2.$$

- Likewise, replace

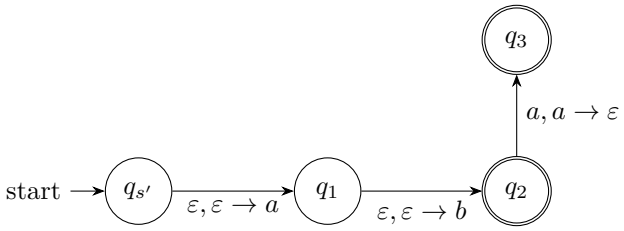
  $$q_1 \xrightarrow{a,\, \varepsilon \to \varepsilon} q_2$$

  with

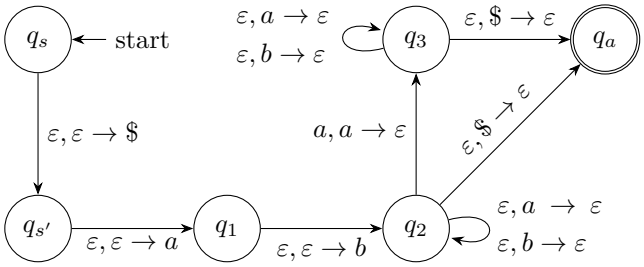
  $$q_1 \xrightarrow{a,\, \varepsilon \to X} q_3, \qquad q_3 \xrightarrow{\varepsilon,\, X \to \varepsilon} q_2,$$

  where X is a fresh stack marker introduced for this simulation.

For another example, consider the PDA



After the modification, we have



The new PDA will accept the string a but the original PDA rejects it. Hence, we need to modify something else:

- A new start $q_s \to q_{s'}$ with $\varepsilon, \varepsilon \to \$$
- A new state q_{pop} that have $\varepsilon, a \to \varepsilon$ back to q_{pop}, $\forall a$.
- For $q \in F$, add a transition $\varepsilon, \varepsilon \to \varepsilon$ from q to q_{pop}
- Add a new accept state q_a and a transition $\varepsilon, \$ \to \varepsilon$ from q_{pop} to q_a

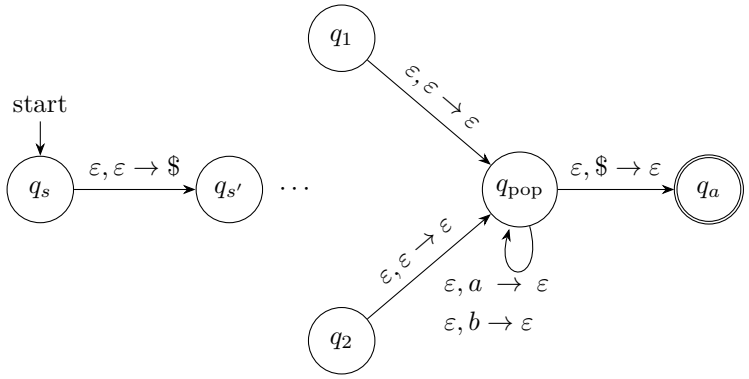


Figure 2.14: PDA with single accept state and empty stack before accepting

2.4 Deterministic Pushdown Automata

Lecture 7

PDA is non-deterministic in general. However, there is a special class of PDA called Deterministic Pushdown Automata (DPDA). From Ch.1 we know 2025-11-03

$$\text{DFA} \equiv \text{NFA}$$

but

$$\text{DPDA} \neq \text{PDA} \implies \text{CFL} \neq \text{DCFL}$$

Definition 2.4.1 (Deterministic Pushdown Automaton (DPDA)). A deterministic pushdown automaton (DPDA) is a 6-tuple

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

where

- Q: States
- Σ: Input alphabet
- Γ: Stack alphabet
- δ: Transition function
$$Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to (Q \times \Gamma_\varepsilon) \cup \{\emptyset\}$$
- $q_0 \in Q$: Start state
- $F \subset Q$: Set of accepting states

To build a DPDA, we first look at the different between PDA and DPDA.

As previously seen. For PDA,

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$$

Note. In DPDA, for $\forall q \in Q, a \in \Sigma, x, \gamma \in \Gamma$, at most and at least one of the following is true:

$$\delta(q, a, x) = (p, \gamma), \quad \delta(q, a, \varepsilon) = (p, \gamma), \quad \delta(q, \varepsilon, x) = (p, \gamma), \quad \delta(q, \varepsilon, \varepsilon) = (p, \gamma)$$

the rest must be $\emptyset$.

2.4.1 Acceptance, Rejection of DPDA

The Rejection of DPDA is similar to PDA, which should only happen when

- Not end at an accept state after the last symbol.
- DPDA fails to read the input
  1. `pop` an empty stack
  2. Endless ε-transition

Example. $L = \{0^n 1^n \mid n \geq 0\}$

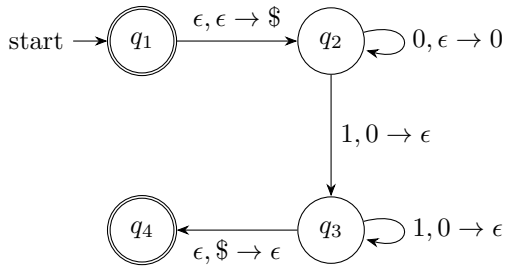


Figure 2.15: DPDA for $L = \{0^n 1^n \mid n \geq 0\}$

The Transition function is defined as follows:

	0			1			ϵ		
	0	\$	ϵ	0	\$	ϵ	0	\$	ϵ
q_1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$(q_2, \$)$
q_2	$\emptyset$	$\emptyset$	$(q_2, 0)$	(q_3, ϵ)	q_r	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
q_3	q_r	$\emptyset$	$\emptyset$	(q_3, ϵ)	$\emptyset$	$\emptyset$	$\emptyset$	(q_4, ϵ)	$\emptyset$
q_4	q_r	q_r	$\emptyset$	q_r	q_r	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
q_r	q_r	q_r	$\emptyset$	q_r	q_r	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

To find this transition table, for instance,

- consider the state q_1:

  $$\delta(q_1, \varepsilon, \varepsilon) = (q_2, \$)$$

  then we can implies that

  $$\delta(q_1, a, \gamma) = \delta(q_1, a, \varepsilon) = \delta(q_1, \varepsilon, \gamma) = \emptyset, \quad \forall a \in \Sigma = \{0, 1\},\ \gamma \in \Gamma = \{0, \$\}$$

- consider the state q_2:

  $$\delta(q_2, 1, 0) = (q_3, \varepsilon)$$

  then we can implies that

	0			1			ϵ		
	0	\$	ϵ	0	\$	ϵ	0	\$	ϵ
q_2	$\emptyset$	$\emptyset$	$(q_2, 0)$	(q_3, ϵ)	$\neq \emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

  due to

  $$\delta(q_2, 1, \varepsilon) = \delta(q_2, \varepsilon, \$) = \delta(q_2, \varepsilon, \varepsilon) = \emptyset$$

  Formally we have

  $$\delta(q_2, 1, \$) = (q_r, \varepsilon)$$

For the string 011, the computation of the DPDA is as follows:

$$q_1 \xrightarrow{\epsilon} q_2, \{\$\}\ q_1 \xrightarrow{0} q_2, \{0, \$\}\ q_1 \xrightarrow{1} q_3, \{\$\}\ q_1 \xrightarrow{\epsilon} q_4, \emptyset$$

Follow the graph,

$$\delta(q_4, 1, \epsilon) \text{ and } \delta(q_4, \epsilon, \epsilon) = \emptyset$$

hence, the DPDA rejects 011.

Chapter 3

The Church-Turing Thesis

Lecture 8

3.1 Turing Machines

2025-11-10

> **As previously seen.** To discuss the computability of problems. We need a more powerful model. We already have seen
>
> - Finite Automata (FA): with limited memory (states)
> - Pushdown Automata (PDA): unlimited memory with LIFO structure (stack)

We now introduce a new computational model called **Turing Machine** (TM), which has an unlimited tape as memory.

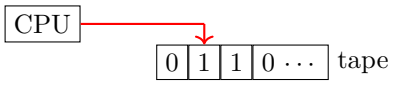


Figure 3.1: Illustration of a Turing Machine

A Turing Machine consists of these properties different from FA and PDA:

- write/read tape
- head that can move left/right on the tape
- unlimited tape length
- reject/accept take immediate effect
- machine can **never** halt

Example.

$$B = \{w\#w \mid w \in \{0,1\}^*\}$$

Remark. We can proof that this language is not CFL by pumping lemma for CFL.

Notation. $\sqcup$ is the blank symbol on the tape.

Idea. Zig-zag to the corresponding places on the two sides of the # and determine whether they match.

- Scan to check if ther is a #.
- Check w and w if they match.

$$\begin{array}{cccccccccccccc} \overset{\vee}{0} & 1 & 1 & 0 & 0 & 0 & \# & 0 & 1 & 1 & 0 & 0 & 0 & \sqcup \\ x & \overset{\vee}{1} & 1 & 0 & 0 & 0 & \# & 0 & 1 & 1 & 0 & 0 & 0 & \sqcup \\ x & 1 & 1 & 0 & 0 & 0 & \# & \overset{\vee}{x} & 1 & 1 & 0 & 0 & 0 & \sqcup \end{array}$$

Figure 3.2: Illustration of algorithm

Definition 3.1.1 (Turing Machine (TM)). A Turing Machine is a 7-tuple

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

where

- Q: States
- Σ: Input alphabet, where $\sqcup \notin \Sigma$
- Γ: Tape alphabet, where $\Sigma \subset \Gamma$ and $\sqcup \in \Gamma$
- δ: Transition function
$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$
- $q_0 \in Q$: Start state
- $q_{\text{accept}} \in Q$
- $q_{\text{reject}} \in Q,\ q_{\text{reject}} \neq q_{\text{accept}}$

The input

$$w = w_1 w_2 \cdots w_n \in \Sigma^*$$

will be put in the position $1, 2, \cdots, n$ of the tape, and the rest of the tape is filled with $\sqcup$.

Example.

$$L = \{0^{2^n} \mid n \geq 0\}$$

Idea. Cross off every second, and check if the remaining is even (except the last one).

$$\begin{array}{c} 0\underline{0}0\underline{0} \\ 0\underline{0} \\ 0 \end{array}$$

The procedure should be:

1° left $\to$ right, make remark on every second 0

2° if step 1° left with only one unmarked 0, accept

3° if step 1° left with odd #0 left, reject

4° move head to the leftmost

5° go to step 1°

The definition of the machine is

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_{\text{accept}}, q_{\text{reject}}\}$$
$$\Sigma = \{0\}$$
$$\Gamma = \{0, x, \sqcup\}$$

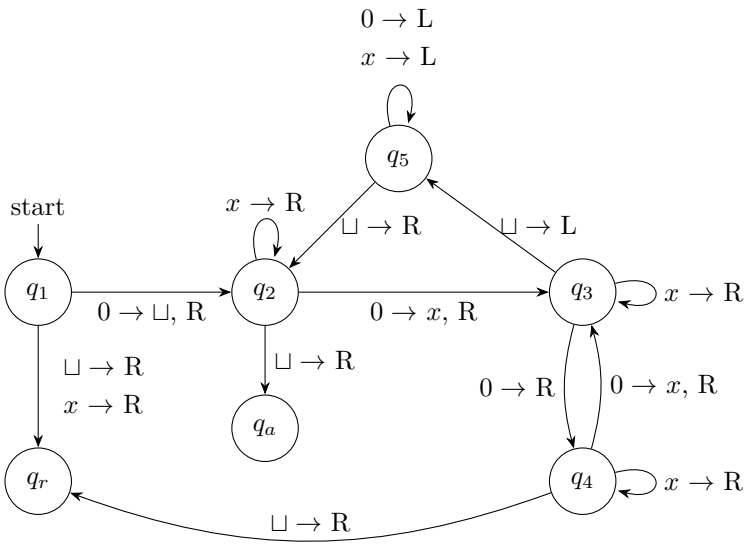


Figure 3.3: TM for $L = \{0^{2^n} \mid n \geq 0\}$

Notation.

$$0 \to R \equiv 0 \to 0, R$$

Consider the input 0000:

$q_1 0000$	$\sqcup q_2 000$	$\sqcup x q_3 00$	$\sqcup x 0 q_4 0$	$\sqcup x 0 x q_3$
$\sqcup x 0 q_5 x$	$\sqcup x q_5 0 x$	$\sqcup q_5 x 0 x$	$q_5 \sqcup x 0 x$	$\sqcup q_2 x 0 x$
$\sqcup x q_2 0 x$	$\sqcup x x q_3 x$	$\sqcup x x x q_3 \sqcup$	$\sqcup x x q_5 x$	$\sqcup x q_5 x x$
$\sqcup q_5 x x x$	$q_5 \sqcup x x x$	$\sqcup q_2 x x x$	$\sqcup x q_2 x x$	$\sqcup x x q_2 x$
$\sqcup x x x q_2$	$\sqcup x x x \sqcup q_a$			

The transition function table is as follows:

	0	x	$\sqcup$
q_1	$q_2, \sqcup, R$	q_{reject}, x, R	$q_{reject}, \sqcup, R$
q_2	q_3, x, R	q_2, x, R	$q_{accept}, \sqcup, R$
$\vdots$			

Note. There is no need for transition for q_{accept} and q_{reject} since the machine halts when it enters these states.

Idea. We can get the design idea of Turing Machine

- q_1 : mark the start by $\sqcup$
  - first element must be 0, otherwise, reject
  - Using $\sqcup$, so the start is known
- $q_2 \to q_3$: handle initial 00
- $q_3 \to q_4 \to q_3$: sequentially $00 \to 0x$
  - If not pairs (e.g., 0x0x0x), fails
  - This is the place of checking if # of remained zeros is even
- $q_3 \to q_5 \to q_2$ back to beginning
- first First 0 (or $\sqcup$) is considered the single final 0

  $$q_2 \to \cdots \to q_2 \to \cdots \to q_{accept}$$

  check if a single 0 is left in the string.

3.1.1 Configuration of Turing Machine

Definition 3.1.2 (current configuration)**.** The current configuration of a Turing Machine is represented as

$$uqv$$

where

- $u \in \Gamma^*$: the string on the left of the head
- $q \in Q$: the current state
- $v \in \Gamma^*$: the string on the right of the head

The head is reading the first symbol of v. If $v = \epsilon$, then the head is reading a blank symbol $\sqcup$.

Definition 3.1.3. $a, b, c \in \Gamma$, $u, v \in \Gamma^*$, $q_i, q_j \in Q$ then the transition from configuration

- If $\delta(q_i, b) = (q_j, c, L)$, then
  $$uaq_ibv \vdash uq_jacv$$
- If $\delta(q_i, a) = (q_j, b, L)$, then
  $$uaq_ibv \vdash uacq_jv$$

3.1.2 Turing Recognizable and Turing Decidable Languages

Definition 3.1.4 (Turing Recognizable). A language L is Turing recognizable if some Turing Machine M recognizes it.

For a Turing Machine there are three possible outcomes:

- Accept the input by entering q_{accept}
- Reject the input by entering q_{reject}
- Loop forever without halting

A language is very difficult to difficult to decide if the TM loops forever on some inputs. We now define a more restricted type of model, called **Decider**.

Definition 3.1.5 (Turing Decidable). A language L is Turing decidable if some Turing Machine M decides it.

We will discuss more about Decidability in later chapters (Ch.4).

3.1.3 Example of Turing Machine

Example. $L = \{w\#w \mid w \in \{0,1\}^*\}$

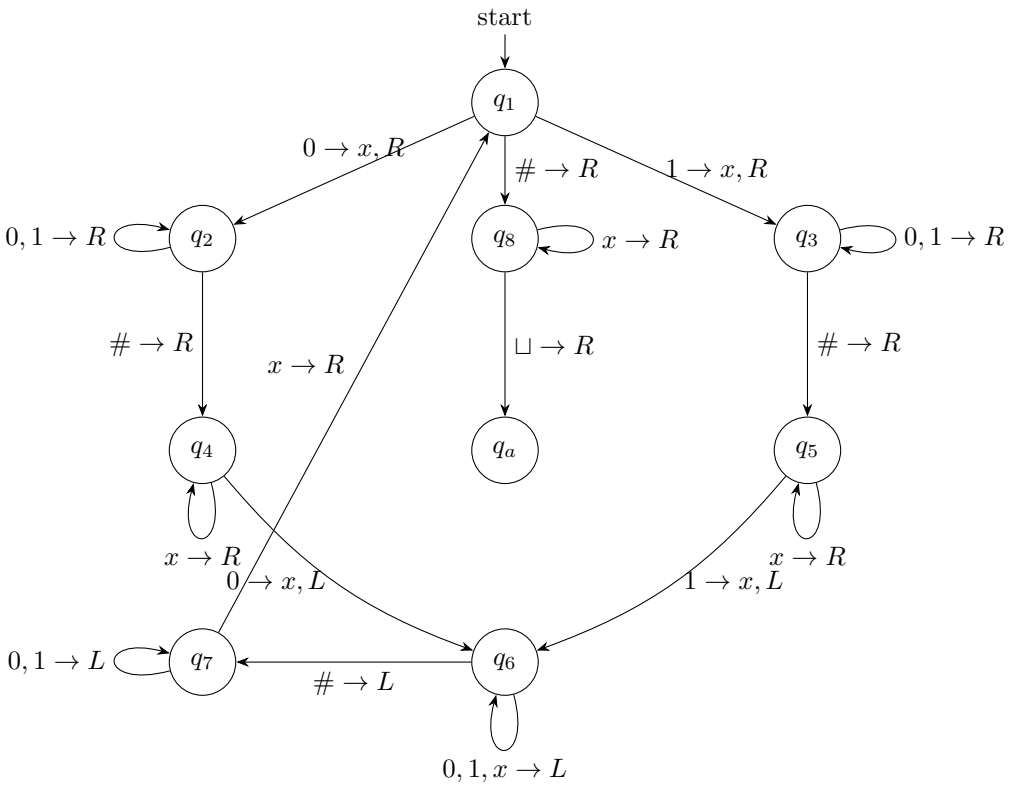


Figure 3.4: Turing Machine of $L = \{w\#w \mid w \in \{0,1\}^*\}$

Remark. Links to q_r are not shown

Simulate 01#01

$$\begin{array}{llll} q_1 01\#01 & xq_2 1\#01 & x1q_2\#01 & x1\#q_4 01 \\ x1q_6\#x1 & xq_7 1\#x1 & q_7 x1\#x1 & xq_1 1\#x1 \\ xxq_3\#x1 & xx\#q_5 x1 & xx\#xq_5 1 & xx\#q_6 xx \\ xxq_6\#xx & xq_7 x\#xx & xxq_1\#xx & xx\#q_8 xx \\ xx\#xxq_8\sqcup & xx\#xx\sqcup q_a & & \end{array}$$

Idea. The diagram:

$$q_1 \to q_2 \to q_4 \to q_6$$

check 0 at the same position of the two strings

$$q_1 \to q_3 \to q_5 \to q_6$$

check 1 at the same position of the two strings

Example. $C = \{a^i b^j c^k \mid i \times j = k,\ i, j, k \geq 1\}$

Idea. The procedure should be:

1° check if the input is $a^+b^+c^+$

2° back to the leftmost a

3° fix an a, for each b, cross off a c

4° store b back, cancel one a, repeat step 3

- Step 1 can be done by a DFA (as DFA is a special case of TM).
- Step 2 can be done by moving left until $\sqcup$ is reached.
- Step 3 is similar to previous examples.

Example. $E = \{\#x_1\#x_2\cdots\#x_l \mid x_i \in \{0,1\}^*,\ x_i \neq x_j\}$

Idea. Sequentially compare every pairs

$$\begin{array}{l} x_1x_2, x_1x_3, \ldots, x_1x_l \\ x_2x_3, \ldots, x_2x_l \\ \vdots \\ x_{l-1}x_l \end{array}$$

For $x_i,\ x_j$, mark #'s strings by $\dot{\#}$ i.e.

$$\dot{\#}x_1\#x_2\dot{\#}x_3 : x_1 \text{ and } x_3 \text{ are compared}$$

We can copy x_i, x_j to the right end of the tape and compare them there with the pattern of $w\#w$.

Lecture 9

2025-11-24

3.2 Multi-tape Turing machines

3.2.1 Variants of Turing machines

Example. The transition function may be defined as

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$$

Notation. S stands for "stay", meaning the head does not move.

This kind of Turing machine can be simulated by a standard Turing machine.

$$\begin{aligned} q_1, a \to q_2, b, S \quad &\equiv q_1, a \to q_{\text{temp}}, b, R \\ &\quad q_{\text{temp}}, \gamma \to q_2, \gamma, L \quad \forall \gamma \in \Gamma \end{aligned}$$

3.2.2 Multi-tape Turing machines

In this variant, there are multiple tapes, each with its own head.

- Input: In the first tape.
- Other tapes: Blank initially.

Definition 3.2.1 (Multi-tape Turing machine)**.** Transition function:

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k \quad \text{(for } k \text{ tapes)}$$

Example. Given $w = 0^{2n},\ n \geq 1 \ \Rightarrow$ Generate ww.

Idea. We have the following simulation:

- Copy w into the second tape.
- Check if $|w|$ is even.
- copy w from the second tape to the first tape (append).

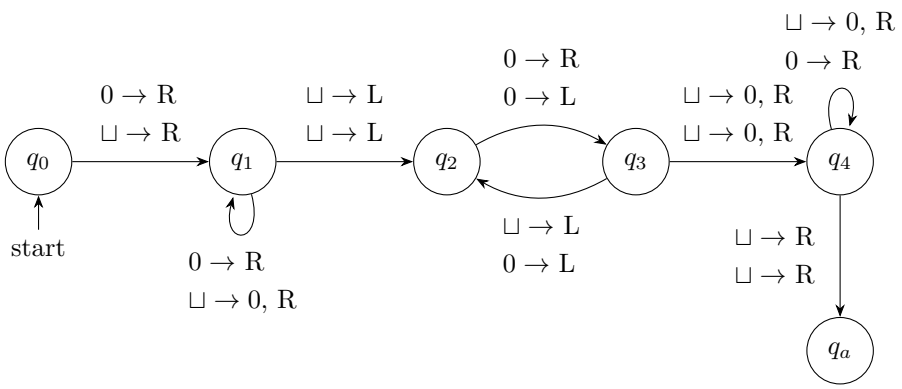


Figure 3.5: Multi-tape Turing machine to compute ww from $w = 0^{2n}$

For the simulation details,

- $q_0 \to q_1$: Use $\sqcup$ to indicate the beginning of the second tape.
- loop in q_1: Copy w from the first tape to the second tape.
- $q_2 \to q_3 \to q_2$:

  1° Check if length of w is even.

  2° Head of the first tape zig-zag between last 0 and then $\sqcup$ after.

  3° Head of the second tape moves to the beginning.

  > **Remark.** If length of w is even, we will at q_3 when reaching the beginning of the second tape.

- q_4: copy w from the second tape to the first tape (append).

Below is the illustration of the simulation 0000:

$$q_0\,\begin{smallmatrix}0&0&0&0\\ \sqcup&\sqcup&\sqcup&\sqcup\end{smallmatrix} \qquad \begin{smallmatrix}0\\ \sqcup\end{smallmatrix} q_1\,\begin{smallmatrix}0&0&0\\ \sqcup&\sqcup&\sqcup\end{smallmatrix} \qquad \cdots \qquad \begin{smallmatrix}0&0&0&0\\ \sqcup&0&0&0\end{smallmatrix} q_1\,\begin{smallmatrix}\sqcup\\ \sqcup\end{smallmatrix}$$

$$\begin{smallmatrix}0&0&0\\ \sqcup&0&0\end{smallmatrix} q_2\,\begin{smallmatrix}0\\ 0\end{smallmatrix} \qquad \begin{smallmatrix}0&0\\ \sqcup&0\end{smallmatrix} \begin{smallmatrix}0&0\\ q_3&0\end{smallmatrix} \qquad \begin{smallmatrix}0\\ \sqcup\end{smallmatrix} \begin{smallmatrix}0&0\\ q_2&0\end{smallmatrix} \begin{smallmatrix}q_2&0\\ 0&\end{smallmatrix} \qquad \begin{smallmatrix}0&0&0&0\\ q_3&\sqcup&0&0\end{smallmatrix} \begin{smallmatrix}q_3\\ 0\end{smallmatrix}$$

$$\begin{smallmatrix}0\\ 0\end{smallmatrix} \begin{smallmatrix}0&0&0&0\\ q_4&0&0&0\end{smallmatrix} \begin{smallmatrix}q_4\\ \sqcup\end{smallmatrix} \qquad \cdots \qquad \begin{smallmatrix}0&0&0&0\\ 0&0&0&0\end{smallmatrix} \begin{smallmatrix}0&0&0&0\\ q_4&&&\end{smallmatrix}$$

Below is the illustration of the simulation 000:

$$q_0\,\begin{smallmatrix}0&0&0\\ \sqcup&\sqcup&\sqcup\end{smallmatrix} \qquad \begin{smallmatrix}0\\ \sqcup\end{smallmatrix} q_1\,\begin{smallmatrix}0&0\\ \sqcup&\sqcup\end{smallmatrix} \qquad \cdots \qquad \begin{smallmatrix}0&0&0\\ \sqcup&0&0\end{smallmatrix} q_1\,\begin{smallmatrix}\sqcup\\ \sqcup\end{smallmatrix}$$

$$\begin{smallmatrix}0&0\\ \sqcup&0\end{smallmatrix} q_2\,\begin{smallmatrix}0\\ 0\end{smallmatrix} \qquad \begin{smallmatrix}0\\ \sqcup\end{smallmatrix} \begin{smallmatrix}0&0\\ q_3&0\end{smallmatrix} \begin{smallmatrix}q_3\\ 0\end{smallmatrix} \qquad \begin{smallmatrix}0&0\\ q_2&\sqcup\end{smallmatrix} \begin{smallmatrix}q_2&0\\ 0&\end{smallmatrix}$$

> **Note.** Due to the properties of "deterministic",
>
> $$\delta(q_0,0,0),\ \delta(q_0,0,\sqcup)\ \delta(q_0,\sqcup,0),\ \delta(q_0,\sqcup,\sqcup)$$
>
> those not specified transitions go to q_r.

3.2.3 Multi-tape TM ≡ Single-tape TM

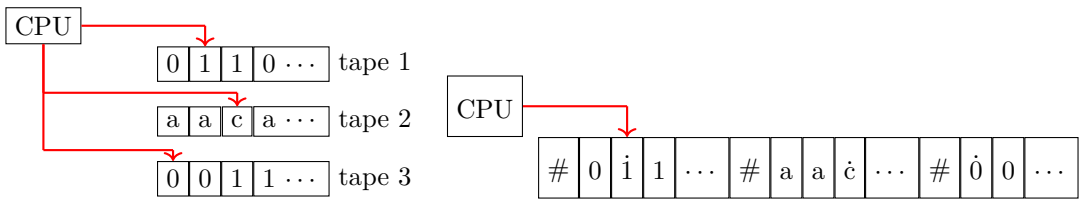


Figure 3.6: Simulation of multi-tape Turing machine by single-tape Turing machine

- # can be used to separate different tapes.
- $\dot{c}$ can be used to indicate the head position of each tape.
- $\Gamma' = \{\Gamma, \dot{\Gamma}\}$

Example. Right shifting a sequece w.

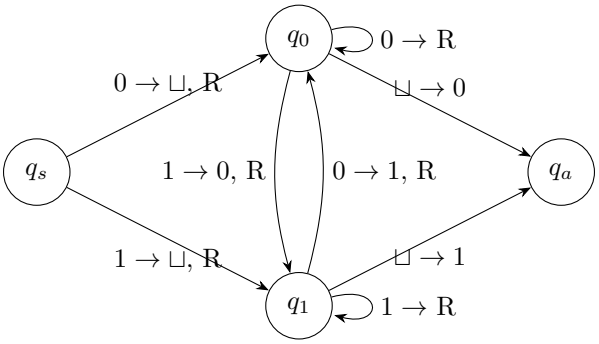


Figure 3.7: Single-tape simulation Turing machine to right shift a sequence

Note. Because Γ is finite, the simulation must succeed by add state for each $\gamma \in \Gamma$.

3.3 Nondeterministic Turing Machines

Definition 3.3.1 (Nondeterministic Turing machine)**.** Transition function:

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

In NTM, by definition w is accepted if any branch works, which means unless all branches are finite,

$$\text{NTM} \longrightarrow \text{accept or loop}$$

Thus, NTM is an "acceptor".

Example. $A = \{w \text{ contain } aab\}$

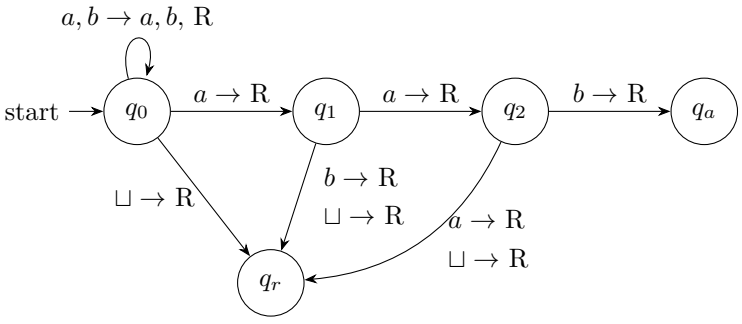


Figure 3.8: Nondeterministic Turing machine to accept $A = \{w \text{ contain } aab\}$

Note. Determine where aab starts nondeterministically.

Example. $L = \{0^n \mid n\text{-composite number}\}$

- Nondeterministically guess p, q, sequentially try from 2 to $n - 1$.
- Check if $n = p \times q$

3.3.1 NTM ≡ TM

A language is Turing-recognizable ⇒ it is recognized by a NTM, due to TM being a special case of NTM. Proof done for ⇐ direction.

For the other direction, we need to simulate NTM by TM. Like NFA we use a tree structure to represent the computation finite # branches. To traverse the tree, we can use BFS or DFS.

Remark. BFS is preferred, because DFS may get stuck in an infinite branch.

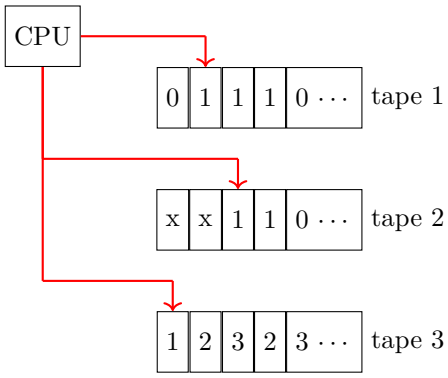


Figure 3.9: Simulation of NTM by TM using multi-tape Turing machine

- Tape 1: Store the input, never alter.
- Tape 2: Simulate the current branch up to certain layer by copying Tape 1.
- Tape 3: Store the path to a node.

Definition 3.3.2. Suppose max # branches is 3 at each node. If the content of the 3rd tape is 231 that means

root → 2nd child → 3rd child → 1st child

Hence, NTM can be simulated by 3-tape TM, and we have shown that multi-tape TM can be simulated by single-tape TM. Thus, NTM ≡ TM. ■

Corollary 3.3.1. NTM is a decider if all branches halt on all inputs.

Language decidable ⇔ some NTM decides it

Proof. We separate into two directions.

⇒ One TM decides it and a TM is an NTM. This TM halts on all inputs (one branch)

⇐ Now NTM terminates on all branches. We can construct a TM to decide the language

- each branch is finite every input halts ∃ a finite max length.
- # branches finite. The tree to process this input is finite.
- Thus the three-tape TM used earlier can accept/reject the input in a finite number of steps.

■

3.4 Hilbert's problems

Informally, an algorithm is a collection of instructions. Not until 1900 did Hilbert propose 23 unsolved problems in mathematics. The 10th problem is:

> *Is there a general method to determine whether a given polynomial equation with integer coefficients has an integer solution?*

Hilbert didn't use the word "algorithm" but "general method". However, Hilbert explicitly asked the algorithm be "devises".

3.4.1 Church-Turing thesis

This is proposed by Alonzo Church and Alan Turing in 1936.

> *Any function which would naturally be regarded as computable can be computed by a Turing machine.*

Definition 3.4.1.

$$\text{Intuitive Algorithm} \equiv \text{Turing machine algorithm}$$

3.4.2 Hilbert's 10th problem

Using the Church-Turing thesis, we have a question:

Question. Define

$$D = \{P \mid P : \text{polynomial with integer root}\}$$

is D decidable?

We first simplify the problem to one variable case.

$$D_1 = \{P \mid P : \text{polynomial of } x \text{ with integer root}\}$$

If we try all integers one by one, it may not halt if no integer root exists. Thus, D_1 is Turing-recognizable but not decidable.

However, it can be proved that roots of a 1-variable polynomial is within the range

$$-M \le x \le M, \quad M = \pm k \frac{\max |c_i|}{|c_1|}$$

where

- k: # of terms
- c_i: coefficients of the polynomial
- c_1: leading coefficient

For instance, for $4x^3 - 2x^2 + x - 7$, we have

$$M = \pm 4 \cdot \frac{7}{4} = \pm 7$$

The multi-variable case is much more complicated. In 1970, Matiyasevich proved that no such algorithm exists. Thus, we have the conclusion:

Theorem 3.4.1 (Matiyasevich, 1970)**.** D is not decidable.

3.4.3 Description of Turing machines

A Turing machine can be describe in 3 levels:

- **High-level description**: Describe the operations of the Turing machine without manage the tape and head.
- **Implementation-level description**: Describe how the Turing machine move the head.
- **Formal description**: Specify the states, input alphabet, tape alphabet, transition function, start state, accept state, and reject states of the Turing machine.

Example. Describe a Turing machine that decides the language

$$A = \{\langle G\rangle \mid G : \text{ a connected undirected graph}\}$$

- **High-level description**: We separate into three steps.

  1° Mark node in G.

  2° Repeat until no new nodes marked:

  – For each node in G, if it is marked, mark all its neighbors.

  3° If all nodes marked: accept, otherwise: reject.

- **Implementation-level description**:

  $$\langle G\rangle = (1,2,3,4)((1,2),(2,3),(3,1),(1,4))$$

  is the input format.

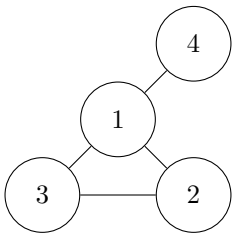


Figure 3.10: Graph representation on tape

  – The first step is to check if the input is in the correct format.

  – In the first step we begin with seeing if the first part of the input $\langle G\rangle$ includes distinct numbers (as node IDs should be different)

  – This is similar to an example before.

  $$\{\#x_1\#x_2\#\cdots x_n\# \mid x_i \in \{0,1\}^*, x_i \neq x_j\}$$

  – Then we can talk about how the head is moved.

Chapter 4

Decidability

Lecture 10

2025-12-1

4.1 Decidability

If we have a algorithm, we want to check if the problem is solvable or not on the computer. We need a TM to decide it, i.e. accept or reject in finite # steps.

Definition. We first give some definitions of the languages.

Definition 4.1.1 (A). A is the language

$$A = \{\langle M, w\rangle \mid M \text{ accepts } w\}$$

Definition 4.1.2 (E). E is the language

$$E = \{\langle M\rangle \mid M : L(M) = \emptyset\}$$

Definition 4.1.3 (EQ). EQ is the language

$$EQ = \{\langle M_1, M_2\rangle \mid M_1, M_2 : L(M_1) = L(M_2)\}$$

Definition 4.1.4 ($HALT$). $HALT$ is the language

$$HALT = \{\langle M, w\rangle \mid M : L(M) \text{ halts on } w\}$$

Definition 4.1.5 ($REGULAR$). $REGULAR$ is the language

$$REGULAR = \{\langle M\rangle \mid M : L(M) \text{ is regular}\}$$

Definition 4.1.6 (ALL). ALL is the language

$$ALL = \{\langle M\rangle \mid M : L(M) = \Sigma^*\}$$

4.1.1 A_{DFA} is Decidable

Example. Consider the language

$$A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts } w\}$$

The input of the problem is a pair of a DFA and a string, note that both can be encoded as a string.

Idea. Input: $\langle B, w\rangle$ where B is a DFA and w is a string.

1° Simulate B on input w.

2° If B accepts, then accept. If B rejects, then reject.

We first put

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

into the tape, then we put w after it. Then checking if $w \in \Sigma^*$. Then simulate w according to δ. After reading the whole w, check if the current state is in F.

4.1.2 A_{NFA} is Decidable

Example. Consider the language

$$A_{\text{NFA}} = \{\langle B, w\rangle \mid B \text{ is an NFA that accepts } w\}$$

We can use the subset construction to convert NFA to DFA, then use the previous algorithm.

4.1.3 A_{REX} is Decidable

Example. Consider the language

$$A_{\text{REX}} = \{\langle R, w\rangle \mid R: \text{ regular expression generates } w\}$$

We first convert R to an NFA B using the standard construction, then use the previous algorithm.

Remark. We have a procedure to convert a regular expression to an equivalent NFA. Then we can use the algorithm for A_{NFA} to decide A_{REX}.

The key idea is that we have procedures to convert of regular languages is in **finite** steps.

4.1.4 E_{DFA} is Decidable

Example. Consider the language

$$E_{\text{DFA}} = \{\langle A\rangle \mid A: \text{ DFA, } L(A) = \emptyset\}$$

i.e. A accepts no strings.

Idea. Input: $\langle A\rangle$ where A is a DFA.

DFA accepts something $\Leftrightarrow$ reaching a final state from q0 after several links

1° Mark q_0.

2° Repeat until no new state is marked:

- For each transition $\delta(q, a) = p$, if q is marked, then mark p.

3° If no $q \in F$ is marked, then accept. Otherwise, reject.

There are at least one new $q \in Q$ marked in each iteration, so the algorithm halts in at most $|Q|$ iterations.

4.1.5 EQ_{DFA} is Decidable

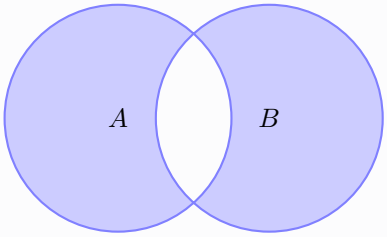
Example. Consider the language

$$EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A, B : \text{ DFA}, \ L(A) = L(B)\}$$

Idea. Let a DFA C be the **exclusive or** of A and B.

$$L(A) = L(B) \Leftrightarrow L(C) = \emptyset$$

Exclusive or of A and B



Formally, we can construct C as follows:

$$L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$$

- B is DFA $\Rightarrow \overline{B}$ is DFA.
- A, B DFA $\Rightarrow A \cap B$ is DFA.

4.1.6 A_{CFG} is Decidable

Example. Consider the language

$$A_{\text{CFG}} = \{\langle G, w\rangle \mid G : \text{ CFG that generates } w\}$$

Note. The possible derivation of w is ∞, but for a CFG in Chomsky Normal Form (CNF), any derivation of w has exactly $2|w| - 1$ steps. If $q = |R|$, the number of variables, then the number of possible derivations is at most $q^{2|w|-1}$.

Idea. Input: $\langle G, w\rangle$ where G is a CFG.

1° Convert G to an equivalent CFG G' in CNF.

2° Check all $q^{2|w|-1}$ possible derivations.

4.1.7 E_{CFG} is Decidable

Example. Consider the language

$$E_{\mathrm{CFG}} = \{\langle G\rangle \mid G:\ \text{CFG},\ L(G) = \emptyset\}$$

Idea. Input: $\langle G\rangle$ where G is a CFG. We use the bottom-up approach to find all variables that can generate some terminal strings.

- From $A \to a$ we search for

$$B \to A$$

We repeat this

1° Mark all the terminals.

2° Repeat until no new variable is marked:

- if

$$A \to U_1U_2\cdots U_k$$

and

$$\text{all } U_1, U_2, \ldots, U_k \text{ are marked}$$

then mark A.

- If start variable is not marked, accept. Otherwise, reject.

Number of variables is finite, so the algorithm halts in finite steps. Furthermore, each iteration is finite procedure with checking all the rule.

4.1.8 EQ_{CFG} is Undecidable

Example. Consider the language

$$EQ_{\mathrm{CFG}} = \{\langle G, H\rangle \mid G, H:\ \text{CFG},\ L(G) = L(H)\}$$

Because CFL is not closed under the complement operation, we cannot use the same idea as EQ_{DFA}. We should use the technique of reduction to prove it is undecidable (Ch.5).

Converting PDA to TM is not really work, due to the fact that PDA has infinite stack, one can be stuck in the operation of `push()` some symbols. We can only find the corresponding grammar G for the CFL. Then running the TM that decides A_{CFG} on input $\langle G, w\rangle$.

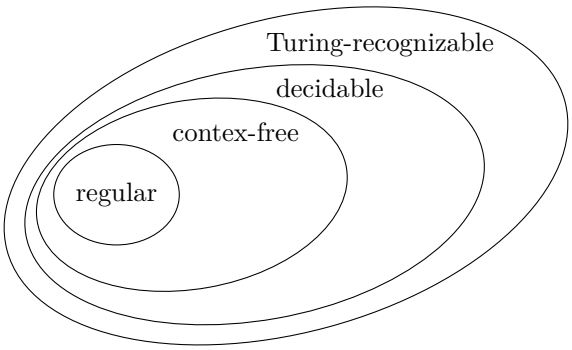


Figure 4.1: Classes of Languages

4.2 Halting Problem

In program verification is in general undecidable (unsolvable). Here is a classic example of undecidable problem, the **Halting Problem**.

4.2.1 Diagonalization Method

Definition 4.2.1. Two set are equal if elements can be paired up.

Giving an example

Definition 4.2.2 (one-to-one function)**.** A fucntion $f : A \to B$ is one-to-one if

$$f(a) \neq f(b) \text{ if } a \neq b$$

Definition 4.2.3 (onto function)**.** A function $f : A \to B$ is onto if

$$\forall b \in B,\ \exists a \in A \text{ such that } f(a) = b$$

Definition 4.2.4 (correspondence)**.** A **correspondence** between two sets A and B is a function $f : A \to B$ that is both one-to-one and onto.

Example. Consider the function

$$f(a) = a^2, \text{ where } A = (-\infty, \infty) \text{ and } B = (-\infty, \infty)$$

This is not an onto function because for $b = -1$, there is no $a \in A$ such that $f(a) = b$.

Example. Consider the function

$$f(a) = a^2, \text{ where } A = [0, \infty) \text{ and } B = [0, \infty)$$

Becoming an onto function. However, it is not one-to-one because $f(1) = f(-1) = 1$.

Example. Consider the function

$$f(a) = a^3, \text{ where } A = (-\infty, \infty) \text{ and } B = (-\infty, \infty)$$

This is a correspondence because it is both one-to-one and onto.

Remark. Correspondence is a way to pair the elements in two sets. If there is a correspondence between two sets, then they have the same cardinality (size).

Definition 4.2.5 (countable)**.** A set A is **countable** if there is a correspondence between A and $\mathbb{N}$ or a finite subset of $\mathbb{N}$.

Theorem 4.2.1. Consider the set of rational numbers $\mathbb{Q}$, where

$$\mathbb{Q} = \left\{ \frac{m}{n} \;\middle|\; m, n \in \mathbb{N} \right\}$$

which is countable.

Proof. Here is one way to list all the rational numbers:

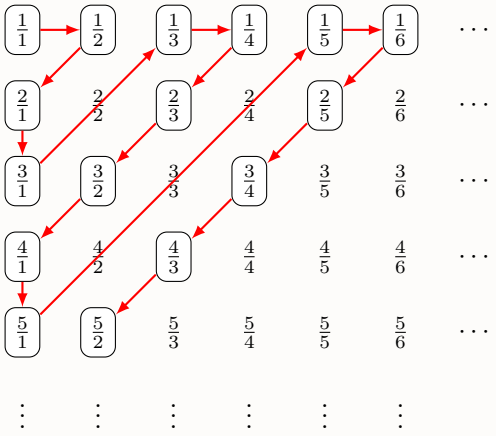


Figure 4.2: Counting the rational numbers

However, for the set of real numbers $\mathbb{R}$, it is uncountable. We can use the diagonalization method to prove it. ■

Theorem 4.2.2. Consider the set of real numbers $\mathbb{R}$, which is uncountable.

Proof. Assume $\mathbb{R}$ is countable, then we can list all the real numbers, we can construct a table of the corresponing decimal as follow:

Index	Real Number
1	$0.d_{11}d_{12}d_{13}d_{14}d_{15}\ldots$
2	$0.d_{21}d_{22}d_{23}d_{24}d_{25}\ldots$
3	$0.d_{31}d_{32}d_{33}d_{34}d_{35}\ldots$
4	$0.d_{41}d_{42}d_{43}d_{44}d_{45}\ldots$
5	$0.d_{51}d_{52}d_{53}d_{54}d_{55}\ldots$
$\vdots$	$\vdots$

Figure 4.3: Listing all the real numbers

Then we consider a new real number $r = 0.r_1r_2r_3\ldots$ where

$$r_i = \begin{cases} 5 & \text{if the } i\text{-th digit of } f(i) \neq 5 \\ 6 & \text{if the } i\text{-th digit of } f(i) = 5 \end{cases}$$

By construction,

$$r \neq f(n),\ \forall n \in \mathbb{N}$$

But $r \in \mathbb{R}$, which contradicts our assumption. Therefore, $\mathbb{R}$ is uncountable. ■

Remark. To avoid

$$1 = 0.999999\ldots$$

we can simply avoid using $0, 9$ in our construction.

4.2.2 NOT Turing-Recognizable

We known that Σ^* is countable, because we can list all the strings in order of length. Also, the set of all **TMs is countable**, because each TM can be encoded as a string (i.e. set of TMs is a subset of Σ^*).

Now, let

$$\begin{cases} L: \text{ set of all languages over } \Sigma \\ B: \text{ set of all infinite binary sequences} \end{cases}$$

For any language $A \subseteq \Sigma^*$, we can construct a corresponding infinite binary sequence $\chi_A = b_1 b_2 b_3 \ldots$ where

$$b_i = \begin{cases} 1 & \text{if } s_i \in A \\ 0 & \text{if } s_i \notin A \end{cases}$$

where $s_1, s_2, s_3, \ldots$ is the enumeration of all strings in Σ^*. This construction is in diagonalization way. This gives a correspondence between L and B. Since B is uncountable (by diagonalization method), L **is also uncountable**.

Each Turing machine corresponds to a language that it recognizes. Since the set of all TMs is countable, the set of all Turing-recognizable languages is also countable. Therefore, there are languages that are not Turing-recognizable.

4.2.3 Halting Problem is Undecidable

Racall that halting problem is defined as

Theorem 4.2.3. We define the language

$$A_{\text{TM}} = \{\langle M, w\rangle \mid M: \text{ TM that accept } w\}$$

The language A_{TM} is undecidable.

Proof. Assume H is a TM that decides A_{TM}. Then we get

$$H(\langle M, w\rangle) = \begin{cases} \text{accept} & \text{if } M \text{ accepts } w \\ \text{reject} & \text{Otherwise} \end{cases}$$

Then we can construct a new TM D with H by running it on input $\langle M, \langle M\rangle\rangle$. The behavior of D is defined as follows:

$$D(\langle M\rangle) = \begin{cases} \text{accept} & \text{if } H(\langle M, \langle M\rangle\rangle) = \text{reject} \\ \text{reject} & \text{if } H(\langle M, \langle M\rangle\rangle) = \text{accept} \end{cases}$$

which can be simplified as

$$D(\langle M\rangle) = \begin{cases} \text{accept} & \text{if } M \text{ rejects } \langle M\rangle \\ \text{reject} & \text{if } M \text{ accepts } \langle M\rangle \end{cases}$$

But we have a contradiction when we run D on input $\langle D \rangle$:

$$D(\langle D \rangle) = \begin{cases} \text{accept} & \text{if } D \text{ rejects } \langle D \rangle \\ \text{reject} & \text{if } D \text{ accepts } \langle D \rangle \end{cases}$$

■

Note. The diagonalization method is use in here again. Set of TMs is countable, so we can list all the TMs as $M_1, M_2, M_3, \ldots$. Then we can construct a table as follows:

	M_1	M_2	M_3	$\cdots$
$\langle M_1 \rangle$	b_{11}	b_{12}	b_{13}	$\cdots$
$\langle M_2 \rangle$	b_{21}	b_{22}	b_{23}	$\cdots$
$\langle M_3 \rangle$	b_{31}	b_{32}	b_{33}	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Figure 4.4: Listing all the TMs and their behavior on their own encoding

where

$$b_{ij} = \begin{cases} 1 & \text{if } M_j \text{ accepts } \langle M_i \rangle \\ 0 & \text{if } M_j \text{ rejects, or loops on } \langle M_i \rangle \end{cases}$$

Then we know H decides A_{TM} as it is a decider, we have

	M_1	M_2	M_3	$\cdots$
$\langle M_1 \rangle$	b_{11}	b_{12}	b_{13}	$\cdots$
$\langle M_2 \rangle$	b_{21}	b_{22}	b_{23}	$\cdots$
$\langle M_3 \rangle$	b_{31}	b_{32}	b_{33}	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Figure 4.5: Listing all the TMs and their behavior on their own encoding

where

$$b_{ii} = \begin{cases} A & \text{if } M_i \text{ accepts } \langle M_i \rangle \\ R & \text{if } M_i \text{ rejects } \langle M_i \rangle \end{cases}$$

Then we can construct a new TM D which output the opposite of the diagonal entries, then we get a contradiction when we run D on input $\langle D \rangle$.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\ldots$	$\langle D \rangle$
M_1	R			
M_2		R		
			$\ddots$	
D				?

Figure 4.6: Contradiction on the diagonal entries

4.2.4 co-Turing-Recognizable

Definition 4.2.6 (co-Turing-recognizable)**.** A language L is **co-Turing-recognizable** if its complement $\overline{L}$ is Turing-recognizable.

Theorem 4.2.4. A language L is decidable if and only if it is both Turing-recognizable and co-Turing-recognizable.

Proof. We seperately prove the two directions.

$\Rightarrow$ If L is decidable, then there is a TM M that decides L. We can use M to recognize L and $\overline{L}$. Hence, L is both Turing-recognizable and co-Turing-recognizable.

$\Leftarrow$ Now $A, \overline{A}$ are both Turing-recognizable by TM M_1 and M_2 respectively. We can construct a new TM M that decides L as follows:

1° On input w, run M_1 and M_2 in parallel on input w.

2° If M_1 accepts, then accept. If M_2 accepts, then reject.

Since $w \in L$ or $w \notin L$, either M_1 or M_2 will eventually accept w. Thus, M halts on all inputs, and decides L.

Proof complete. ■

Chapter 5

Reducibility

Lecture 11

As previously seen. 2025-12-1

$$A_{\mathrm{TM}} \text{ is undecidable.}$$

We want to prove that other languages are undecidable. We can do a technique called **reducibility**.

5.1 Reducibility

Definition 5.1.1 (reduction). Let A and B be languages. We say that A is **mapping reducible** to B, written $A \leq_m B$, if there exists a computable function $f : \Sigma^* \to \Sigma^*$ such that for every $w \in \Sigma^*$,

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

The function f is called a **reduction** from A to B. Which means the converting process from an instance of problem A to an instance of problem B, and B can be used to solve A.

Theorem 5.1.1. If $A \leq_m B$

$$B \text{ is decidable } \Rightarrow A \text{ is decidable.}$$

and

$$A \text{ is undecidable } \Rightarrow B \text{ is undecidable.}$$

5.1.1 E_{TM} is Undecidable

Example. Consider the language

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

Assume E_{TM} is decidable. We can get a decider R for E_{TM}. From this, we can construct a decider S for A_{TM} as follows:

- On input $\langle M, w \rangle$, where M is a TM and w is a string:
  1. Construct a new TM M_1 as follows:

$$L(M_1) = \emptyset \Leftrightarrow M \text{ does not accept } w \tag{1}$$

2. Run R on input $\langle M_1 \rangle$.
3. If R accepts, then reject. If R rejects, then accept.

This construction is valid because of (1). Thus, if we had a decider for E_{TM}, we could construct a decider for A_{TM}. But we know that A_{TM} is undecidable, so our assumption that E_{TM} is decidable must be false. Therefore, E_{TM} is undecidable.

Note. M_1 takes input x and have

1. If $x \neq w$, then reject.
2. If $x = w$, run M on input w and accept if M accepts w.

Clearly,

$$L(M_1) = \emptyset \text{ or } \{w\}$$

We see that

$$\begin{cases} M \text{ accepts } w & \Rightarrow L(M_1) = \{w\} \neq \emptyset \\ L(M_1) \neq \emptyset & \Rightarrow M \text{ accepts } w \end{cases}$$

Thus, condition (1) holds.

5.1.2 REGULAR$_{\text{TM}}$ is Undecidable

Example. Consider the language

$$\text{REGULAR}_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}.$$

As before, assume this language is decidable and has a decider R. We can construct a decider S for A_{TM} as follows:

- On input $\langle M, w \rangle$, where M is a TM and w is a string:
  1. Construct a new TM M_2 recognize:
  $$\begin{cases} \text{a regular language} & \text{if } M \text{ accepts } w \\ \text{a non-regular language} & \text{if } M \text{ rejects } w \end{cases} \tag{1}$$
  2. Run R on input $\langle M_2 \rangle$.
  3. If R accepts, then accept. If R rejects, then reject.
- Then we get S
  $$\begin{cases} S \text{ accepts} & \text{if } M \text{ accepts } w \\ S \text{ rejects} & \text{if } M \text{ rejects } w \end{cases}$$
  which is a decider for A_{TM}. Combining the R and M_2 we can get a decider for A_{TM}. Getting a contradiction. Thus, REGULAR$_{\text{TM}}$ is undecidable.

Note. M_2 recognizes the language

$$L(M_2) = \begin{cases} \Sigma^* & \text{if } M \text{ accepts } w \\ \{0^n 1^n \mid n \geq 0\} & \text{if } M \text{ rejects } w \end{cases}$$

We know that Σ^* is regular and $\{0^n 1^n \mid n \geq 0\}$ is not regular. Thus, condition (1) holds.

The implementation of M_2 on input x is as follows:

1. If x is in the form 0^n1^n, then accepts.
2. If x is not in the form 0^n1^n, then simulate M on input w, and accept if M accepts w.

5.1.3 $\mathbf{EQ_{TM}}$ is Undecidable

Example. Consider the language

$$EQ_{\text{TM}} = \{\langle M_1, M_2\rangle \mid M_1, M_2 : \text{ TM}, \ L(M_1) = L(M_2)\}.$$

Assume EQ_{TM} is decidable. We can get a decider R for EQ_{TM}. From this, we can construct a decider S for E_{TM} as follows:

- On input $\langle M\rangle$, where M is a TM:
  1. Running R on input $\langle M, M_\emptyset\rangle$, where $M_\emptyset$ is a TM such that
  $$L(M_\emptyset) = \emptyset.$$
  2. If R accepts, then accept. If R rejects, then reject.

This construction is valid because

$$L(M) = \emptyset \Leftrightarrow L(M) = L(M_\emptyset).$$

Thus, if we had a decider for EQ_{TM}, we could construct a decider for E_{TM}. But we know that E_{TM} is undecidable, so our assumption that EQ_{TM} is decidable must be false. Therefore, EQ_{TM} is undecidable

5.2 Computation Histories

Definition 5.2.1. M is a TM and w is an input string. An accepting **computation history** of M on w is

$$C_1, C_2, \ldots, C_l$$

where

- C_1 is the start configuration of M on input w,
- C_l is an accepting configuration, and
- for each i, C_i legally yields C_{i+1}.

Note. A rejection computation history is defined similarly, except that C_l is a rejecting configuration. If M loops on input w, then there is no computation history of M on w.

Remark. Deterministic TM has at most one (maybe reject or loop) computation history on input w, while a nondeterministic TM may have many computation histories on input w.

5.2.1 Linear Bounded Automata (LBA)

Definition 5.2.2 (LBA). A **linear bounded automaton** (LBA) is a TM with a tape of limited length. Specifically, for an input string of length n, the tape head is not allowed to move beyond the first n cells on the tape. If the head tries to move right at the end of input, the head stays

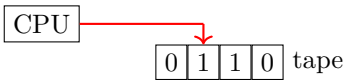


Figure 5.1: Linear Bounded Automaton (LBA)

Note. $A_{\text{DFA}}, A_{\text{CFG}}, E_{\text{DFA}}, E_{\text{CFG}}$ are all LBA-decidable.

Theorem 5.2.1. LBA has a finite number of configurations.

Proof. For an LBA M with

$$\begin{cases} q : \#\text{ states of } M = |Q| \\ g : \#\text{ symbols of } M = |\Gamma| \end{cases}$$

Then M has at most

$$q \cdot g^n \cdot n$$

distinct configurations on an input of length n. Which is because:

- q ways to choose the current state,
- g^n ways to choose the content of the tape (only first n cells matter),
- n ways to choose the position of the tape head.

Thus, the total number of configurations is finite qng^n. ■

5.2.2 A_{LBA} is Decidable

Example. Consider the language

$$A_{\text{LBA}} = \{\langle M, w\rangle \mid M : \text{ LBA that accepts } w\}.$$

We only have to concern about is there loop or not, because if it halts then it either accepts or rejects. From the previous theorem, we know that an LBA has a finite number of configurations. Thus, if an LBA M on input w ever repeats a configuration, then M will loop forever.

5.2.3 E_{LBA} is Undecidable

Example. Consider the language

$$E_{\text{LBA}} = \{\langle M\rangle \mid M : \text{ LBA, } L(M) = \emptyset\}.$$

The question of M accepts w can be solved by checking if $L(B) = \emptyset$, where B is an LBA constructed from M and w. Thus, we assume E_{LBA} is decidable and has a decider R. We can construct a decider S for A_{TM} as follows:

- On input $\langle M, w\rangle$, where M is a TM and w is a string:

- B recognize all accepting computation histories of M on input w.

$$\begin{cases} M \text{ accepts } w & \Rightarrow L(B) \neq \emptyset \\ M \text{ rejects } w & \Rightarrow L(B) = \emptyset \end{cases}$$

- Run R on input $\langle B \rangle$.
- If R accepts, then reject. If R rejects, then accept.

The implementation of B on input x we check if it is an accepting computation history for M on w. To be specifically, we check if x is

$$\#C_1\#C_2\#\cdots\#C_l\# \tag{1}$$

and $C_1 \dots C_l$ satisfies that

- C_1 is the start configuration of M on input w,
- C_l is an accepting configuration, and
- for each i, C_i legally yields C_{i+1}.

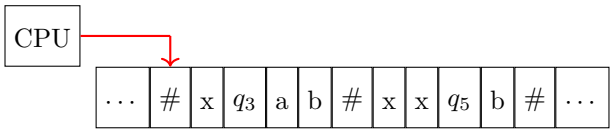


Figure 5.2: Machine B checking computation history

Note. To check the conditions in (1), B can scan through the input multiple times. Each time checking one of the conditions.

1. For the first condition, B checks if C_1 matches $q_0 w$.
2. For the last condition, B checks if C_l contains an q_{accept}.
3. To check the middle condition, B checks each pair C_i and C_{i+1} to see if C_i legally yields C_{i+1}. This can be done by zigzags between C_i and C_{i+1}. If this requires more space than the input length, but it is fine since the extra space for comparision is no more than $|\#C_1 \dots \#C_l\#|$ which is finite.

5.2.4 ALL_{CFG} is Undecidable

As previously seen.

$$E_{\text{CFG}} = \{\langle G \rangle \mid G : \text{ CFG, } L(G) = \emptyset\} \text{ is decidable.}$$

Example. Consider the language

$$ALL_{\text{CFG}} = \{\langle G \rangle \mid G : \text{ CFG, } L(G) = \Sigma^*\}.$$

It is impossible to check if G gernerates all strings. We assume ALL_{CFG} is decidable. We have

$$G \text{ generates } \Sigma^* \Leftrightarrow M \text{ does not accept } w$$

which is equivalent to

$$\begin{cases} G \text{ generates } \Sigma^* & \text{if } M \text{ does not accept } w \\ G \text{ fails some strings} & \text{if } M \text{ accepts } w \end{cases}$$

If we have a decider on G, then we can have a decider for A_{TM}. Getting a contradiction. If M accepts w, we let G fail to generate an accepting computation history of M on w. i.e. G generates all strings

1. Not starting with C_1, or
2. Not ending with an accepting configuration, or
3. C_i does not legally yield C_{i+1} for some i.

To construct such a CFG G, we can construct a PDA to nondeterministically checks three branches for the three requirements. The hardest one is the third branch, which can be done by pushing C_i into the stack and popping from the stack to compare with C_{i+1}. To fix the order issue, we can push(pop) in this order

$$\#\underbrace{\rightarrow}_{C_1}\#\underbrace{\leftarrow}_{C_2^R}\#\underbrace{\rightarrow}_{C_3}\#\underbrace{\leftarrow}_{C_4^R}\#\cdots\#\underbrace{\quad}_{C_l}\#$$

Chapter 6

Complexity Theory

Lecture 12

2025-12-8

6.1 Big-O Notation

> **As previously seen.** We have dicussed the concept of solvable
>
> $$\text{Decidable} \Rightarrow \text{Computationally Solvable}$$

However, not all algorithms are created equal. Some algorithms are more efficient than others. To analyze the efficiency of algorithms, we use **Big-O Notation**.

6.1.1 Analysis of Algorithms

- **Worst Case**: The maximum number of steps taken by an algorithm for any input of size n.
- **Average Case**: The expected number of steps taken by an algorithm for a random input of size n.

Usually, we focus on the worst-case analysis to ensure that our algorithm performs.

Definition 6.1.1. We use a function

$$f : \mathbb{N} \to \mathbb{R}^+$$

to represent the number of steps.

- n: length of input
- $f(n)$: number of steps

Then we give the definition of Big-O notation.

Definition 6.1.2 (Big-O Notation)**.** We say

$$f(n) = O(g(n))$$

if

$$\exists c > 0, n_0 \in \mathbb{N} \text{ such that } \forall n \geq n_0, f(n) \leq c \cdot g(n)$$

Consider the following example.

Example. $f(n) = 6n^3 + 5$

We have

$$6n^3 + 5 \leq 7n^3 \text{ for } n \geq 2$$

That is, we can choose $c = 7$ and $n_0 = 2$. Thus,

$$f(n) = O(n^3)$$

Additionally, we can also say $f(n) = O(n^4)$ as

$$6n^3 + 5 \leq 7n^4 \text{ for } n \geq 2$$

Example. $f(n) = 3n \log_2 n + 5n \log_2 \log_2 n$

We can prove that

$$f(n) = O(n \log n)$$

Note. Note that the base of the logarithm does not matter in Big-O notation since

$$\log_a n = \frac{\log_b n}{\log_b a} = c \cdot \log_b n = O(\log_b n)$$

From

$$n \leq 2^n, \ \forall n \geq 1$$

we have

$$\log_2 n \leq n$$

From this, we can deduce that

$$\log_2 \log_2 n \leq \log_2 n$$

Therefore,

$$f(n) \leq 3n \log_2 n + 5n \log_2 n = 8n \log_2 n, \ \forall n \geq 1$$

Lemma 6.1.1.

$$O(n) + O(n^2) = O(n^2)$$

Proof. Formally,

$$f(n) = O(n), \ g(n) = O(n^2) \Rightarrow f(n) + g(n) = O(n^2)$$

By definition,

$$\begin{cases} \exists c_1, n_1, \ \forall n \geq n_1, f(n) \leq c_1 n \\ \exists c_2, n_2, \ \forall n \geq n_2, g(n) \leq c_2 n^2 \end{cases}$$

Then,

$$f(n) + g(n) \leq c_1 n + c_2 n^2 \leq (c_1 + c_2) n^2, \ \forall n \geq \max(n_1, n_2)$$

Thus, we choose $c = c_1 + c_2$ and $n_0 = \max(n_1, n_2)$ ■

Lemma 6.1.2 (Exponential Function).

$$f(n) = 2^{O(n)}$$

if $\exists\ c, n_0$ such that

$$f(n) \leq 2^{c \cdot n},\ \forall n \geq n_0$$

Lemma 6.1.3 (Constant Function).

$$f(n) = O(1)$$

if $\exists\ c, n_0$ such that

$$f(n) \leq c \cdot 1,\ \forall n \geq n_0$$

Thus,

$$f(n) \leq \max\{f(1), \ldots, f(n_0 - 1), c\},\ \forall n$$

i.e.

$$f(n) \text{ is bounded by a constant for all } n$$

6.1.2 small o Notation

Definition 6.1.3 (small o Notation). We say

$$f(n) = o(g(n))$$

if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Note. Follow the definition of limit, if

$$\lim_{n \to \infty} f(n) = L$$

then for any $\epsilon > 0$, $\exists \delta$ such that

$$\forall n > \delta, |f(n) - L| < \epsilon$$

Then, for small o notation, we have

$$\forall c > 0, \exists n_0 \text{ such that } \forall n \geq n_0,\ \frac{f(n)}{g(n)} \leq (<) c$$

Remark. O versus o:

$$\begin{cases} f(n) = O(g(n)) & \text{if } \exists c > 0, n_0 \text{ such that } \forall n \geq n_0, f(n) \leq c \cdot g(n) \\ f(n) = o(g(n)) & \text{if } \forall c > 0, \exists n_0 \text{ such that } \forall n \geq n_0, f(n) < c \cdot g(n) \end{cases}$$

Example. Consider

$$A = \{0^k 1^k \mid k \geq 0\}$$

What is the # steps taken by a one-tape Turing machine to process a string?

We can separate the process into these steps:

1° We have to check if input is

$$0 \ldots 01 \ldots 1$$

which takes $O(n)$ steps.

2° Then, move back, which takes $O(n)$ steps.

3° Next, we cross off one 0 and one 1, which takes $O(n)$ steps.

4° We repeat steps 2 and 3 until all 0s and 1s are crossed off, which has $n/2$ iterations.

Thus, the total number of steps is

$$O(n) + \frac{n}{2} \cdot O(n) + O(n) = O(n^2)$$

Definition 6.1.4 (Time Complexity Class).

$$\text{TIME}(t(n)) \equiv \{L \mid \text{a language decided by an } O(t(n)) \text{ TM}\}$$

Now we have

$$A = \{0^k 1^k \mid k \geq 0\} \in \text{TIME}(n^2)$$

But we can do better: We first cross off every other 0 and then cross off every other 1. This way, we can do

$$\underline{0}0\underline{0}0\underline{1}1\underline{1}1\underline{1}$$
$$\underline{0}0\underline{1}1$$
$$\underline{01}$$
$$\epsilon$$

The key is the length of the string left must be always even.

Algorithm 6.1: Decide Language $A = \{0^k 1^k \mid k \geq 0\}$

Input: String w
Output: Accept or Reject

```
if format is not 0*1* then
    return Reject;
while tape contains any 0s or 1s do
    Scan tape to count total number of active 0s and 1s;
    if total count is Odd then
        return Reject;
    for each type x ∈ {0, 1} do
        Keep the 1st x, cross off the 2nd x, keep the 3rd x...
return Accept;
```

The whole process takes

$$1 + \log_2 n$$

iterations, and each iteration takes $O(n)$ steps. Thus, the total number of steps is

$$O(n \log n)$$

Thus, we have

$$A \in \text{TIME}(n \log n)$$

We can't do any better than this since

Theorem 6.1.1. Any language decided in $o(n \log n)$ time by a one-tape Turing machine is regular.

But we know that A is not regular.

Remark. If we want to use the method of copying, the problem is that the copy operation is expensive. It takes $O(n^2)$ times to copy n symbols.

We can also do an $O(n)$ algorithm using a two-tape Turing machine.

1° Check if input is

$$0 \ldots 01 \ldots 1$$

which takes $O(n)$ steps.

2° Copy all 0s to the 2nd tape, which takes $O(n)$ steps.

3° Sequentially match each 1 on the 1st tape with a 0 on the 2nd tape, which takes $O(n)$ steps. (if no 0 left, reject)

4° If all 1s are matched, accept; else, reject.

The total number of steps is

$$O(n) + O(n) + O(n) = O(n)$$

But this requires a two-tape Turing machine.

6.2 Time Complexity

As previously seen. In Ch.3 we have dicussed the concept of various Turing machines, which are all equivalent in Computability Theory.

However, in Complexity Theory, different Turing machines may have different time complexities for the same language.

6.2.1 Multi-tape TM and Time Complexity

Theorem 6.2.1. Let $t(n) \geq n$. For a $t(n)$ time multi-tape Turing machine, there exists an equivalent $O(t(n)^2)$ time single-tape Turing machine.

Proof. Recall the simulation of multi-tape TM by single-tape TM.

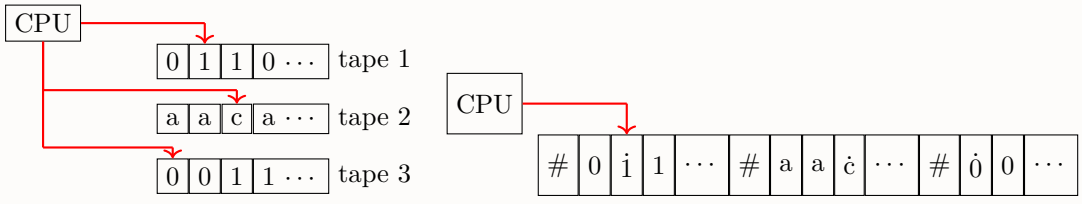


Figure 6.1: Simulation of Multi-tape TM by Single-tape TM

To simulate each step of multi-tape TM, we scan to know where heads point to and do the update. However, we have to right shift the tape. Si we need to know the tape length which is

$$k \times O(t(n)) = O(t(n)) \quad \text{for constant } k$$

A $t(n)$ multi-tape TM generates at most $O(t(n))$ contens in $O(t(n))$ time. Thus, the cost of simu-

lating each step of multi-tape TM is $O(t(n))$. Therefore, the total time is

$$O(t(n)) \times O(t(n)) = O(t(n)^2)$$

■

Definition 6.2.1. NTM Time Complexity $t(n)$ is the maximum # steps the machine uses for any path from root to leaf in the computation tree for any input of size n.

Theorem 6.2.2. Let $t(n) \geq n$. For a $t(n)$ single-tape NTM, there exists an equivalent $2^{O(t(n))}$ time single-tape TM.

Proof. Assume b is the maximal number of branches at each node. Recall the way to simulate NTM by multi-tape TM.

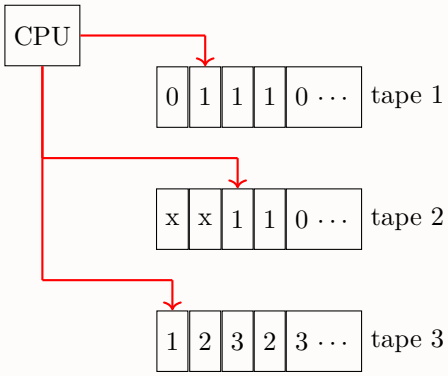


Figure 6.2: Simulation of NTM by Multi-tape TM

We use BFS to do the simulation in the computation tree. The total nodes in the computation tree, which can be found in the tape 3 (record the path from root to node) is

$$1 + b + b^2 + \cdots + b^{t(n)} = \sum_{i=0}^{t(n)} b^i = O(b^{t(n)})$$

Cost of running from root to one node in tape 2 is $O(t(n))$, and the cost of updating tape 3 is also $O(t(n))$. Thus, the total cost of simulating each node is

$$\#\text{ nodes} \times \text{ cost per node } = O(b^{t(n)}) \times O(t(n)) = 2^{O(t(n))}$$

Note. that

$$b^{t(n)} \times t(n) = 2^{\log_2(b^{t(n)} t(n))} = 2^{t(n)\log_2 b + \log_2 t(n)} = 2^{O(t(n))}$$

This is by the three-tape TM simulating the NTM. By the previous theorem, we can simulate the three-tape TM by a single-tape TM in

$$(2^{O(t(n))})^2 = 2^{O(t(n))}$$

■

6.3 Languages in P

Definition 6.3.1 (Class P).

$$\mathrm{P} \equiv \bigcup_k \mathrm{TIME}(n^k)$$

i.e. the class of languages decidable by a polynomial-time deterministic Turing machine.

Note. P class is roughly the class of solvable problems in computer.

6.3.1 PATH Problem

Example.

$$\mathrm{PATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph s.t. } \exists \text{ path from } s \text{ to } t\}$$

We can prove that

$$\mathrm{PATH} \in \mathrm{P}$$

Intuition. Let's start with a brute force way

1° $m : |V(G)|$

2° $|\text{path}| \leq m$ (since no loop)

3° $\#\text{paths} \leq m^m$

4° sequentially check if on has s to t

Total time:

$$O(m^m \cdot m) = O(m^{m+1})$$

which is exponential time.

For an input $\langle G, s, t\rangle$, we can use the following algorithm including $V(G), E(G)$

1° Mark s

2° Repeat until no new node is marked:

- For each edge $(u, v) \in E(G)$, if u is marked, mark v

3° If t is marked, accept; else, reject.

steps in the main loop is at most $|V|$ (if no newly marked node, we stop). at each iteration, we have to scan $|E| \leq |V|^2$. The cost to mark node is polynomial. Thus, the total time is

$$O(|V|) \times O(|E|) = O(|V|^3)$$

Therefore,

$$\mathrm{PATH} \in \mathrm{P}$$

6.3.2 Relatively Prime Problem

Definition 6.3.2 (Relatively Prime). Let $x, y \in \mathbb{Z}$. We say x and y are **relatively prime** if their greatest common divisor (gcd) is 1, i.e.,

$$\gcd(x, y) = 1$$

Example.

$$\text{RELPRIME} = \{\langle x, y\rangle \mid x, y \in \mathbb{N} \text{ are relatively prime}\}$$

From the definition, we have to find a way to compute $\gcd(x, y)$ efficiently. We can use the Euclidean Algorithm.

Algorithm 6.2: Euclidean Algorithm

Input : $\langle x, y\rangle$
Output: Greatest Common Divisor of x and y

```
while y ≠ 0 do
    x ← x mod y;
    exchange x and y;
return x;
```

Note. If $x < y$, then in the first iteration, we have

$$x \leftarrow x \bmod y = x$$

and then exchange x and y. Thus, after the first iteration, we have $x \geq y$.

At each iteration, x or y is reduced by at least half.

- If $x > y$

$$x \bmod y \leq \frac{x}{2}$$

  - If $x < 2y$, then

$$x \bmod y = x - y < x - \frac{x}{2} = \frac{x}{2}$$

  - If $x \geq 2y$, then

$$x \bmod y \leq y \leq \frac{x}{2}$$

Therefore,

$$\#\text{ iterations} \leq 2\max(\log_2 x, \log_2 y) = O(n)$$

where n is the length of the input (x, y are stored as bit string), $\log_2 x + \log_2 y = O(n)$.
In each iteration, $x \bmod y$ is polynomial time computable. Exhanging x and y is also polynomial time. Thus, the total time is

$$O(n) \times \text{poly}(n) = \text{poly}(n)$$

Theorem 6.3.1.

$$\text{Context Free Language} \subseteq \text{P}$$

Proof. Recall the CYK algorithm for CFLs in CNF. For an input string of length n, the total time is

$$O(n^3)$$

■

6.4 Languages in NP

6.4.1 Hamiltonian Path Problem

For some problems, it is difficult to find an algorithm in P. Consider the following example.

Definition 6.4.1 (Hamiltonian Path). A **Hamiltonian Path** in a directed graph is a path that visits all vertex exactly once.

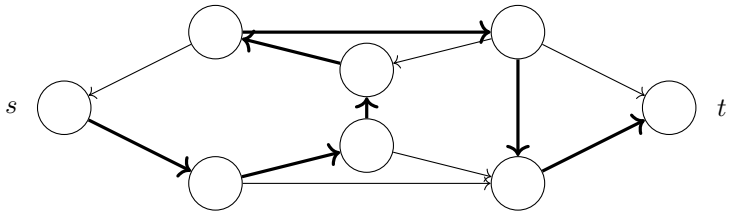


Figure 6.3: Hamiltonian Path Example

Example.

$$\text{HAMPATH} = \{\langle G, s, t\rangle \mid G : \text{a directed graph with a Hamiltonian path from } s \text{ to } t\}$$

A brute-force way: checking all possible paths, but the number is exponential. So we can do a **polynomial-time verification** instead.

$$\text{for a path, in P time} \Rightarrow \text{a Hamiltonian path or not}$$

6.4.2 Compositeness Problem

Definition 6.4.2 (Compositeness). We say $x \in \mathbb{N}$ is **composite** if $\exists\ p, q \in \mathbb{N}$ such that

$$x = p \cdot q, \quad 1 < p, q < x$$

Given x, it is difficult to find such p, q efficiently. However, if we are given p, q, we can verify it in polynomial time by multiplication.

However, there are still some problems that are difficult to verify in polynomial time, such as the **Graph Isomorphism Problem**, or the complement of Hamiltonian path problem

$$\overline{\text{HAMPATH}} = \{\langle G, s, t\rangle \mid G : \text{a directed graph without a Hamiltonian path from } s \text{ to } t\}$$

Verification is difficult since we have to check all possible paths.

6.4.3 Verifier

Definition 6.4.3 (Verifier). An algorithm V is called a **verifier** for a language L if

$$L = \{w \mid \exists c \text{ (certificate) such that } V \text{ accepts } \langle w, c\rangle\}$$

Example. Compositeness problem: V accepts

$$\langle w, c\rangle = \langle x, p\rangle, \text{ where } p \text{ is a factor of } x$$

Example. HAMPATH problem: V accepts

$$\langle w, c\rangle = \langle\langle G, s, t\rangle, \text{ path from } s \text{ to } t\rangle$$

c is called a **certificate** or **witness** that helps to verify $w \in L$.

Definition 6.4.4 (Polynomial-time Verifier)**.** A verifier V is called a **polynomial-time verifier** if V runs in polynomial time with respect to $|w|$.

Remark. Note that the running time of V is with respect to $|w|$, not $|c|$. For a polynomial-time verifier, we have

$$|c| \in \text{poly}(|w|)$$

otherwise, V reading c alone would take super-polynomial time.

6.4.4 Class NP

Definition 6.4.5 (Class NP)**.**

$$\text{NP} \equiv \{L \mid L \text{ has a polynomial-time verifier}\}$$

Another equivalent definition of NP is as follows.

Definition 6.4.6 (NTIME class)**.**

$$\text{NTIME}(t(n)) = \{L \mid L \text{ is decidable by a } O(t(n)) \text{ nondeterministic TM}\}$$

Theorem 6.4.1.

$$\text{NP} = \bigcup_k \text{NTIME}(n^k)$$

For the NTM for language HAMPATH, we do

1° Nondeterministically get a path from s to t in the list $p_1 \cdots p_m$

2° For each list:

- Check for repetitions.
- Check if each edge (p_i, p_{i+1}) exists in G.
- Check if $s = p_1$ and $t = p_m$

Cost on each list is polynomial. The repetitions cost $O(m^2)$, checking edges cost $O(m^2)$, checking s and t cost $O(m)$. Thus, the total time is

$$O(m^2) + O(m^2) + O(m) = O(m^2)$$

Therefore,

$$\text{HAMPATH} \in \text{NTIME}(n^2) \Rightarrow \text{HAMPATH} \in \text{NP}$$

6.4.5 NP ≡ Polynomial-time NTM

Theorem 6.4.2.

$$\bigcup_k \text{NTIME}(n^k) = \{L \mid L \text{ decide by a polynomial-time NTM}\}$$

Proof. We start from the definition:

> **Idea.** We consider both directions.
>
> "⇒" NTM by guessing certificate.
>
> "⇐" using NTM 's accepting branch as certificate.

"⇒" Recall the definition of NP.

$$L = \{w \mid \exists c \text{ (certificate) such that } V \text{ accepts } \langle w, c\rangle\}$$

We have

$$|c| \in \text{poly}(|w|) \Rightarrow |c| \leq |w|^k$$

because to handle $\langle w, c\rangle$ in $|w|^k$, $|c|$ should be bounded by polynomial of $|w|$.

Then we use NTM to

1° Nondeterministically guess c with length $\leq |w|^k$.

2° Simulate V on input $\langle w, c\rangle$.

That is, we run all c in parallel and each is polynomial time. We have that for any $w \in L$, the NTM accepts in polynomial time. Thus,

$$L \in \text{NTIME}(n^k)$$

"⇐" Assume

$$L \in \text{NTIME}(n^k)$$

i.e. w is accepted by a polynomial NTM. We let c be any accepting branch where each branch is polynomial time. Then we run the verifier V that handles $\langle w, c\rangle$ in polynomial time. Thus,

$$L \text{ has a polynomial-time verifier}$$

Proof complete. ■

6.4.6 SUBSET-SUM Problem

Example.

$$\text{SUBSET-SUM} = \left\{ \langle S, t\rangle \;\middle|\; \exists\, S' \subseteq S \text{ such that } \sum_{x \in S'} x = t \right\}$$

Note that we allow repetition of elements in S. We will prove that

$$\text{SUBSET-SUM} \in \text{NP}$$

Consider any input $\langle\langle S, t\rangle, c\rangle$. We

1° Check if $\sum c_i = t$

2° Check if $c_i \in S$

3° If both hold, accept; else, reject.

The cost of summation is $O(|S|)$, and checking $c_i \in S$ also takes $O(|S|)$. Thus, the total time is

$$O(|S|) + O(|S|) = O(|S|)$$

Therefore,

$$\text{SUBSET-SUM} \in \text{NP}$$

6.4.7 P versus NP and NP-completeness

Roughly

- P: problems that can be **solved** in polynomial time.
- NP: problems that can be **verified** in polynomial time.

The greatest open question in computer science is whether

$$\text{P} = \text{NP} \ ?$$

It has shown that

$$\text{P} \subseteq \text{NP}$$

. However, it is not known whether

$$\text{P} = \text{NP} \text{ or } \text{P} \neq \text{NP}$$

For certain problems in NP, if we can find a polynomial time algorithm to solve one of them,

$$\text{P} = \text{NP}$$

These problems are called **NP-complete** problems. We will discuss NP-completeness in the next lecture.