Formal Languages

Handout

Instructor: Ying-ping Chen

Semester: 2019 Fall

Contents

Course Information

Instructor

- Ying-ping Chen
  - Contact Information
    - Office Hour: 1E (Monday 1:20pm–2:10pm) (By Appointment Only)
    - Office Location: EC 711
    - Office Phone: 31446
    - Email: ypchen@cs.nctu.edu.tw
    - Website: http://www.cs.nctu.edu.tw/~ypchen/
  - Natural Computing Laboratory
    - Location: EC 018
    - Website: http://nclab.tw/
  - Research Fields
    - Evolutionary Computation

Time & Place

- 1G (Monday 3:30pm–4:20pm); EC 122
- 4CD (Thursday 10:10am–12:00pm); EC 122

Books & References

- Required textbook

  1. *Introduction to the Theory of Computation (3rd Edition, International Edition)*, Michael Sipser, Thomson Course Technology. ISBN: 0619217642 (Hardback) / ISBN: 1133187811 (Softback).

- References

  1. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*, John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman, Addison-Wesley. ISBN: 0321476174 (Softback).
  2. *Introduction to Languages and the Theory of Computation (3rd Edition)*, John C. Martin, McGraw-Hill. ISBN: 0071240187 (Softback).
  3. *An Introduction to Formal Languages and Automata (4th Edition)*, Peter Linz, Jones and Bartlett Publishers. ISBN: 0763737984 (Hardback).
  4. *Problem Solving in Automata, Languages, and Complexity*, Ding-Zhu Du, and Ker-I Ko, Wiley-Interscience, Sep. 15, 2001. ISBN: 0471439606 (Hardback), 0471224642 (Electronic). [Downloadable in the NCTU campus].

Rules, Rules, and Rules

- **No cellular phone** (except mine) can ring in class
- Speak whatever you think in class

Grading Policy

- Mid-term examination #1: 30%
- Mid-term examination #2: 30%
- Final examination: 40%
- **Total: 100%**

Important Websites & Useful Links

- NCTU New E3
  - URL: http://e3new.nctu.edu.tw
  - Using the system is *mandatory*
  - Everything is and will be on the class page
  - Handout, class announcements, supporting materials, discussion boards, etc.

Current Events

- Week 8: Oct. 31, 2019;
  Mid-term exam #1
- Week 12: Nov. 28, 2019;
  Mid-term exam #2
- Week 17: Jan. 2, 2020;
  Final exam

Chapter 0

Introduction

0.1 The BIG question

What are the fundamental capabilities and limitations of computers?

The question is interpreted differently in

- automata;
- computability;
- complexity.

0.1.1 Complexity Theory

What makes some problems computationally hard and others easy?

- Sorting vs. scheduling
- Classification based on the computational difficulty
- When facing a difficult problem,
  - Finding the root of the difficulty—altering it;
  - Accepting less than a perfect solution;
  - Hard only in the worst case—going for usual cases;
  - Considering alternative types of computation.

0.1.2 Computability Theory

Some basic problems cannot be solved by computers. Solvable vs. unsolvable.

0.1.3 Automata Theory

Properties of mathematical models of computation.

- Finite automata
  - Text processing, compilers, and hardware design.
- Context-free grammar
  - Programming languages and artificial intelligence.

0.2 Notions and Terminology

0.2.1 Sets

A *set* is a group of

- *elements* or
- *members*

represented as a unit, such as $\{7, 21, 57\}$. Relations between members and sets:

- Membership: $7 \in \{7, 21, 57\}$;
- Nonmembership: $8 \notin \{7, 21, 57\}$.

Relations between sets:

- Subset: $A \subseteq B$;
- Proper subset: $A \subsetneq B$.

Some sets:

- Multiset: Number of occurrences of members counts.
- Infinite set: Infinitely many elements (...).
- Natural numbers $\mathcal{N}$: $\{1, 2, 3, \ldots\}$.
- Integers $\mathcal{Z}$: $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$.
- Empty set $\emptyset$: 0 members.

Set definitions:

- Listing: $\{7, 21, 57\}$.
- Using rules: $\{n \mid n = m^2 \text{ for some } m \in \mathcal{N}\}$.

Set operations:

- Union: $A \cup B$.
- Intersection: $A \cap B$.
- Complement: $\overline{A}$.

Visualization tool: *Venn Diagram*.

0.2.2 Sequences and Tuples

A *sequence* of objects is a list of these objects in some order. Finite sequences often are called *tuples*.

- k-tuple: A sequence with k elements.
- Pair: $k = 2$.

Sets and sequences may appear as elements:

- Power set of A: The set of all subsets of A ($\mathcal{P}(A)$).
- Cartesian product of A and B: $A \times B$.

0.2.3 Functions and Relations

A *function*, also called *mapping*, is an objects that sets up an input-output relationship, such as

$$f(a) = b \ .$$

For a function,

- Domain: The set of possible inputs;
- Range: The set of the outputs;

$$f : D \to R \ .$$

- k-ary function: if the input is a k-tuple $(a_1, a_2, \ldots, a_k)$.
- Arguments: $a_1, a_2, \ldots, a_k$ in the k-tuple;
- Unary function: $k = 1$;
- Binary function: $k = 2$;
- Infix notation;
- Prefix notation;
- Predicate or property: A function whose range is {TRUE, FALSE};
- Relation: A property whose domain is a set of k-tuples $A \times \cdots \times A$;
- Equivalence relation R:
  1. Reflexive: xRx;
  2. Symmetric: $xRy \Rightarrow yRx$;
  3. Transitive: $xRy \wedge yRz \Rightarrow xRz$.

0.2.4 Graphs

An *undirected graph*, or simply a *graph*, $G = (V, E)$ is a set of points with lines, called *edges*, connecting some of the points, called *nodes* or *vertices*.
For a graph,

- Degree: Number of edges at a particular node;
- Subgraph: Subset of nodes + subset of edges;
- Path: A sequence of nodes connected by edges;
- Simple path: No repetition of nodes;
- Connected: A path exists for every pair of nodes;
- Cycle: A path starts and ends in the same node;
- Simple cycle: ≥ 3 nodes, repeats only first and last;
- Tree: Connected + no simple cycles;
  - Root;
  - Leaves: Nodes of degree 1, except for root.

If the edges are arrows instead of lines, the graph is called *directed graph*. For a directed graph,

- Outdegree: Number of arrows pointing from a node;
- Indegree: Number of arrows pointing to a node;
- Directed path: A path in which all the arrows point in the same direction as its steps;
- Strongly connected: A directed path exists for every pair of nodes.

0.2.5 Strings and Languages

- Alphabet Σ or Γ: A finite nonempty set of symbols.
- String w: A finite sequence of symbols from Σ.
- Length of w ($|w|$): Number of symbols w contains.
- Empty string ε (ϵ, λ, or Λ): Length $= 0$.
- Reverse of w ($w^{\mathcal{R}}$): w in the opposite order.
- Substring of w: Strings appear consecutively in w.
- Concatenations of strings:

  xy denotes the concatenation of x and y.

  - $|xy| = |x| + |y|$
  - $\epsilon w = w\epsilon = w$
  - $x^k = \overbrace{xx\cdots x}^{k}$

- Lexicographic ordering: Short first + dictionary.

  $$\varepsilon, \mathtt{0}, \mathtt{1}, \mathtt{00}, \mathtt{01}, \mathtt{10}, \mathtt{11}, \mathtt{000}, \mathtt{001}, \mathtt{010}, \ldots$$

- Language: A set of strings.

0.2.6 Boolean Logic

- {TRUE, FALSE}: Boolean values.
- Boolean operations:
  - NOT, negation, $\neg$;
  - AND, conjunction, $\wedge$;
  - OR, disjunction, $\vee$;

  - XOR, exclusive or, $\oplus$;
  - equality, $\leftrightarrow$;
  - implication, $\rightarrow$.
- Distributive law for AND and OR.
  - $P \wedge (Q \vee R)$ equals $(P \wedge Q) \vee (P \wedge R)$;
  - $P \vee (Q \wedge R)$ equals $(P \vee Q) \wedge (P \vee R)$.

0.3 Definitions, Theorems, and Proofs

We use *definitions* to describe the objects and notions.
We make *mathematical statements* about them.
A *proof* is a convincing logical argument that a statement is true in an absolute sense.
If a mathematical statement is proved true, it is called a *theorem* (or *lemma*, *corollary*).

Finding proofs is not always easy. Reading pp. 17–20 may help you a lot.

0.4 Types of Proofs

0.4.1 Proof by Construction

To prove that a particular type of object exists by demonstrating how to construct the object.

E.g., Theorem 0.22.

p. 21

0.4.2 Proof by Contradiction

To prove a theorem is true by firstly assuming that it is false and obtaining an obviously false consequence.

E.g., Example 0.23, Theorem 0.24.

p. 22

0.4.3 Proof by Induction

To prove that all elements of an infinite set have a specified property by establishing a *basis*, making the *induction hypothesis*, and providing the *induction step*.

p. 24

E.g., Theorem 0.25.

Chapter 1

Regular Languages

What is a computer?

Let's begin with the simplest *computational model*: the *finite state machine* or *finite automaton*.

1.1 Finite Automata

1.1.1 An Example: Automatic Door

- State diagram; Fig. 1.2, p. 32
- State transition table. Fig. 1.3, p. 33

1.1.2 Another Example: M_1

- State diagram; Fig. 1.4, p. 34
- States: q_1, q_2, q_3;
- Start state: q_1;
- Accept state: q_2; string accepting if stop here
- Transitions.

1.1.3 Formal Definition of Finite Automata

Definition 1.5, p. 35

A *finite automaton* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

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

Fig. 1.6, p. 36

We can describe M_1 formally.

p. 36

If A is the set of all strings that machine M accepts, we say that A is the *language of machine* M and write

$$L(M) = A \ .$$

recognize for languages

We say that M *recognizes* A or M *accepts* A. A machine always recognizes only **one** language.

1.1.4 Formal Definition of Computation

Finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ *accepts* string $w = w_1 w_2 \cdots w_n$ if a sequence of states $r_0, r_1, \cdots, r_n$ in Q exists with three conditions:

1. $r_0 = q_0$;
2. $\delta(r_i, w_{i+1}) = r_{i+1}, \forall i = 0, \ldots, n-1$;
3. $r_n \in F$.

Definition 1.16, p. 40

A language is called a *regular language* if some finite automaton recognizes it.

1.1.5 The Regular Operations

Let A and B be languages, let's define the following three *regular operations*:

Definition 1.23, p. 44

- **Union**:
  $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$;
- **Concatenation**:
  $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$;
- **Star**:
  $A^* = \{x_1 x_2 \cdots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$.
  =A^0 U A^1 U A^2 U...

Theorem 1.25: The class of regular languages is closed under the union operation.

p. 45

Proof idea: Construct a new automaton simulating the two given machines simultaneously.

Theorem 1.26: The class of regular languages is closed under the concatenation operation.

p. 47

Proof idea: Break the string into two pieces such that the first one can be accepted by the first machine, and the other one can be accepted by the second machine. But, **where** to break?

1.2 Nondeterminism

Deterministic vs. nondeterministic

- Deterministic computation: Determined next move.
- Nondeterministic computation: Several choices may exist for the next move.

Fig. 1.27, p. 48

`NFA`: Nondeterministic finite automata

Figs. 1.28 & 1.29, p. 49

Nondeterministic computation—"fork a process" on every possible choice/computational path

1.2.1 Formal Definition of `NFA`s

Definition 1.37, p. 53

A *nondeterministic finite automaton* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

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

Example 1.38, p. 54

We can describe N_1 formally.

Nondeterministic finite automaton $N = (Q, \Sigma, \delta, q_0, F)$ *accepts* string $w = y_1 y_2 \cdots y_m$, where $y_i \in \Sigma_\varepsilon$, if a sequence of states $r_0, r_1, \cdots, r_m$ in Q exists with three conditions:

1. $r_0 = q_0$;
2. $r_{i+1} \in \delta(r_i, y_{i+1}), \forall i = 0, \ldots, m-1$;
3. $r_m \in F$.

Then, what kind of languages do nondeterministic finite automata recognize?

1.2.2 Equivalence of NFAs and DFAs

Theorem 1.39: Every NFA has an equivalent DFA.

p. 55

Proof idea: Convert the NFA N into an equivalent DFA M that simulates the NFA N.

Proof: As stated on pp. 55–56.

Example 1.41, p. 57

Corollary 1.40: A language is regular if and only if some nondeterministic finite automaton recognizes it.

p. 56

Correctness proof for the conversion
(no ε arrows for simplicity)

For DFAs, let's define the *extended* transition function

$$\hat{\delta} : Q \times \Sigma^* \to Q \ .$$

Let $w = xa, w \in \Sigma^*, x \in \Sigma^*, a \in \Sigma$, and $|w| \geq 1$. Define

$$\begin{aligned} \hat{\delta}(q, \varepsilon) &= q \\ \hat{\delta}(q, w) &= \delta(\hat{\delta}(q, x), a) \ . \end{aligned}$$

For NFAs, let's define the *extended* transition function

$$\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q) \ .$$

Let $w = xa, w \in \Sigma^*, x \in \Sigma^*, a \in \Sigma$, and $|w| \geq 1$. Define

$$\begin{aligned} \hat{\delta}(q, \varepsilon) &= \{q\} \\ \hat{\delta}(q, xa) &= \textstyle\bigcup_{p \in \hat{\delta}(q,x)} \delta(p, a) \ . \end{aligned}$$

In other words, suppose

$$\hat{\delta}(q, x) = \{p_1, p_2, \ldots, p_k\} \ ,$$

then

$$\hat{\delta}(q, w) = \hat{\delta}(q, xa) = \bigcup_{i=1}^{k} \delta(p_i, a) \ .$$

Claim: For every $w \in \Sigma^*$, $\hat{\delta}_M(q_0{}', w) = \hat{\delta}_N(q_0, w)$.
Proof: Induct on $|w|$.
Basis: $|w| = 0$,

$$\hat{\delta}_M(q_0{}', \varepsilon) = \{q_0\} = \hat{\delta}_N(q_0, \varepsilon) \ .$$

Induction step: Assume that for $|x| = n, x \in \Sigma^*$,

$$\hat{\delta}_M(q_0{}', x) = \hat{\delta}_N(q_0, x) \ .$$

Then, when $|w| = n + 1, w = xa$, we must show that

$$\hat{\delta}_M(q_0{}', xa) = \hat{\delta}_N(q_0, xa) \ .$$

$$\begin{array}{rcll} \hat{\delta}_M(q_0{}', xa) & = & \delta_M(\hat{\delta}_M(q_0{}', x), a) & \texttt{DFA } \hat{\delta} \\ & = & \delta_M(\hat{\delta}_N(q_0, x), a) & \text{Hypothesis} \\ & = & \bigcup_{p \in \hat{\delta}_N(q_0, x)} \delta_N(p, a) & M\text{'s } \delta \\ & = & \hat{\delta}_N(q_0, xa) & \texttt{NFA } \hat{\delta} \end{array}$$

□

Claim: $L(M) = L(N)$.
Proof:

$$L(M) = \{w \mid \hat{\delta}_M(q_0{}', w) \in F'\} \ ,$$

where F' is the set of subsets S of Q such that $S \cap F \neq \emptyset$.

$$\begin{array}{rl} L(N) & = \{w \mid \hat{\delta}_N(q_0, w) \cap F \neq \emptyset\} \\ & = \{w \mid \hat{\delta}_M(q_0{}', w) \cap F \neq \emptyset\} \\ & = \{w \mid \hat{\delta}_M(q_0{}', w) \in F'\} \\ & = L(M) \end{array}$$

□

Try to prove the case with ε arrows on your own.

1.2.3 Closure under the Regular Operations

Theorem 1.45: The class of regular languages is closed under the union operation.

p. 59

Proof idea: Construct an `NFA` to recognize the two regular languages.

Remember Corollary 1.40, p. 56

Proof: As stated on p. 60.

Fig. 1.46, p. 59

Theorem 1.47: The class of regular languages is closed under the concatenation operation.

p. 60

Proof idea: Construct an `NFA` to split the string non-deterministically.

Proof: As stated on p. 61.

Fig. 1.48, p. 61

Theorem 1.49: The class of regular languages is closed under the star operation.

p. 62

Proof idea: Construct an `NFA` to repeatedly accept strings from the same language.

Proof: As stated on pp. 62–63.

Fig. 1.50, p. 62

1.3 Regular Expressions

1.3.1 Formal Definition (Inductive Definition)

Definition 1.52, p. 64

R is a *regular expression* if R is

1. a for some a in the alphabet Σ $[L(a) = \{a\}]$;
2. ε $[L(\varepsilon) = \{\varepsilon\}]$;
3. $\emptyset$ $[L(\emptyset) = \emptyset]$;
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions $[L(R_1 \cup R_2) = L(R_1) \cup L(R_2)]$;
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions $[L(R_1 \circ R_2) = L(R_1) \circ L(R_2)]$;
6. (R_1^*), where R_1 is a regular expression $[L(R_1^*) = (L(R_1))^*]$.

Example 1.53, p. 65

1.3.2 Equivalence with Finite Automata

p. 66

Theorem 1.54: A language is regular if and only if some regular expression describes it.
Proof: Lemma 1.55 + Lemma 1.60 + Claim 1.65.

p. 67

Lemma 1.55: If a language is described by a regular expression, then it is regular.
Proof idea: Use an `NFA` to recognize the language described by a regular expression.

Example 1.56, p. 68

Proof: As stated on pp. 67–68.

p. 69

Lemma 1.60: If a language is regular, then it is described by a regular expression.
Proof idea: Convert `DFA`s into regular expressions.

Example 1.66, pp. 74–75

Proof: As stated on pp. 72–73.

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

Definition 1.64, p. 73

Fig. 1.61, p. 70

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

A `GNFA` accepts a string w in Σ^* if $w = w_1 w_2 \cdots w_k$, where each $w_i \in \Sigma^*$ and a sequenceof states $q_0, q_1, \cdots, q_k$ exists such that

1. $q_0 = q_{\text{start}}$;
2. $q_k = q_{\text{accept}}$;
3. for each i, we have $w_i \in L(R_i)$, where $R_i = \delta(q_{i-1}, q_i)$.

CONVERT(G)

Fig. 1.63, p. 72

1. Let k be the number of states of G.
2. If $k = 2$, then G must consist of a start state, an accept state, and a single arrow connecting them and labeled with a regular expression R. Return the expression R.
3. If $k > 2$, we select any state $q_{\text{rip}} \in Q$ different from q_{start} and q_{accept} and let G' be the `GNFA`

$$(Q', \Sigma, \delta', q_{\text{start}}, q_{\text{accept}}) \ ,$$

where

$$Q' = Q - \{q_{\text{rip}}\} \ ,$$

and for any $q_i \in Q' - \{q_{\text{accept}}\}$ and any $q_j \in Q' - \{q_{\text{start}}\}$ let

$$\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4) \ ,$$

for $R_1 = \delta(q_i, q_{\text{rip}})$, $R_2 = \delta(q_{\text{rip}}, q_{\text{rip}})$, $R_3 = \delta(q_{\text{rip}}, q_j)$, and $R_4 = \delta(q_i, q_j)$.
4. Compute CONVERT(G') and return this value.

p. 74

Claim 1.65: For any `GNFA` G, CONVERT(G) is equivalent to G.

Proof: As stated on p. 74.

1.4 Nonregular Languages

Consider the language

p. 77

$$B = \{ \mathtt{0}^n\mathtt{1}^n \mid n \geq 0 \} .$$

1.4.1 Pumping Lemma

Theorem 1.70—Pumping lemma: If A is a regular language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

p. 78

1. for each $i \geq 0, xy^iz \in A$;
2. $|y| > 0$;
3. $|xy| \leq p$.

Proof idea: When the string is long enough, some state(s) must be repeated during the computation.

Fig. 1.71, p. 78

Proof: As stated on p. 79.

How to use the pumping lemma:

Example 1.73, p. 80

1. Assume the language, say, B, is regular in order to obtain a contradiction.
2. By the pumping lemma, a pumping length p exists, and any string $w \in B$ can be pumped if $|w| \geq p$.
3. Find a string $s \in B$, $|s| \geq p$, that s cannot be pumped as described in the pumping lemma.
4. The contradiction is obtained, and therefore, B is proved to be nonregular.

1.5 Closure Properties

Let A and B be regular languages. The results of the following operations are all regular languages:

- Union: $A \cup B$;
- Concatenation: AB;
- Star: A^*;
- Intersection: $A \cap B$;
- Complement: $\overline{A}$ (i.e., $\Sigma^* - A$);
- Difference: $A - B$ ($A - B = A \cap \overline{B}$);
- Reversal: A^R;
- Homomorphism: $h(A)$;
- Inverse homomorphism: $h^{-1}(A)$.

Homomorphism: A string substitution such that each symbol is replaced by a single string. That is,

$$h : \Sigma \to \Pi^* \ .$$

For example, for an alphabet $\Sigma = \{ \mathtt{a}, \mathtt{b}, \mathtt{c} \}$, one possibility might be

$$\begin{aligned} \Pi &= \{ \mathtt{0}, \mathtt{1} \} \\ h(\mathtt{a}) &= \mathtt{11} \\ h(\mathtt{b}) &= \mathtt{010} \\ h(\mathtt{c}) &= \varepsilon \end{aligned}$$

$$\begin{aligned} \mathtt{abba} &\to \mathtt{1101001011} \\ \mathtt{abbcacc} &\to \mathtt{1101001011} \end{aligned}$$

Inverse homomorphism:

$$h^{-1}(A \subseteq \Pi^*) = \{ \, w \mid w \in \Sigma^*, h(w) \in A \, \} \ .$$

1.6 Myhill-Nerode Theorem

Problem 1.47, p. 90

Relation defined by a given language: Let x and y be strings and L be any language. We say that x and y are ***distinguishable by L*** if some string z exists whereby exactly one of the strings xz and yz is a member of L; otherwise, for every string z, we have $xz \in L$ whenever $yz \in L$ and we say that x and y are ***indistinguishable by L***. If x and y are indistinguishable by L we write $x \equiv_L y$. In other words, for $x, y \in \Sigma^*$,

$$x \equiv_L y \text{ iff for each } z \in \Sigma^* : xz \in L \Leftrightarrow yz \in L \ .$$

$\equiv_L$ is an equivlance relation because obviously

1. Reflexive: $x \equiv_L x$;
2. Symmetric: $x \equiv_L y \Rightarrow y \equiv_L x$;
3. Transitive: $x \equiv_L y \wedge y \equiv_L z \Rightarrow x \equiv_L z$.

For example, $\Sigma = \{ \mathtt{0}, \mathtt{1} \}$. Let $L_\mathtt{0} = \{ \mathtt{0}^n \mid n > 0 \}$, and define $\equiv_{L_\mathtt{0}}$ by using $L_\mathtt{0}$. We have

- $\mathtt{0} \equiv_{L_\mathtt{0}} \mathtt{000}$
- $\mathtt{11} \equiv_{L_\mathtt{0}} \mathtt{101}$
- $\varepsilon \not\equiv_{L_\mathtt{0}} \mathtt{00}$

Problem 1.48, p. 90

Myhill-Nerode theorem: Let L be a language and let X be a set of strings. Say that X is ***pairwise distinguishable by L*** if every two distinct strings in X are distinguishable by L. Define the ***index of L*** to be the maximum number of elements in any set that is pairwise distinguishable by L. The index of L may be finite or infinite.

1. If L is recognized by a `DFA` with k states, L has index at most k.
2. If the index of L is a finite number k, it is recognized by a `DFA` with k states.
3. L is regular iff it has finite index. Moreover, its index is the size of the smallest `DFA` recognizing it.

Proof: As stated on pp. 97–98.

Example 1.73, p. 80

How to use Myhill-Nerode theorem:

1. Given a langauge B, construct the equivalence relation $\equiv_B$ by using B.
2. Prove that B has finite or infinite index:
   - Finite: B is a regular language.
   - Infinite: B is not a regular language.

1.7 Minimization of DFAs

Equivalent states: State p and state q are said to be *equivalent* if for all $w \in \Sigma^*$, $\hat{\delta}(p, w) \in F \Leftrightarrow \hat{\delta}(q, w) \in F$; and *distinguishable* otherwise.

1.7.1 Table-filling Algorithm

HMU 3rd, p. 157

Given DFA $A = (Q, \Sigma, \delta, q_0, F)$, the following algorithm finds all distinguishable pairs in A.
Basis: If $p \in F$ and $q \notin F$, $\{ p, q \}$ is distinguishable.
Induction step: If $\{ r, s \}$ is distinguishable, and

$$r = \delta(p, a)$$
$$s = \delta(q, a)$$

for some $a \in \Sigma$, then $\{ p, q \}$ is distinguishable.

Why? Let w be the string that distinguishes $\{ r, s \}$.

$$\hat{\delta}(p, aw) = \hat{\delta}(r, w)$$
$$\hat{\delta}(q, aw) = \hat{\delta}(s, w)$$

Thus, aw distinguishes $\{ p, q \}$. □

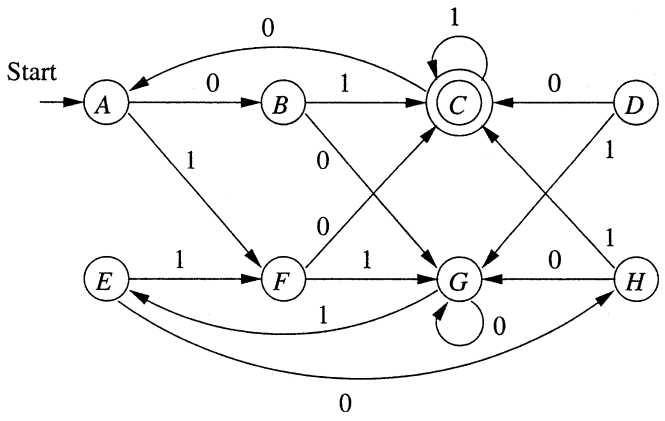
Equivalent states in a given DFA can be found by the table-filling algorithm, and therefore, the DFA can be minimized by combining each set of equivalent states into one single state. See problem 7.40 for the detailed algorithm.

Problem 7.25, pp. 324–325

1.7.2 An Example for Minimizing DFAs

HMU 3rd, Fig. 4.8, p. 156

Given the DFA:



HMU 3rd, Fig. 4.9, p. 157

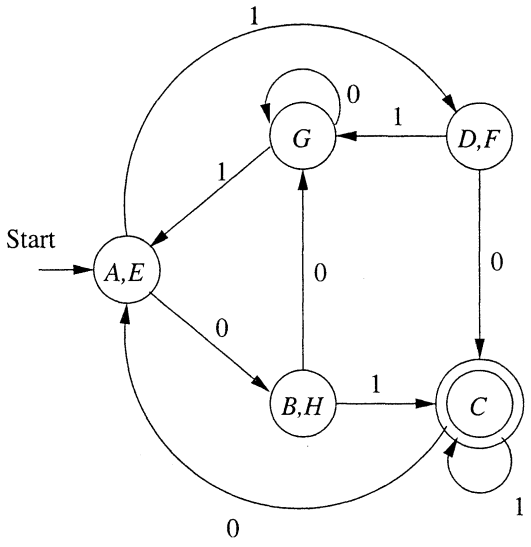
Find all the distinguishable pairs of states:

	A	B	C	D	E	F	G
B	x						
C	x	x					
D	x	x	x				
E		x	x	x			
F	x	x	x		x		
G	x	x	x	x	x	x	
H	x		x	x	x	x	x

Equivalent state pairs: $\{ A, E \}$, $\{ B, H \}$, and $\{ D, F \}$.

HMU 3rd, Fig. 4.12, p. 163

Hence, the result is



Chapter 2

Context-Free Languages

2.1 Context-Free Grammars (CFGs)

2.1.1 Formal Definition of CFGs

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

Definition 2.2, p. 104

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

Each rule consists of

- *head*, a variable in V ,
- symbol $\rightarrow$, and
- a string in $(V \cup \Sigma)^*$, called the *body*.

If $u, v, w \in (V \cup \Sigma)^*$, and $A \to w$ is a rule of the grammar, we say that uAv ***yields*** uwv, written $uAv \Rightarrow uwv$. Say that u ***derives*** v, written $u \overset{*}{\Rightarrow} v$, if $u = v$ or if a sequence $u_1, u_2, \cdots, u_k$ exists for $k \geq 0$ and

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow v \ .$$

Example 2.3, p. 105

The ***language of the grammar*** is

$$\left\{ w \in \Sigma^* \;\middle|\; S \overset{*}{\Rightarrow} w \right\} \ .$$

2.1.2 Ambiguity

Leftmost derivation: A derivation of a string w in a grammar G is a ***leftmost derivation*** if at every step the leftmost remaining variable is the one replaced.

Definition 2.7, p. 108

Ambiguous: A string w is derived ***ambiguously*** in context-free grammar G if it has two or more different leftmost derivations. Grammar G is ***ambiguous*** if it generates some string ambiguously.

Fig. 2.6, p. 108

Inherently ambiguous: The context-free languages that can be generated only by ambiguous grammars are called ***inherently ambiguous***. For example,

Problem 2.41, p. 158

$$\left\{ \mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \;\middle|\; i = j \text{ or } j = k \right\} \ .$$

2.1.3 Chomsky Normal Form

A context-free grammar is in ***Chomsky normal form*** if every rule is of the form

Definition 2.8, p. 109

$$\begin{aligned} A &\to BC \\ A &\to a \end{aligned}$$

where a is any terminal and A, B, and C are any variables—except that B and C may not be the start variable. In addition we permit the rule $S \to \varepsilon$, where S is the start variable.

Theorem 2.9: Any context-free language is generated by a context-free grammar in Chomsky normal form.

p. 109

Proof idea: Convert any context-free grammar into Chomsky normal form:

1. Add a new start variable;
2. Eliminate all ε rules ($A \to \varepsilon$);
3. Handle all unit rules ($A \to B$);
4. Convert all rules into the proper form.

Proof: As stated on pp. 109–110.

Example 2.10, pp. 110–111

2.2 Pushdown Automata (PDAs)

2.2.1 Formal Definition of PDAs

Definition 2.13, p. 113

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

1. Q is the set of states;
2. Σ is the input alphabet;
3. Γ is the stack alphabet;
4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function;
5. $q_0 \in Q$ is the start state;
6. $F \subseteq Q$ is the set of accept states.

Example 2.14, pp. 114–115

A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ accepts input w if w can be written as $w = w_1 w_2 \cdots w_m$, where $w_i \in \Sigma_\varepsilon$ and sequences of states $r_0, r_1, \cdots, r_m \in Q$ and strings $s_0, s_1, \cdots, s_m \in \Gamma^*$ exist that satisfy the following three conditions. The stings s_i represent the sequences of stack contents that M has on the accepting branch of the computation.

1. $r_0 = q_0$ and $s_0 = \varepsilon$;
2. For $i = 0, \cdots, m-1$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$;
3. $r_m \in F$.

Frequently used mechanisms:

- Empty stack detection: `$`;
- Input end detection: Accept states.

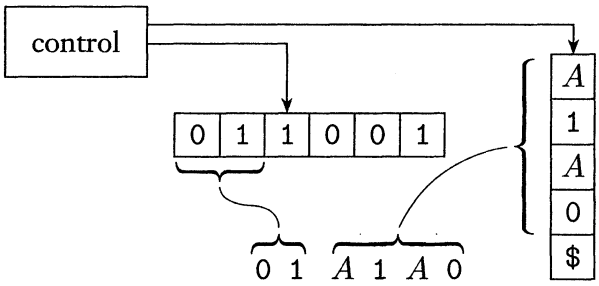
2.2.2 Equivalence with CFGs

Theorem 2.20: A language is context free if and only if some pushdown automaton recognizes it. p. 117

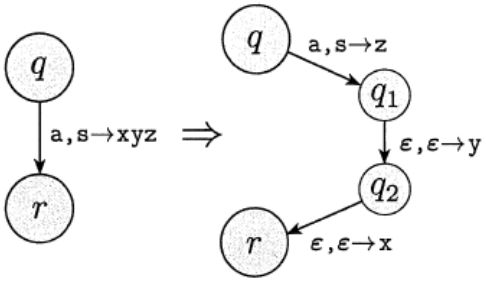
Proof: Lemma 2.21 + Lemma 2.27 + Claim 2.30 + Claim 2.31.

Lemma 2.21: If a language is context free, then some pushdown automaton recognizes it. p. 117

Proof idea: Convert the given CFG into a PDA. Use PDA's nondeterminism to guess the substitution sequence (from the leftmost derivation) and to match the input and the **intermediate strings**, stored in the stack.



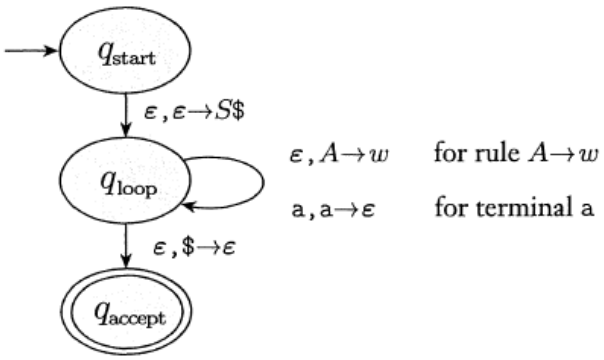
The shorthand for $(r, xyz) \in \delta(q, a, s)$:



For the symbol on the stack top, two kinds of transitions:

- For terminal a, match the input: $\mathtt{a}, \mathtt{a} \to \varepsilon$
- For variable A, apply the rule: $\varepsilon, A \to w$

The state diagram of a PDA obtained by converting a given CFG.



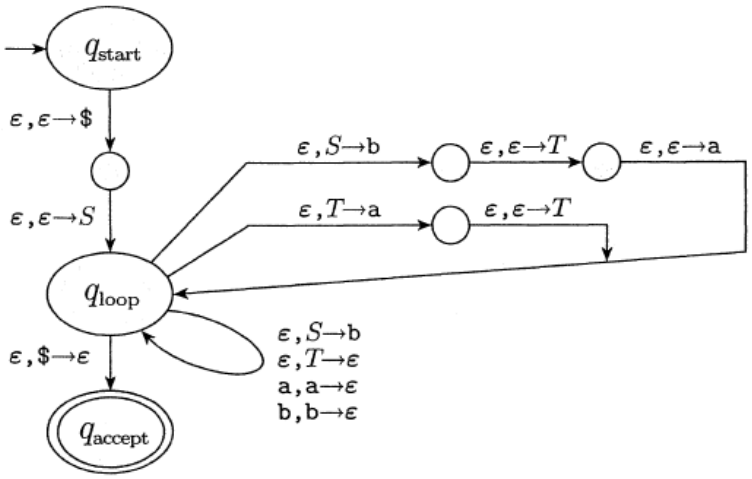
Proof: As stated on pp. 118–120.

p. 120

Example 2.25:
Given CFG $G = (\{S, T\}, \{\mathtt{a}, \mathtt{b}\}, R, S)$, where R contains the following productions:

$$\begin{aligned} S &\to \mathtt{a}T\mathtt{b} \mid \mathtt{b} \\ T &\to T\mathtt{a} \mid \varepsilon \end{aligned}$$

The transition function of the converted PDA is shown in the following diagram:



Lemma 2.27: If a pushdown automaton recognizes some language, then it is context free.

p. 121

Proof idea: Convert the given PDA into an equivalent CFG. Modify the PDA to fit the framework and use one variable for **each pair of states** to generate all the strings that are processed by the PDA for the transitions from the first state to the second state.

Modify the PDA P such that

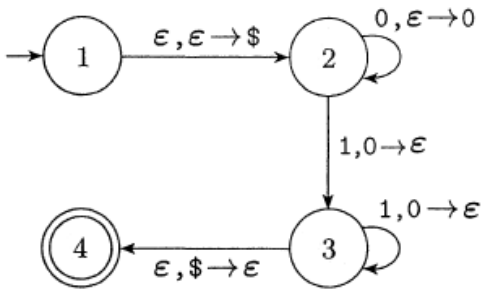
1. It has a single accept state, q_{accept}.
2. It empties its stack before accepting.
3. Each transition either pushes a symbol onto the stack (a *push* move) or pops one off the stack (a *pop* move), but it does not do both at the same time.

For a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$, we construct a CFG $G = (V, \Sigma, R, S)$ with the following steps:

1. $V = \{A_{pq} \mid p, q \in Q\}$
2. $S = A_{q_0, q_{\text{accept}}}$
3. R contains the following three sets of rules:
   - For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\varepsilon$, if $(r, t) \in \delta(p, a, \varepsilon)$ and $(q, \varepsilon) \in \delta(s, b, t)$, the rule
   $$A_{pq} \to aA_{rs}b$$
   - For each $p, q, r \in Q$, the rule
   $$A_{pq} \to A_{pr}A_{rq}$$
   - For each $p \in Q$, the rule $A_{pp} \to \varepsilon$

Proof: As stated on pp. 122–123.

Example: PDA P that recognizes $\{\mathtt{0}^n\mathtt{1}^n \mid n > 0\}$:



No need to modify P because it fits the three criteria. Construct the CFG $G = (V, \Sigma, R, A_{14})$ as

1. $V = \{A_{11}, A_{12}, A_{13}, A_{14}, A_{21}, A_{22}, \ldots, A_{44}\}$
2. R contains the rules:
   - For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\varepsilon$, if $(r, t) \in \delta(p, a, \varepsilon)$ and $(q, \varepsilon) \in \delta(s, b, t)$, the rule
   $$A_{pq} \to aA_{rs}b$$
   $\delta(1, \varepsilon, \varepsilon) = \{(2, \mathtt{\$})\}$,
   $\delta(2, \mathtt{0}, \varepsilon) = \{(2, \mathtt{0})\}, \delta(2, \mathtt{1}, \mathtt{0}) = \{(3, \varepsilon)\}$,
   $\delta(3, \mathtt{1}, \mathtt{0}) = \{(3, \varepsilon)\}, \delta(3, \varepsilon, \mathtt{\$}) = \{(4, \varepsilon)\}$
   $$\begin{aligned} A_{14} &\to A_{23} \\ A_{23} &\to \mathtt{0}A_{22}\mathtt{1} \\ A_{23} &\to \mathtt{0}A_{23}\mathtt{1} \end{aligned}$$
   - $A_{11} \to A_{11}A_{11}, A_{11} \to A_{12}A_{21}, A_{11} \to A_{13}A_{31}, \ldots,$
   $A_{12} \to A_{11}A_{12}, A_{12} \to A_{12}A_{22}, A_{12} \to A_{13}A_{32}, \ldots,$
   $\ldots,$
   $A_{43} \to A_{41}A_{13}, A_{43} \to A_{42}A_{23}, A_{43} \to A_{43}A_{33}, \ldots,$
   $\ldots, A_{44} \to A_{43}A_{34}, A_{44} \to A_{44}A_{44}$
   - $A_{11} \to \varepsilon, A_{22} \to \varepsilon, A_{33} \to \varepsilon, A_{44} \to \varepsilon$

Claim 2.30: If A_{pq} generates x, then x can bring P from p with empty stack to q with empty stack. p. 123

Proof: As stated on p. 123.

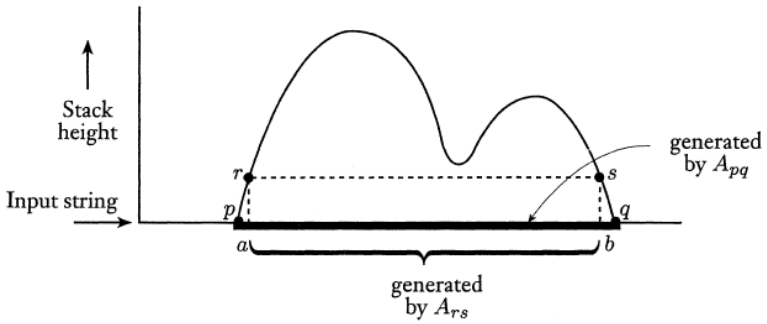
Induction on the number of steps in the derivation.

Basis: The derivation has 1 step: $A_{pp} \to \varepsilon$.

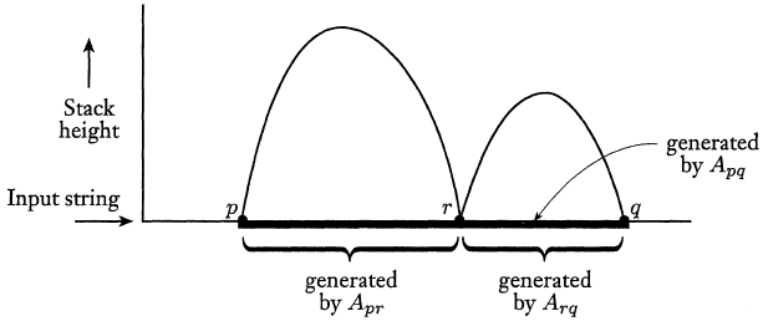
Induction step: Assume true for derivations of length at most k, where $k \geq 1$, and prove true for derivations of length $k+1$.

Suppose $A_{pq} \stackrel{*}{\Rightarrow} x$ with $k+1$ steps. The first step is:

- $A_{pq} \Rightarrow aA_{rs}b$
  Consider that $A_{rs} \stackrel{*}{\Rightarrow} y$, so $x = ayb$. Because $A_{rs} \stackrel{*}{\Rightarrow} y$ with k steps, P can go from r on empty stack to s on empty stack. Because $A_{pq} \to aA_{rs}b$ is a rule in the grammar, $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), for some stack t. Hence,



- $A_{pq} \Rightarrow A_{pr}A_{rq}$

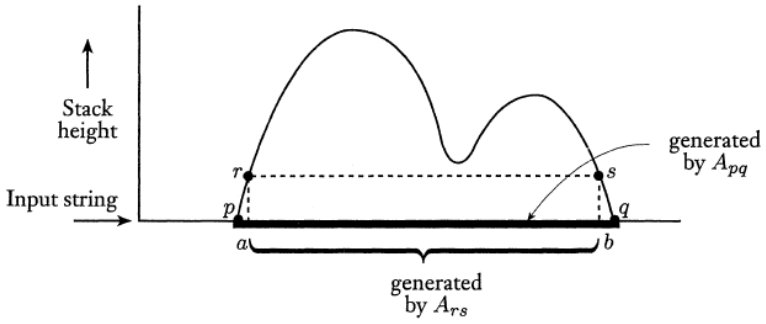


p. 123

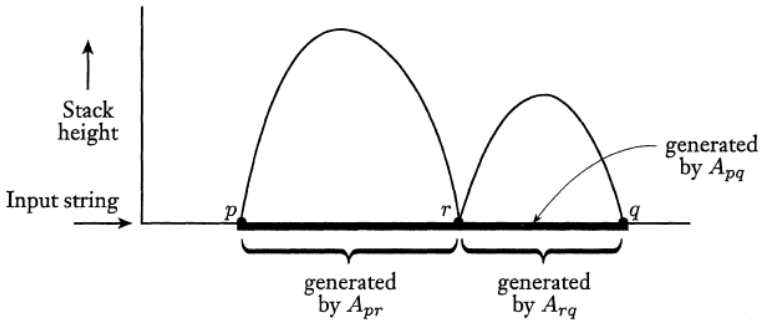
Claim 2.31: If x can bring P from p with empty stack to q with empty stack, A_{pq} generates x.
Proof: As stated on pp. 123–124.

Induction on the number of steps in the computation.
Basis: The computation has 0 steps.
It starts and ends at the same state—say, p. Because the rule $A_{pp} \to \varepsilon$ is in the grammar, $A_{pp} \stackrel{*}{\Rightarrow} \varepsilon$.
Induction step: Assume true for computations of length at most k, where $k \geq 0$, and prove true for computations of length $k+1$.
Suppose x brings P from p to q with empty stacks in $k+1$ steps. There are only two conditions:

- The stack is empty only at the beginning and end. The symbol, t, that is pushed at the first move must be popped at the last move. Let a be the input in the first move, b be the input in the last move, r be the state after move, and s be the state before the last move. $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), and so rule $A_{pq} \to aA_{rs}b$ is in the grammar. Let y be the portion of x without a and b, so $x = ayb$. P can go from r with an empty stack to s with an empty stack on input y with $(k+1) - 2 = k - 1$ steps. Thus, $A_{rs} \stackrel{*}{\Rightarrow} y$. Hence, $A_{pq} \stackrel{*}{\Rightarrow} x$.

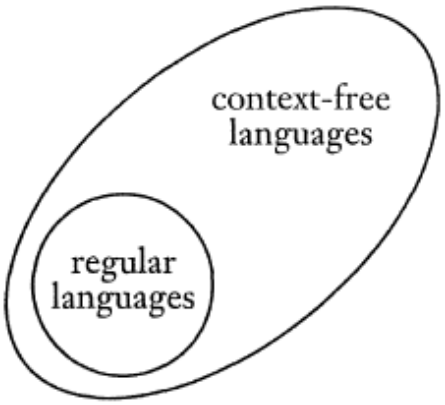


- The stack is empty elsewhere.
  Let r be a state where the stack becomes empty other than at the beginning or end of the computation on x. Then the portions of the computation from p to r and from r to q each contain at most k steps. Say that y is the input during the first portion and z is the input during the second portion. The induction hypothesis tells us that $A_{pr} \stackrel{*}{\Rightarrow} y$ and $A_{rq} \stackrel{*}{\Rightarrow} z$. Because rule $A_{pq} \rightarrow A_{pr}A_{rq}$ is in the grammar, $A_{pq} \stackrel{*}{\Rightarrow} x$.



Corollary 2.32: Every regular language is context free.

p. 124



2.3 Non-Context-Free Languages

Example 2.36, p. 128

Consider the language

$$B = \{ \mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0 \} .$$

2.3.1 Pumping Lemma for CFLs

p. 125

Theorem 2.34—Pumping lemma for context-free languages: If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces, $s = uvxyz$ satisfying the conditions

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

Fig. 2.35, p. 126

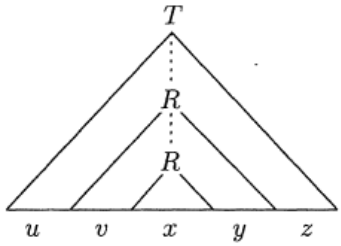
Proof idea: When the string is long enough, some variable(s) must be repeatedly used in derivations.
Proof: As stated on p. 127.

Let G be a CFG for CFL A. Let b be the **maximum number of symbols** in the right-hand side of a rule. From the start variable, there are at most b leaves for 1 step; at most b^2 leaves for 2 steps; at most b^h leaves for h steps. Conversely, if a generated string is at least $b^h + 1$ long, each of its parse tree must be at least $h + 1$ high.

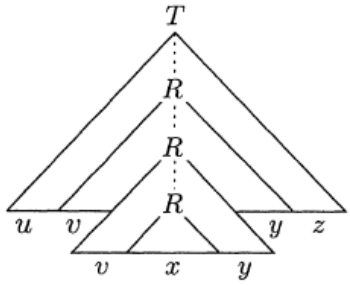
Say $|V|$ is the number of variables in G. We set p, the pumping length, to be $b^{|V|} + 1$. Now if s is a string in A and its length is p or more, its parse tree must be at least $|V| + 1$ high.

For any such string s, let τ be the parse tree with the smallest number of nodes. We know that τ must be at least $|V|+1$ high, so it must contain a path from the root to a leaf of length at least $|V|+1$. That path has at least $|V|+2$ nodes; one at a terminal, the others at variables. Hence that path has at least $|V|+1$ variables. With G having only $|V|$ variables, some variable R appears more than once on that path. So we have:

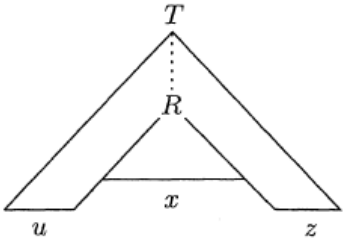
- $s = uvxyz$



- uv^ixy^iz, where $i = 2$



- uv^0xy^0z



Condition 2: Both v and y are not ε. τ is the parse tree with the smallest number of nodes.

Condition 3: vxy has length at most p. Choose R so that both occurrences fall within the bottom $|V|+1$ variables on the path.

How to use the pumping lemma for CFLs:

1. Assume the language, say, B, is context free in order to obtain a contradiction.
2. By the pumping lemma for CFLs, a pumping length p exists, and any string $w \in B$ can be pumped if $|w| \geq p$.
3. Find a string $s \in B$, $|s| \geq p$, that s cannot be pumped as described in the pumping lemma.
4. The contradiction is obtained, and therefore, B is proved to be non-context-free.

p. 128

Example 2.36:
Use the pumping lemma to show that the language $B = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ is not context free.

Let p be the pumping length for B. Select the string $s = \mathtt{a}^p\mathtt{b}^p\mathtt{c}^p$. Two possibilities:

1. Both v and y contain only one type of alphabet symbol: Check uv^2xy^2z.
2. Either v or y contains more than one type of alphabet symbol: Check uv^2xy^2z.

2.3.2 Ogden's Lemma for CFLs

Exercise 7.2.3: HMU 3rd, p. 286

Ogden's lemma for context-free languages: If A is a context-free language, then there is a number p where, if s is any string in A of length at least p, in which we select any p or more ***positions*** to be distinguished, then s may be divided into five pieces, $s = uvxyz$ satisfying the conditions

1. for each $i \geq 0, uv^ixy^iz \in A$;
2. vy has at least one distinguished positions;
3. vxy has at most p distinguished positions.

Note: By letting all positions of the string to be distinguished, Ogden's lemma yields the pumping lemma.

Example: $L = \left\{ \mathtt{a}^i\mathtt{b}^j\mathtt{c}^k\mathtt{d}^l \;\middle|\; i = 0 \text{ or } j = k = l \right\}$.

We would fail if we use the pumping lemma to show L is not a CFL. For any $p > 0$ and $s \in L$ such that $|s| \geq p$,

1. $s = \mathtt{b}^j\mathtt{c}^k\mathtt{d}^l$
   j, k, l can be any integer. Let $s = uvxyz$, then it is always possible to choose u, v, x, y, z so that $|vy| > 0, |vxy| \leq p$ and $\forall n \geq 0, uv^nxy^nz \in L$. For example, choose vxy to have only $\mathtt{b}$'s, then $uv^nxy^nz \in L$ for all $n \geq 0$. In particular, choose $v = \mathtt{b}, x = \varepsilon, y = \varepsilon$, then $vxy = \mathtt{b}$ and $uv^nxy^nz \in L$ for all $n \geq 0$.

2. $s = \mathtt{a}^i\mathtt{b}^j\mathtt{c}^k\mathtt{d}^l, i > 0$
   Let $s = uvxyz$, then it is always possible to choose u, v, x, y, z so that $|vy| > 0, |vxy| \leq p$, and $\forall n \geq 0, uv^nx^nyz \in L$. For example, choose $v = \mathtt{a}, x = \varepsilon, y = \varepsilon$, then $\forall n \geq 0, uv^nx^nyz \in L$.

Now we use Ogden's lemma to show that L is not a CFL.

Suppose L is a CFL. Let p be the integer in the lemma. Choose $s = \mathtt{a}\mathtt{b}^p\mathtt{c}^p\mathtt{d}^p$, and select all positions but the first of s to be distinguished. Let $s = uvxyz$. Then vy must contain one of $\{ \mathtt{b}, \mathtt{c}, \mathtt{d} \}$ but cannot contain all of $\{ \mathtt{b}, \mathtt{c}, \mathtt{d} \}$ because vxy has at most p positions. Therefore, uv^2xy^2z has at least one a and does not have equal numbers of b's, c's, and d's, which means that uv^2xy^2z cannot be in L. A contradiction! □

The Language of Machines, Floyd and Beigel, pp. 361–374

2.3.3 Ogden's Lemma for CFGs

Lemma: Let G be a CFG. There exists a natural number p (depending on G) such that for all $s \in L(G)$, if s is marked so that it has p or more distinguished positions, then there exists a variable A and terminal strings u, v, x, y, z satisfying the following conditions:

1. $s = uvxyz$;
2. either v or y has distinguished positions;
3. vxy has no more than p distinguished positions;
4. $S \stackrel{*}{\Rightarrow} uAz$;
5. $A \stackrel{*}{\Rightarrow} vAy$;
6. $A \stackrel{*}{\Rightarrow} x$;
7. for each $i \geq 0$, $A \stackrel{*}{\Rightarrow} v^i x y^i$.

2.3.4 Inherently Ambiguous CFLs

Theorem: There exists an inherently ambiguous CFL.

Problem 2.41, p. 158

Proof: One of such languages is

$$L = \left\{ \mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \;\middle|\; i = j \text{ or } j = k \right\} .$$

We will show that for all possible grammars generating L, there is a string of the form $\mathtt{a}^m\mathtt{b}^m\mathtt{c}^m$ that can be parsed in two different ways.

Let G be any CFG for L, and let p be a number for G as mentioned in Ogden's lemma for CFGs. Without loss of generality, assume that $p > 1$.

Let $s = \mathtt{a}^p\mathtt{b}^p\mathtt{c}^{p+p!}$ ($s' = \mathtt{a}^{p+p!}\mathtt{b}^p\mathtt{c}^p$). Mark all the $\mathtt{b}$'s as distinguished. By Ogden's lemma, the grammar G contains a variable A (B) such that the following conditions hold:

$$S \stackrel{*}{\Rightarrow} uAz \quad (S \stackrel{*}{\Rightarrow} u'Bz')$$

for all $n \geq 0$,

$$A \stackrel{*}{\Rightarrow} v^n x y^n \quad (B \stackrel{*}{\Rightarrow} v'^n x' y'^n)$$

The only way to pump is for $s = uvxyz$ ($s' = u'v'x'y'z'$) where $v = \mathtt{a}^k, y = \mathtt{b}^k$, and $1 \leq k \leq p$ ($v' = \mathtt{b}^{k'}, y' = \mathtt{c}^{k'}$, and $1 \leq k' \leq p$). Letting $n = \frac{p!}{k} + 1$ ($n' = \frac{p!}{k'} + 1$), we see that

$$A \stackrel{*}{\Rightarrow} \mathtt{a}^{p!+k} x \mathtt{b}^{p!+k} \quad (B \stackrel{*}{\Rightarrow} \mathtt{b}^{p!+k'} x' \mathtt{c}^{p!+k'})$$

So

$$\begin{array}{rcll} S & \stackrel{*}{\Rightarrow} & uAz & (S \stackrel{*}{\Rightarrow} u'Bz' \\ & \stackrel{*}{\Rightarrow} & u\mathtt{a}^{p!+k}x\mathtt{b}^{p!+k}z & \stackrel{*}{\Rightarrow} u'\mathtt{b}^{p!+k'}x'\mathtt{c}^{p!+k'}z' \\ & \stackrel{*}{\Rightarrow} & u\mathtt{a}^{p!}vx\mathtt{b}^{p!}yz & \stackrel{*}{\Rightarrow} u'\mathtt{b}^{p!}v'x'\mathtt{c}^{p!}y'z' \\ & \stackrel{*}{\Rightarrow} & \mathtt{a}^{p+p!}\mathtt{b}^{p+p!}\mathtt{c}^{p+p!} & \stackrel{*}{\Rightarrow} \mathtt{a}^{p+p!}\mathtt{b}^{p+p!}\mathtt{c}^{p+p!}) \end{array}$$

Claim: There are two different parse trees for $\mathtt{a}^m\mathtt{b}^m\mathtt{c}^m$, where $m = p + p!$

$$\begin{aligned} S &\stackrel{*}{\Rightarrow} uAz &&\stackrel{*}{\Rightarrow} \mathtt{a}^{p+p!}\mathtt{b}^{p+p!}\mathtt{c}^{p+p!} \\ S &\stackrel{*}{\Rightarrow} u'Bz' &&\stackrel{*}{\Rightarrow} \mathtt{a}^{p+p!}\mathtt{b}^{p+p!}\mathtt{c}^{p+p!} \end{aligned}$$

Proof: Let

$$\alpha = \mathtt{a}^{p!+k}x\mathtt{b}^{p!+k}, \beta = \mathtt{b}^{p!+k'}x'\mathtt{c}^{p!+k'} \ .$$

Then $A \stackrel{*}{\Rightarrow} \alpha$ and $B \stackrel{*}{\Rightarrow} \beta$. α has at least $p! + 1$ $\mathtt{b}$'s, so does β. But $\mathtt{a}^m\mathtt{b}^m\mathtt{c}^m$ has only $p! + p$ $\mathtt{b}$'s. Since $p! + 1 > \frac{1}{2}(p! + p)$, α has more than half of the $\mathtt{b}$'s in the final string; so does β.
$\therefore$ α and β must overlap.
Furthermore, α and β both contain some symbols that the other does not have.
$\therefore$ Neither α nor β is a substring of the other.
By way of contradition, suppose α and β belong to the same parse tree. Then either

1. α and β are disjoint, or
2. either α or β is a substring of the other.

This contradicts the results we just derived. $\square_{\text{Claim}}$

$\therefore$ G must be ambiguous!
Since G is an arbitrary grammar for L, L is inherently ambiguous. $\square$

2.4 Properties of CFLs

2.4.1 Substitutions

Substitution: $s : \Sigma \to \mathcal{P}(\Pi^*)$

- is a generalization of the homomorphism we defined in the study of regular languages
- can be extended in for strings and languages:
  $s : \Sigma^* \to \mathcal{P}(\Pi^*)$

$$\begin{aligned} s(\epsilon) &= \epsilon \\ s(xa) &= s(x)s(a) \end{aligned}$$

$s : \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Pi^*)$

$$s(L) = \{\ s(w)\ \mid\ w \in L\ \}$$

Theorem 7.23, HMU 3rd, p. 288

Theorem: If L is a `CFL` over Σ, s is a substitution on Σ such that $s(a)$ is a `CFL` for each $a \in \Sigma$, then $s(L)$ is also a `CFL`.

Proof idea: Let $G = (V, \Sigma, R, S)$ be the `CFG` for L and $G_a = (V_a, \Sigma_a, R_a, S_a)$ be the `CFG` for $s(a)$, for each symbol $a \in \Sigma$. Construct a new grammar $G' = (V', \Sigma', R', S')$ for $s(L)$:

- V' is the union of V and all the V_a's for $a \in \Sigma$.
- Σ' is the union of all the Σ_a's for $a \in \Sigma$.
- R' consists of
  1. All rules in each R_a;
  2. The rule in R but with each terminal a in their bodies replaced by S_a everywhere a occurs.

Claim: $w \in L(G')$ iff $w \in s(L)$.
Proof: Omitted. □

2.4.2 Closure Properties

Let A and B be context-free languages. Let R be a regular language. The results of the following operations are all context-free languages:

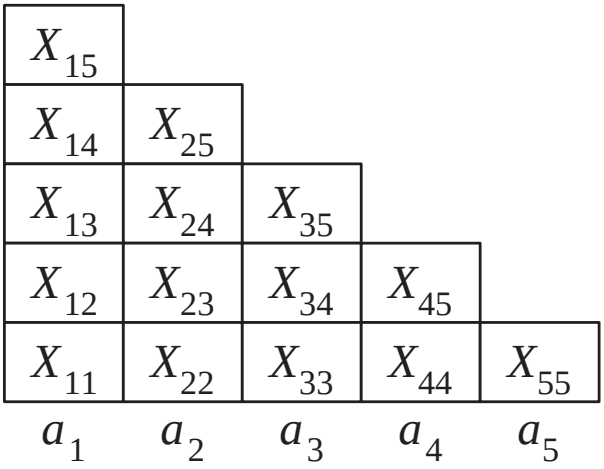
- Substition: $s(A)$;
- Union: $A \cup B$;
- Concatenation: AB;
- Star: A^*;
- Reversal: A^R;
- Homomorphism: $h(A)$;
- Inverse homomorphism: $h^{-1}(A)$;
- †Intersection: $A \cap R$;
- †Difference: $A - R$ ($A - R = A \cap \overline{R}$).

2.4.3 Membership Test: CYK Algorithm

HMU 3rd, pp. 303–307

($w \in L$?)

1. Let $G = (V, \Sigma, R, S)$ is the context-free grammar for L in Chomsky normal form.
2. The input string $w = a_1 a_2 \cdots a_n$ is of length n.
3. Construct the table with horizontal axis (i) being the positions of the symbols in w, X_{ij} is the set of variables A such that $A \stackrel{*}{\Rightarrow} a_i a_{i+1} \cdots a_j$. We ask whether $S \in X_{1n}$ (i.e., $S \stackrel{*}{\Rightarrow} a_1 a_2 \cdots a_n$.)



Basis: X_{ii} is the set of variables A such that $A \to a_i$ can be found in the rule.

Induction step: X_{ij} contains A if we can find variables B and C, and integer k such that

1. $i \leq k < j$;
2. B is in X_{ik} ;
3. C is in $X_{k+1,j}$;
4. $A \to BC$ is a rule.

Example: $w = baaba$

$$
\begin{array}{rrcl|l}
G: & S & \to & AB & BC \\
 & A & \to & BA & a \\
 & B & \to & CC & b \\
 & C & \to & AB & a
\end{array}
$$

{S,A,C}				
Ø	{S,A,C}			
Ø	{B}	{B}		
{S,A}	{B}	{S,C}	{S,A}	
{B}	{A,C}	{A,C}	{B}	{A,C}
b	*a*	*a*	*b*	*a*

Since S is in $X_{15} = \{ S, A, C \}$, we have $w \in L(G)$.
Answer: Yes! □

Chapter 3

The Church-Turing Thesis

Intuitive notion of algorithms
equals
Turing machine algorithms

Fig. 3.22, p. 183

3.1 Turing Machines

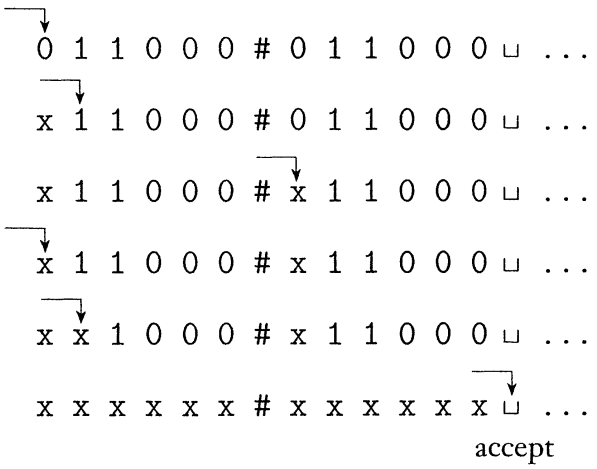
Finite control + unlimited and unrestricted memory.

Fig. 3.1, p. 166

1. Write on the tape and read from it.
2. Move the head to the left and to the right.
3. The tape is infinite.
4. Reject and accept immediately.

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

Fig. 3.2, p. 167



3.1.1 Formal Definition of A Turing Machine

Definition 3.3, p. 168

A ***Turing machine*** is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states;
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$;
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$;
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{ L, R \}$ is the transition function;
5. $q_0 \in Q$ is the start state;
6. $q_{\text{accept}} \in Q$ is the accept state;
7. $q_{\text{reject}} \in Q$ is the reject state.

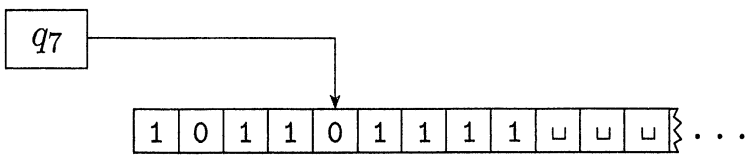
A ***configuration*** of the Turing machine consists of

- the current state;
- the current tape contents;
- the current head location.

and is represented as uqv for a state q, two strings u and v over the tape alphabet Γ, and the current head at the first symbol of v.

Fig. 3.4, p. 169

Example: $\texttt{1011}q_7\texttt{01111}$



If the Turing machine can legally go from configuration C_1 to configuration C_2, we say that C_1 ***yields*** C_2.

Formally, if $a, b, c \in \Gamma$, $u, v \in \Gamma^*$, and $q_i, q_j \in Q$, uaq_ibv and uq_jacv are two configurations.

$$ua\ q_i\ bv \quad \text{yields} \quad u\ q_j\ acv$$

if $\delta(q_i, b) = (q_j, c, L)$.

$$ua\ q_i\ bv \quad \text{yields} \quad uac\ q_j\ v$$

if $\delta(q_i, b) = (q_j, c, R)$.

Special cases: p. 169

- The left-hand end of the tape;
- The right-hand end of the tape.

Special configurations: p. 169

- ***start configuration***: q_0w;
- ***accepting configuration***: the state is q_{accept};
- ***rejecting configuration***: the state is q_{reject};
- ***halting configuration***: accepting and rejecting.

A Turing machine M ***accepts*** input w if a sequence of configurations $C_1, C_2, \ldots, C_k$ exists, where

1. C_1 is the start configuration of M on input w;
2. Each C_i yields C_{i+1};
3. C_k is an accepting configuration.

p. 170

The collection of strings that M accepts is ***the language of M***, or ***the language recognized by M***, denoted $L(M)$.

Definition 3.5, p. 170

Call a language ***Turing-recognizable*** if some Turing machine recognizes it. (Also known as ***recursively enumerable language***)

Definition 3.6, p. 170

Call a language ***Turing-decidable*** or simply ***decidable*** if some Turing machine decides it. (Also known as ***recursive language***)

3.2 Variants of Turing Machines

3.2.1 Multitape Turing Machines

k-tape Turing machine:

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

p. 177

Theorem 3.13: Every multitape Turing machine has an equivalent single-tape Turing machine.

Fig. 3.14, p. 177

Proof: As stated on p. 177.

p. 178

Corollary 3.15: A language is Turing-recognizable if and only if some multitape Turing machine recognizes it.

Proof: As stated on p. 178.

3.2.2 Nondeterministic Turing Machines

Nondeterministic—several choices at any point:

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

Theorem 3.16: Every nondeterministic Turing machine has an equivalent deterministic Turing machine.
Proof: As stated on p. 179.

p. 178

Fig. 3.17, p. 179

Corollary 3.18: A language is Turing-recognizable if and only if some nondeterministic Turing machine recognizes it.
Proof: As stated on p. 180.

p. 180

Corollary 3.19: A language is decidable if and only if some nondeterministic Turing machine decides it.
Proof: As stated on p. 190.

Exercise 3.3, p. 187

3.2.3 Enumerators

Enumerator: Enumerates strings.

Fig. 3.20, p. 180

Theorem 3.21: A language is Turing-recognizable if and only if some enumerator enumerates it.
Proof: As stated on p. 181.

p. 181

3.3 The Definition of Algorithm

Algorithms, *procedures*, *recipes*: What are they exactly?

3.3.1 Hilbert's Problems

Find an *algorithm* to determine whether a given polynomial Diophantine equation with integer coefficients has an integer solution.

Define the term *algorithm* by using

- Church: λ-calculus;
- Turing: Turing machines.

Turing-recognizable? Turing-decidable?

$$\begin{aligned} D &= \{\, p \mid p \text{ is with an integral root} \,\} \\ D_1 &= \{\, p \mid p \text{ is over } x \text{ with an integral root} \,\} \end{aligned}$$

3.3.2 Terminology for Describing Turing Machines

The different levels of detail:

- *formal description*: Spells out in full the Turing machine's states, transition function, and so on;
- *implementation description*: Describes the way that the Turing machine moves its head and the way that it stores data on its tape;
- *high-level description*: Describes an algorithm.

Chapter 4

Decidability

4.1 Decidable Languages

4.1.1 Decidable Problems Concerning Regular Languages

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

Theorem 4.1: $A_{\texttt{DFA}}$ is a decidable language. pp. 194–195
Proof: As stated on p. 195.

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

Theorem 4.2: $A_{\texttt{NFA}}$ is a decidable language. p. 195
Proof: As stated on p. 195.

$A_{\texttt{REX}} = \{ \langle R, w \rangle \mid R \text{ is a regular expression that generates string } w \}$

Theorem 4.3: $A_{\texttt{REX}}$ is a decidable language. p. 196
Proof: As stated on p. 196.

$$E_{\texttt{DFA}} = \{ \langle A \rangle \mid A \text{ is a } \texttt{DFA} \text{ and } L(A) = \emptyset \}$$

p. 196

Theorem 4.4: $E_{\texttt{DFA}}$ is a decidable language.
Proof: As stated on p. 196.

$$EQ_{\texttt{DFA}} = \{ \langle A, B \rangle \mid A \text{ and } B \text{ are } \texttt{DFA}\text{s and } L(A) = L(B) \}$$

p. 197

Theorem 4.5: $EQ_{\texttt{DFA}}$ is a decidable language.
Proof idea: $L(C) = \Big(L(A) \cap \overline{L(B)}\Big) \cup \Big(\overline{L(A)} \cap L(B)\Big)$
Proof: As stated on p. 197.

4.1.2 Decidable Problems Concerning Context-Free Languages

$$A_{\texttt{CFG}} = \{ \langle G, w \rangle \mid G \text{ is a } \texttt{CFG} \text{ that generates string } w \}$$

p. 198

Theorem 4.7: $A_{\texttt{CFG}}$ is a decidable language.
Proof idea: The number of derivation steps is bounded for a `CFG` in Chomsky normal form.
Proof: As stated on p. 198.

$$E_{\texttt{CFG}} = \{ \langle G \rangle \mid G \text{ is a } \texttt{CFG} \text{ and } L(G) = \emptyset \}$$

p. 199

Theorem 4.8: $E_{\texttt{CFG}}$ is a decidable language.
Proof idea: Can any terminal string be generated?
Proof: As stated on p. 199.

$EQ_{\texttt{CFG}} = \{ \langle G, H \rangle \mid G \text{ and } H \text{ are } \texttt{CFG}\text{s and } L(G) = L(H) \}$

$EQ_{\texttt{CFG}}$ is **UNDECIDABLE**. See Chapter 5.

Theorem 4.9: Every context-free language is decidable. p. 200

Proof idea: Check whether or not the grammar G can generate the string w by applying Theorem 4.7.

Proof: As stated on p. 200.

Understand the relationship among classes of languages. Fig. 4.10, p. 201

4.2 The Halting Problem

Is there any problem algorithmically unsolvable?

$$A_{\texttt{TM}} = \{ \langle M, w \rangle \mid M \text{ is a } \texttt{TM} \text{ that accepts } w \}$$

Theorem 4.11: $A_{\texttt{TM}}$ is undecidable. p. 202

$A_{\texttt{TM}}$ is Turing-recognizable and can be recognized by

U = "On input $\langle M, w \rangle$, where M is a $\texttt{TM}$ and w is a string:

1. Simulate M on input w.
2. If M ever enters its accept state, *accept*; if M ever enters its reject state, *reject*."

4.2.1 The Diagonalization Method

Measuring the sizes of infinite sets.

Definition 4.12, p. 203

one-to-one:

$$x \neq y \Rightarrow f(x) \neq f(y) \ ;$$

onto:

$$\forall b \in B, \exists a \in A, f(a) = b \ ;$$

same size:

$$f : A \to B \text{ is one-to-one and onto} \ ;$$

Example 4.13, p. 203

and such f is a ***correspondence***.

Definition 4.14, p. 203

A set A is ***countable*** if either it is finite or it has the same size as $\mathcal{N}$.

p. 205

Theorem 4.17: $\mathcal{R}$ is uncountable.
Proof: As stated on pp. 205–206.

p. 206

Theorem 4.18: Some languages are not Turing-recognizable.
Proof: As stated on pp. 206–207.

4.2.2 The Halting Problem Is Undecidable

$$A_{\texttt{TM}} = \{ \langle M, w \rangle \mid M \text{ is a } \texttt{TM} \text{ that accepts } w \}$$

p. 202

Theorem 4.11: $A_{\texttt{TM}}$ is undecidable.
Proof: As stated on p. 207.

pp. 208–209

What exactly H and D do?

4.2.3 A Turing-Unrecognizable Language

A language is ***co-Turing-recognizable*** if it is the complement of a Turing-recognizable language. p. 209

Theorem 4.22: A language is decidable iff it is Turing-recognizable and co-Turing-recognizable. p. 209
Proof: As stated on pp. 209–210.

Corollary 4.23: $\overline{A_{\texttt{TM}}}$ is not Turing-recognizable. p. 210
Proof: As stated on p. 210.

Let's enumerate the binary strings as

w_1	ϵ	w_2	0	w_3	1
w_4	00	w_5	01	w_6	10
w_7	11	w_8	000	w_9	001
$\ldots$					

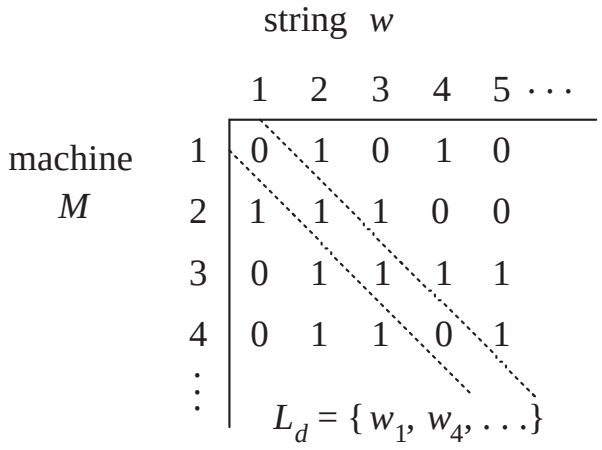
"$1w_i$" is exactly the binary representation of i. Let's consider w_i as a `TM` and define

$$L_d = \{ \langle w_i \rangle \mid w_i \text{ does not accept } w_i \}$$

Theorem: No Turing machine accepts L_d. Theorem 9.2, HMU 3rd, p. 382
Proof: Suppose there is a `TM` M that accepts L_d. Let the binary code for M is w_i.

1. If w_i is in L_d, then M accepts w_i; hence w_i accepts w_i. By the definition of L_d, we have w_i is not in L_d, a contradiction.
2. If w_i is not in L_d, then M does not accept w_i; hence w_i does not accept w_i. By the definition of L_d, we have w_i is in L_d, a contradiction. □

Diagonalization:



Chapter 5

Reducibility

If problem A ***reduces*** to problem B, we can use a solution to B to solve A. ***Reducibility*** says NOTHING about solving A or B alone.

When A is reducible to B, solving A cannot be harder than solving B because a solution to B gives a solution to A. Therefore, if A is reducible to B,

- B is decidable $\Rightarrow$ A is decidable;
- A is undecidable $\Rightarrow$ B is undecidable.

5.1 Undecidable Problems from Language Theory

$HALT_{\texttt{TM}} = \{ \langle M, w \rangle \mid M \text{ is a } \texttt{TM} \text{ and } M \text{ halts on input } w \}$

Theorem 5.1: $HALT_{\texttt{TM}}$ is undecidable. p. 216

Proof idea: Assume a decider R for $HALT_{\texttt{TM}}$ exists. Construct the decider S for $A_{\texttt{TM}}$. Contradiction.

Proof: As stated on p. 217.

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

p. 217

Theorem 5.2: $E_{\texttt{TM}}$ is undecidable.

Proof idea: Assume a decider R for $E_{\texttt{TM}}$ exists. Construct the decider S for $A_{\texttt{TM}}$. Contradiction.

Proof: As stated on p. 218.

$$REGULAR_{\texttt{TM}} =$$
$$\{ \langle M \rangle \mid M \text{ is a } \texttt{TM} \text{ and } L(M) \text{ is a regular language} \}$$

p. 219

Theorem 5.3: $REGULAR_{\texttt{TM}}$ is undecidable.

Proof idea: Assume a decider R for $REGULAR_{\texttt{TM}}$ exists. Construct the decider S for $A_{\texttt{TM}}$. Contradiction.

Proof: As stated on p. 219.

Problem 5.16, p. 240

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

[More formal version]: Let P be a language consisting of Turing machine descriptions where P fulfills two conditions. First, P is nontrivial—it contains some, but not all, TM descriptions. Second, P is a property of the TM's language—whenever $L(M_1) = L(M_2)$, we have $\langle M_1 \rangle \in P$ iff $\langle M_2 \rangle \in P$. Here M_1 and M_2 are any TMs. Prove that P is an undecidable language.

Proof: As stated on p. 243.

$EQ_{\texttt{TM}} = \{ \langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are } \texttt{TM}\text{s and } L(M_1) = L(M_2) \}$

Theorem 5.4: $EQ_{\texttt{TM}}$ is undecidable. p. 220
Proof idea: Assume a decider R for $EQ_{\texttt{TM}}$ exists. Construct the decider S for $E_{\texttt{TM}}$. Contradiction.
Proof: As stated on p. 220.

5.1.1 Reductions via Computation Histories

The ***computation history*** for a Turing machine on an input is simply the sequence of configurations that the machine goes through as it processes the input. p. 220

An ***accepting computation history*** for M on w is a sequence of configurations, $C_1, C_2, \ldots, C_\ell$, where C_1 is the start configuration of M on w, C_ℓ is an accepting configuration of M, and each C_i legally follows from C_{i-1} according to the rules of M. Definition 5.5, p. 221

A ***rejecting computation history*** for M on w is defined similarly, except that C_ℓ is a rejecting configuration.

A ***linear bounded automaton*** is a restricted type of Turing machine wherein the tape head isn't permitted to move off the portion of the tape containing the input. Definition 5.6, Fig. 5.7, p. 221

`LBA`s can be used to decide $A_{\texttt{DFA}}$, $A_{\texttt{CFG}}$, $E_{\texttt{DFA}}$, and $E_{\texttt{CFG}}$. Every `CFL` can be decided by an `LBA`. p. 222

$$A_{\texttt{LBA}} = \{ \langle M, w \rangle \mid M \text{ is an } \texttt{LBA} \text{ that accepts input string } w \}$$

p. 222

Lemma 5.8: Let M be an `LBA` with q states and g symbols in the tape alphabet. There are exactly qng^n distinct configurations of M for a tape of length n.
Proof idea: Just count it: q states $\times$ n positions $\times$ g^n possible tape content $= qng^n$.
Proof: As stated on p. 222.

p. 222

Theorem 5.9: $A_{\texttt{LBA}}$ is decidable.
Proof idea: Because the number of all possible configurations can be exactly computed, the looping of a machine can therefore be determined.
Proof: As stated on p. 223.

$$E_{\texttt{LBA}} = \{ \langle M \rangle \mid M \text{ is an } \texttt{LBA} \text{ where } L(M) = \emptyset \}$$

p. 223

Theorem 5.10: $E_{\texttt{LBA}}$ is undecidable.

Fig. 5.11, p. 224; Fig. 5.12, p. 225

Proof idea: Given a `TM` M and an input w, construct an `LBA` B that can recognize a language comprising all accepting computation histories for M on w.
Proof: As stated on p. 223.

$$ALL_{\texttt{CFG}} = \{ \langle G \rangle \mid G \text{ is a } \texttt{CFG} \text{ and } L(G) = \Sigma^* \}$$

p. 225

Theorem 5.13: $ALL_{\texttt{CFG}}$ is undecidable.

Fig. 5.14, p. 226

Proof idea: Given a `TM` M and an input w, construct an `PDA` D that can accept all strings except the accepting computation histories for M on w.
Proof: As stated on pp. 225–226.

5.2 A Simple Undecidable Problem

Post correspondence problem (***PCP***)

p. 227

A collection of dominos:

$$\left\{ \left[\frac{\texttt{b}}{\texttt{ca}}\right], \left[\frac{\texttt{a}}{\texttt{ab}}\right], \left[\frac{\texttt{ca}}{\texttt{a}}\right], \left[\frac{\texttt{abc}}{\texttt{c}}\right] \right\}$$

A ***match*** (abcaaabc):

$$\left[\frac{\texttt{a}}{\texttt{ab}}\right] \left[\frac{\texttt{b}}{\texttt{ca}}\right] \left[\frac{\texttt{ca}}{\texttt{a}}\right] \left[\frac{\texttt{a}}{\texttt{ab}}\right] \left[\frac{\texttt{abc}}{\texttt{c}}\right]$$

No match is possible for this collection of dominos:

$$\left\{ \left[\frac{\texttt{abc}}{\texttt{ab}}\right], \left[\frac{\texttt{ca}}{\texttt{a}}\right], \left[\frac{\texttt{acc}}{\texttt{ba}}\right] \right\}$$

The Post correspondence problem is to determine whether a collection of dominos has a match.

Formally, an instance of PCP is a collection P of dominos:

$$\left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \cdots \left[\frac{t_k}{b_k}\right], \right\} ,$$

and a match is a sequence $i_1, i_2, \ldots, i_\ell$, where $t_{i_1} t_{i_2} \cdots t_{i_\ell} = b_{i_1} b_{i_2} \cdots b_{i_\ell}$. The problem is to determine whether P has a match. Let

$$PCP = \{ \langle P \rangle \mid P \text{ is an instance of PCP with a match} \}$$

p. 228

Theorem 5.15: PCP is undecidable.

Proof idea: From any `TM` M and input w we can construct an instance P where a match is an accepting computation history for M on w. We choose the dominos in P so that making a match forces a simulation of M to occur. Three technical points:

1. Assume that M on w never attempts to move its head off the left-hand end of the tape.
2. If $w = \varepsilon$, we use the string $\sqcup$ in place of w in the construction.
3. Modify the PCP to require that a match starts with the first domino,

$$\left[\frac{t_1}{b_1}\right].$$

We call this problem the modified Post correspondence problem (MPCP). Let

$$MPCP = \{\langle P\rangle \mid P \text{ is an instance of PCP with a match starting with } \left[\frac{t_1}{b_1}\right]\}$$

Proof: As stated on pp. 228–233.

5.3 Mapping Reducibility

p. 234

Being able to reduce problem A to problem B by using a ***mapping reducibility*** means that a computable function, called a ***reduction***, exists that converts instances of problem A to instances of problem B.

5.3.1 Computable Functions

A Turing machine computes a function by starting with the input to the function to the tape and halting with the output of the function on the tape.

A function $f : \Sigma^* \to \Sigma^*$ is a ***computable function*** if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

Definition 5.17, p. 234

5.3.2 Formal Definition of Mapping Reducibility

Language A is ***mapping reducible*** to language B, written $A \leq_m B$, if there is a computable function $f : \Sigma^* \to \Sigma^*$, where for every w,

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

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

Definition 5.20, p. 235

Fig. 5.21, p. 235

Theorem 5.22: If $A \leq_m B$ and B is decidable, then A is decidable.
Proof: As stated on p. 236.

p. 236

Corollary 5.23: If $A \leq_m B$ and A is undecidable, then B is undecidable.

p. 236

Example 5.24, p. 236

Theorem 5.28: If $A \leq_m B$ and B is Turing-recognizable, then A is Turing-recognizable.
Proof: As stated on p. 236 (Theorem 5.22).

p. 237

p. 238

Corollary 5.29: If $A \leq_m B$ and A is not Turing-recognizable, then B is not Turing-recognizable.

p. 238

Theorem 5.30: $EQ_{\texttt{TM}}$ is neither Turing-recognizable nor co-Turing-recognizable.

Proof: As stated on p. 238.

There are only the following situations are possible for a language A and its complement $\overline{A}$:

- A and $\overline{A}$ are both decidable.
- Neither A nor $\overline{A}$ is Turing-recognizable.
- A is Turing-recognizable but undecidable, and $\overline{A}$ is not Turing-recognizable.

Chapter 7

Time Complexity

Decidable: Computationally solvable.
In theory or in practice?

Computational complexity theory—an investigation of

- the time,
- memory, or
- other resources

required for solving computational problems.

7.1 Measuring Complexity

Let M be a deterministic Turing machine that halts on all inputs. The ***running time*** or ***time complexity*** of M is the function $f : \mathcal{N} \to \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.

Definition 7.1, p. 276

We compute the running time of an algorithm purely as a function of the length of the string representing the input.

In ***worst-case analysis***, we consider the longest running time of all inputs of a particular length.

In ***average-case analysis***, we consider the average of all the running times of inputs of a particular length.

7.1.1 Big-O and Small-O Notation

In ***asymptotic analysis***, we seek to understand the running time of the algorithm when it is run on large inputs: Considering the highest order term. For example,

$$f(n) = 6n^3 + 2n^2 + 20n + 45$$

We describe that $f(n) = O(n^3)$.

Definition 7.2, p. 277

Let f and g be functions $f, g : \mathcal{N} \to \mathcal{R}^+$. Say that $\boldsymbol{f(n) = O(g(n))}$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$

$$f(n) \leq cg(n) \ .$$

When $f(n) = O(g(n))$ we say that $g(n)$ is an ***upper bound*** for $f(n)$, or more precisely, that $g(n)$ is an ***asymptotic upper bound*** for $f(n)$, to emphasize that we are suppressing constant factors.

Examples 7.3 & 7.4, p. 277

Polynomial bounds: n^c for some $c > 0$.

p. 278

Exponential bounds: $2^{(n^\delta)}$ for some $\delta > 0$.

Let f and g be functions $f, g : \mathcal{N} \to \mathcal{R}^+$. Say that $\boldsymbol{f(n) = o(g(n))}$ if

Definition 7.5, p. 278

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

In other words, $f(n) = o(g(n))$ means that, for any real number $c > 0$, a number n_0 exists, where $f(n) < cg(n)$ for all $n \geq n_0$.

Example 7.6, p. 278

7.1.2 Analyzing Algorithms

$$A = \left\{ \mathtt{0}^k\mathtt{1}^k \ \middle| \ k \geq 0 \right\}$$

A single-tape TM M_1 for A:

TM M_1, p. 279

- $O(n)$ steps for stage 1: Scan & reposition.
- $(n/2)O(n) = O(n^2)$ steps for stages 2 & 3.
- $O(n)$ steps for stage 4.

The total time of M_1 on an input of length n is

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

Definition 7.7, p. 279

Let $t : \mathcal{N} \to \mathcal{R}^+$ be a function. Define the ***time complexity class,* TIME($t(n)$)**, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

Because M_1 decides A in time $O(n^2)$ and TIME(n^2) contains all languages that can be decided in $O(n^2)$ time, we have $A \in \text{TIME}(n^2)$.

TM M_2, p. 280

A single-tape TM M_2 for A (verify M_2's correctness):

- $O(n)$ steps for every stage.
- $(1+\log_2 n)$ times for stages 2, 3, & 4: $(1+\log_2 n)O(n)$ steps.

The total time of M_2 on an input of length n is

$$O(n) + O(n \log n) = O(n \log n) \ .$$

Because M_2 decides A in time $O(n \log n)$ and TIME($n \log n$) contains all languages that can be decided in $O(n \log n)$ time, we have $A \in \text{TIME}(n \log n)$.

TM M_3, p. 281

A two-tape TM M_3 for A:

- $O(n)$ steps for every stage.

The total time of M_3 on an input of length n is $O(n)$, also called ***linear time***.

Because M_3 decides A in time $O(n)$ and TIME(n) contains all languages that can be decided in $O(n)$ time, we have $A \in \text{TIME}(n)$.

7.1.3 Complexity Relationships among Models

The time complexity relationships for the three models:

- the single-tape Turing machine;
- the multitape Turing machine;
- the nondeterministic Turing machine.

Theorem 7.8: Let $t(n)$ be a function , where $t(n) \geq n$. Then every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.

p. 282

Proof idea: Analyze the simulation in Theorem 3.13.

Theorem 3.13, p. 177

Proof: As stated on pp. 282–283.

Let N be a nondeterministic Turing machine that is a decider. The ***running time*** of N is the function $f : \mathcal{N} \to \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n.

Definition 7.9, p. 283

Fig. 7.10, p. 283

Theorem 7.11: Let $t(n)$ be a function , where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.

p. 284

Proof idea: Analyze the simulation in Theorem 3.16.

Theorem 3.16, p. 178

Proof: As stated on p. 284.

7.2 The Class P

Single-tape TM vs. Multitape TM:
At most a *polynomial* difference.

Deterministic TM vs. Nondeterministic TM:
At most an *exponential* difference.

7.2.1 Polynomial Time

Polynomial differences in running time are considered to be *small*, whereas exponential differences are considered to be *large*.

Try to think about n^3 and 2^n, where $n = 1000$.

p. 285

All reasonable deterministic computational models are ***polynomially equivalent***. We focus on aspects of time complexity theory that are unaffected by polynomial differences in running time.

Definition 7.12, p. 286

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

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

- Invariant for all polynomially equivalent models.
- Corresponding to realistically solvable problems.

7.2.2 Examples of Problems in P

Describe algorithms with numbered stages for analysis:

- A polynomial upper bound on the number of stages.
- The individual stages can be run in polynomial time.

Have to use reasonable encoding methods, such as base k notation for any $k \geq 2$ for integers.

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

Fig. 7.13, p. 287

Theorem 7.14: $PATH \in$ P.
Proof idea: Use a graph-searching method.
Proof: As stated on p. 288.

p. 288

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

Theorem 7.15: $RELPRIME \in$ P.
Proof idea: Use the ***Euclidean algorithm***.
Proof: As stated on p. 289.

p. 289

Theorem 7.16: Every context-free language is a member of P.
Proof idea: Use the CYK algorithm.
Proof: As stated on pp. 290–291.

p. 290

7.3 The Class NP

pp. 292–293

Polynomial verifiability

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

Fig. 7.17, p. 292

$$COMPOSITES = \{\langle x\rangle \mid x = pq, \text{ for integers, } p, q > 1\}$$

Definition 7.18, p. 293

A ***verifier*** for language A is an algorithm V, where

$$A = \{\, w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c \,\} \ .$$

We measure the time of a verifier only in terms of the length of w, so a ***polynomial time verifier*** runs on polynomial time in the length of w. A language A is ***polynomially verifiable*** if it has a polynomial time verifier. c is called a ***certificate*** or ***proof***.

Definition 7.19, p. 294

NP is the class of languages that have polynomial time verifiers. The term NP comes from ***nondeterministic polynomial time***.

p. 294

Theorem 7.20: A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.
Proof idea: Convert a polynomial time verifier to an equivalent polynomial time `NTM` and vice versa.
Proof: As stated on pp. 294–295.

NTIME($t(n)$) =

Definition 7.21, p. 295

$$\{L \mid L \text{ is a language decided by a } O(t(n)) \text{ time nondeterministic Turing machine.}\} .$$

Corollary 7.22: NP $= \bigcup_k \text{NTIME}(n^k)$.

p. 295

7.3.1 Examples of Problems in NP

$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$

Theorem 7.24: $CLIQUE \in$ NP.

p. 296

Proof idea: The clique is the certificate.
Proof & alternative proof: As stated on p. 296.

$SUBSET\text{-}SUM = \{\langle S, t\rangle \mid S = \{ x_1, \ldots, x_k \} \text{ and for some } \{ y_1, \ldots, y_\ell \} \subseteq \{ x_1, \ldots, x_k \}, \text{ we have } \Sigma y_i = t\}$

Note that $\{ x_1, \ldots, x_k \}$ and $\{ y_1, \ldots, y_\ell \}$ are considered ***multisets***.

Theorem 7.25: $SUBSET\text{-}SUM \in$ NP.

p. 297

Proof idea: The subset is the certificate.
Proof & alternative proof: As stated on p. 297.

coNP contains the languages that are complements of languages in NP, such as $\overline{CLIQUE}$ and $\overline{SUBSET\text{-}SUM}$. We don't know whether coNP is different from NP.

p. 297

7.3.2 The P versus NP Question

P = the class of languages for which membership can be *decided* quickly.
NP = the class of languages for which membership can be *verified* quickly.

Fig. 7.26, p. 298

Fig. 7.26 on p. 298 shows the two possibilities.

p. 298

For now, we can prove that

$$\text{NP} \subseteq \text{EXPTIME} = \bigcup_k \text{TIME}(2^{n^k}) \ ,$$

but we don't know whether NP is contained in a smaller deterministic time complexity class.

7.4 NP-completeness

p. 299

NP-complete problems and their importance.

NP-complete problem: ***satisfiability problem***.

$$SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$$

p. 300

Theorem 7.27:
$SAT \in$ P iff P = NP.
Proof: Later.

7.4.1 Polynomial Time Reducibility

A function $f : \Sigma^* \to \Sigma^*$ is a ***polynomial time computable function*** if some polynomial time Turing machine M exists that halts with just $f(w)$ on its tape, when started on any input w.

Definition 7.28, p. 300

Language A is ***polynomial time mapping reducible***, or simply ***polynomial time reducible***, to language B, written $A \leq_{\mathrm{P}} B$, if a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ exists, where for every w

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

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

Definition 7.29, p. 300

Theorem 7.31: If $A \leq_{\mathrm{P}} B$ and $B \in \mathrm{P}$, then $A \in \mathrm{P}$.
Proof: As stated on p. 301.

p. 301

Literal: x or $\overline{x}$
Clause: $(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$
Conjunctive normal form, cnf-formula:

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6})$$

3cnf-formula:

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6)$$

$$3SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$$

Theorem 7.32: *3SAT* is polynomial time reducible to *CLIQUE*.
Proof: As stated on pp. 302–303.

p. 302

7.4.2 Definition of NP-completeness

Definition 7.34, p. 304

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP;
2. every A in NP is polynomial time reducible to B.

p. 304

Theorem 7.35: If B is NP-complete and $B \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$.
Proof: This theorem follows directly from the definition of polynomial time reducibility.

p. 304

Theorem 7.36: If B is NP-complete and $B \leq_{\mathrm{P}} C$ for C in NP, then C is NP-complete.
Proof: As stated on p. 304.

7.4.3 The Cook-Levin Theorem

p. 304

Theorem 7.37: *SAT* is NP-complete.
Proof: As stated on pp. 305–310. An alternative proof appears in Section 9.3 on p. 379.

p. 310

Corollary 7.42: *3SAT* is NP-complete.
Proof: As stated on pp. 310–311.

7.5 Additional NP-complete Problems

p. 311

Corollary 7.43: *CLIQUE* is NP-complete.

7.5.1 The Vertex Cover Problem

$$VERTEX\text{-}COVER = \{\langle G, k\rangle \mid G \text{ is an undirected graph that has a } k\text{-node vertex cover}\}$$

Theorem 7.44: *VERTEX-COVER* is NP-complete. p. 312
Proof idea: Reduce *3SAT* to *VERTEX-COVER*.
Proof: As stated on pp. 312–313.

7.5.2 The Hamiltonian Path Problem

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

Theorem 7.46: *HAMPATH* is NP-complete. p. 314
Proof idea: Reduce *3SAT* to *HAMPATH*.
Proof: As stated on pp. 315–319.

Theorem 7.55: *UNHAMPATH* is NP-complete. p. 319
Proof idea: Reduce *HAMPATH* to *UNHAMPATH*.
Proof: As stated on p. 319.

7.5.3 The Subset Sum Problem

$$SUBSET\text{-}SUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\} \text{ and for some } \{y_1, \ldots, y_\ell\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \Sigma y_i = t\}$$

Theorem 7.56: *SUBSET-SUM* is NP-complete. p. 320
Proof idea: Reduce *3SAT* to *SUBSET-SUM*.
Proof: As stated on pp. 320–322.