Contents

Part I

Automata and Computation Machines

Chapter 1

Regular Languages

1.1 Finite automata

1.1.1 Formal definition

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

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

A finite automaton machine M **accepts** a string $w = w_1 \dots w_n$ with $w_i \in \Sigma$, when there exists states $r_0, \dots, r_n \in Q$ such that

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

The set of strings that M accept is the **language of machine** M, denoted by $L(M)$. We say that M **recognizes** $L(M)$.

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

1.1.2 Regular operations

Definition: Let A and B be languages. We define the regular operations **unioun**, **concatenation**, and **star** as follows:

Union: $A \cup B = \{x \mid x \in A \vee x \in B\}$.

Concatenation: $A \circ B = \{xy \mid x \in A, y \in B\}$.

Star: $A^* = \{x_1 x_2 \dots x_k \mid k \geq 0, x_i \in A\}$.

Theorem 1.1. *The class of regular languages is closed under regular operations.*

1.2 Nondeterminstic finite automaton

1.2.1 Formal definition

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

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

where $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$.

A nondeterminstic finite automaton machine M **accepts** a string $w = w_1 \dots w_n$ with $w_i \in \Sigma_\epsilon$, when there exists states $r_0, \dots, r_n \in Q$ such that

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

1.2.2 Equivalence of NFA and DFA

Two machine are **equivalent** if they recognize the same language.

Theorem 1.2. *Every nondeterminstic finite automaton has an equivalent deterministic finite automaton.*

Proof. Define $E(R)$ the **epsilon neighborhood of** $R \subset Q$ to be

$$E(R) = \{q \,|\, q \in \delta(r, \epsilon^n) \text{ forsome } r \in R, n \geq 0\}$$ ■

Corollary 1.3. *A language is regular language if and only if some nondeterminstic finite automaton recognizes it.*

Proof. Add. ■

1.2.3 Closure under regular operations

Theorem 1.4. *The class of regular languages is closed under union operation.*

Proof. Add. ■

Theorem 1.5. *The class of regular languages is closed under concatenation operation.*

Proof. Add. ■

Theorem 1.6. *The class of regular languages is closed under star operation operation.*

Proof. Add. ■

1.3 Regular expression

1.3.1 Formal defintion

Definition: R is a **regular expression** if R is

1. $\{a\}$ for some $a \in \Sigma$,
2. $\{\epsilon\}$,
3. $\emptyset$,
4. $R_1 \cup R_2$, where R_1 and R_2 are regular expressions,
5. $R_1 \circ R_2$, where R_1 and R_2 are regular expressions, or
6. R_1^*, where R_1 is a regular expressions,

For convenience, we let R^+ be shorthance for RR^*. i.e. $R^* = R^+ \cup \{\epsilon\}$.

$$\emptyset^* = \{\epsilon\} \qquad R \cup \emptyset = R$$
$$R \circ \emptyset = \emptyset \qquad R \circ \{\epsilon\} = R$$

1.3.2 Equivalence with DFA

Theorem 1.7. *A language is regular if and only if some regular expression describes it.*

Lemma 1.8. *If a language is described by a regular expression, then it is regular.*

Lemma 1.9. *If a language is regular, then it is described by a regular expression.*

Definition: A **generalized nondeterminstic finite automaton**, GNFA for short, is a 5-tuple, $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$ where

1. Q is a finite set called **states**.
2. Σ is a finite set called the **alphabet**.
3. $\delta : Q - \{q_{\text{accept}}\} \times Q - \{q_{\text{start}}\} \to \mathcal{R}$ where $\mathcal{R}$ is the set of regular expressions, is the **transition function**.
4. $q_{\text{start}} \in Q$ is the **start state**.
5. $q_{\text{accept}} \in Q$ is the **accept states**.

A generalized nondeterminstic finite automaton machine M **accepts** a string $w = w_1 \dots w_n$ with $w_i \in \Sigma^*$, when there exists states $r_0, \dots, r_n \in Q$ such that

1. $r_0 = q_{\text{start}}$.
2. $w_i \in L(\delta(r_{i-1}, r_i))$ for $i = 0, \dots, n-1$.
3. $r_n = q_{\text{accept}}$.

Proof. Convert DFA into a GNFA and then reduce that GNFA to a two state GNFA. □

1.4 Nonregular languages

Theorem 1.10 (Pumping lemma). *If A is a regular language, then there is a number p where if s is any string in A of length at least p, then s may be divided into three piece, $s = xyz$, satisfying the following conditions:*

1. $|y| > 0$.
2. $|xy| \geq p$.
3. $xy^n z \in A$ *for all* $n \geq 0$.

Proof. Add. ■

Chapter 2

Context Free Languages

2.1 Context free languages

2.1.1 Formal definition

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

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

We say u derives v, denoted by $u \overset{*}{\Rightarrow} v$, if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ such that

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

The language of the grammar is $\left\{ w \in \Sigma^* \,\middle|\, S \overset{*}{\Rightarrow} w \right\}$.

2.1.2 Designing CFL

Suppose G_i for $i = 1, \ldots, k$ are context free grammars with start state S_i. Then, the union of these grammar G can be obtained by the following rule

$$S \to S_1 | S_2 | \ldots | S_k$$

where S is the start state of G.

Moreover we can construct a context free grammar equivalent of a DFA. For each state q_i, consider a variable R_i. Then, $R_i \to aR_j$ if $\delta(q_i, a) = q_j$.

2.1.3 Ambguity

A string w is derived **ambiguously** in a context free grammar G if it has two or more distince leftmost derivation. Grammar G is **ambiguous** if it generate some strings ambiguously. In leftmost derivation, at each step the left most variable is replace.

Some grammars can only be generated by ambiguous CFLs. These are said to be inherently ambiguous.

2.1.4 Chomsky normal formal

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

$$A \to BC \qquad\qquad A \to a$$

where a is any terminal and A, B, and C are any terminals – except that B and C may not be the start state. In addition, we permit $S \to \epsilon$.

Theorem 2.1. *Any context free language is generated by a context free grammar in Chomsky normal form.*

Proof. Let u and v be any strings of terminals and variables.

- Add a new start variable.
- For ϵ-rules, such as $A \to \epsilon$, remove the rule and replace any $R \to uAv$ with $R \to uv$. If $R \to A$, then add $R \to \epsilon$ unless it had been previously removed.
- For unit rules, such as $A \to B$, remove the rule and replace any $B \to u$ with $A \to u$ unless this was a unit rule previously removed.
- Lastly, consider $A \to u_1 \dots u_k$ where u_i are either a variable or a terminal. We can replace this rule with the following

$$\begin{aligned} A &\to U_1 A_1 \\ A_i &\to U_{i+1} A_{i+1} \qquad\qquad i = 1, \dots, k-3 \\ A_{k-2} &\to U_{k-1} U_k \end{aligned}$$

  where $U_i = u_i$ if u_i is a variable, otherwise we must add $U_i \to u_i$. ■

2.2 Pushdown automata

A pushdown automaton is a nondeterminstic finite automaton with a stack.

2.2.1 Formal definition

Definition: A pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where

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

It accepts $w = w_1 \dots w_n$ where $w_i \in \Sigma_\epsilon$ if there exists a sequence of states $r_0, \dots, r_n \in Q$ and string $s_0, \dots, s_n \in \Gamma^*$ exists that satisfy the following

1. $r_0 = q_0$ and $s_0 = \epsilon$.
2. $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$ where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\epsilon$ and $t \in \Gamma^*$.
3. $r_n \in F$.

2.2.2 Equivalence with context free grammars

Theorem 2.2. *A language is context free if and only if some pushdown automaton recognizes it.*

Corollary 2.3. *Every regular language is context free.*

2.3 Non-context free languages

Theorem 2.4 (Pumping lemma for CFL). *If G is a context free language, then there is a number p where if s is any string in G of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying*

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

2.4 Deterministic context free languages

Definition: A nondeterminstic pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where

1. Q is a finite set of states.
2. Σ is a fintie set of input alphabet.
3. Γ is a finite set of stack alphabet.
4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to Q \times \Gamma_\epsilon \cup \{\emptyset\}$ is the transition function.
5. $q_0 \in Q$ is the start state.
6. $F \subset Q$ is the set of accept states.

and for all $q \in Q$, $a \in \Sigma$, and $x \in \Gamma$ exactly one the

$$\delta(q, a, x) \quad \delta(q, \epsilon, x) \quad \delta(q, a, \epsilon) \quad \delta(q, \epsilon, \epsilon)$$

is not $\emptyset$.

Lemma 2.5. *Every DPDA has an equivalent DPDA that always reads the entire string.*

2.4.1 Closure properties

Theorem 2.6. *The class of DFCL is closed under complementation.*

Let $A \dashv= \{w \dashv \mid w \in A\}$.

Theorem 2.7. *A is DFCL if and only if $A \dashv$ is DCFL.*

Chapter 3

Church-Turing Thesis

3.1 Turing machine

3.1.1 Formal definition

Definition: A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ where Q, Σ, Γ are all finite sets

1. Q is the set of states.
2. Σ is the input alphabet not containing the blank symbol Δ.
3. Γ is the tape alphabet, where $\Delta \in \Gamma$ and $\Sigma \subset \Gamma$.
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. $q_{acc} \in Q$ is the accept state.
7. $q_{rej} \in Q$ is the reject state and $q_{rej} \neq q_{acc}$.

Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ receives input $w = w_1 \dots w_n \in \Sigma^*$ on the leftmost n squares of the tape. And initially the rest of the tape is blank. The content tape together with the current state and head's position is called the **configuration**. A configuration may be denoted as uqv where uv is the content of the tape, q is the current state, and the head is on the first symbol of v. For example, q_0w is the start configuration and q_{acc} and q_{rej} are the accepting and rejecting configuration respectively. The accepting configuration and rejecting configuration are called the **halting configuration**.

Suppose C_1 and C_2 are two configuration. C_1 **yields** C_2 if the Turing machine can legally go from C_1 to C_2 in a single step.

A Turing machine accepts input w if a sequence of configuration $C_1, \dots, C_k$ 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.

The collection of strings that M accepts is the language of M.

Definition: A language is **Turing recognizable** or **recursively enumeratable** if some Turing machine recognizes it.

Deciders are Turing machines that halt on all iputs. A decider that recognizes some language is also said to decide that language.

A language is **Turing decidable** or **recursive** if some Turing machine decides it.

3.1.2 Examples

3.2 Variants of Turing machine

3.2.1 Stay put

Any Turing machine with stay put order S can be emulated by a Turing machine without.

3.2.2 Multipath Turing machine

each tape has its own head for reading and writing.

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

Theorem 3.1. *Every Multipath Turing machine has an equivalent single Turing machine.*

Corollary 3.2. *A language is Turing recognizable if and only if some multipath Turing machine recognizes it.*

3.2.3 Nondeterministic Turing machine

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

Theorem 3.3. *Every nondeterministic Turing machine has an equivalent determinstic Turing machine.*

Corollary 3.4. *A language is Turing recognizable if and only if some nondeterministic Turing machine recognizes it.*

3.2.4 Enumerators

Theorem 3.5. *Every enumerator has an equivalent determinstic Turing machine.*

Corollary 3.6. *A language is Turing recognizable if and only if some enumerator recognizes it.*

3.3 The definition of algorithm

3.3.1 Hilber's 10_{th} problem

Provide an algorithm for determining if a polynomial has an integral root. This is equivalent to

$$D = \{p \,|\, p \text{ is a polynomial with an integral root}\}$$

being decidable which is not. Although, it is Turing recognizable.

Part II

Computability Theory

Chapter 4

Lecture Notes

4.1 Computation

Abstractly, a *computation* is the "work done" by a *computation machine*. For example, we can consider our brains as computation machines and whatever they do is considered a computation. A computation machine M can be viewed as a black box that takes an input w and acts on the input in a step by a step manner. Suppose a brain is tasked with calculating a formula. This formula is the input and at each step the brain may calculate or simplify a certain part of the formula.

A *configuration* is a description of the internals and the state of the machine. Note that the state of the machine may be described in many ways, however, we usually fix a method of describing so that the resulting configurations are unique. A configuration of a brain computing a formula might be the state of neurons in the brain and the steps of the calculation as written in a piece of paper.

In each unit of time the configuration of the machine changes based on its *configuration graph*. This is like saying that in each time unit the brain does an algebraic step such as addition or multiplication and then changes the content of the paper to the new intermediate result. This graph is a directed graph with all the possible configurations of M as nodes. There is an edges from configuration C to C' if the machine go from C to C' in one unit of time. In our case, the brain can go from C to C' if the corresponding formula of C' can be arrived by doing one algebraic step from the corresponding formula of C. This graph might be infinite, and in fact, finite configuration graphs can be dealt with tools from graph theory and thus do not provide a rich theory of computation.

In this way, we can formalize the notion of computation machines as a discrete dynamical system.

Definition: A discrete dynamical system is a system whose state varies with a discrete variable, time, according to a directed graph $G = (\mathcal{V}, \mathcal{E})$ such that $\mathcal{V}$, the set of states of the system, is a discrete set. The set $\mathcal{E}$ determines how the state of the system varies with time. Let $V_t \in \mathcal{V}$ be the state of the system at time t, then the state of the system a unit of time later, V_{t+1}, must be a neighbor of V_t. That is,

$$V_{t+1} \in N(V_t) = \{V \in \mathcal{V} \,|\, (V_t, V) \in \mathcal{E}\}$$

Remark 1. In general, $N(V_t)$ might have zero, one, or more than one states. When $N(V_t)$ then the system can not progess and it stops. When V_t has exactly one neighbouring state, the next state is uniquely determined. However, when it has more than one neighbouring

states, then the next state is selected non-determinstically. Such selections are not based on some probability distribution or any other rules.

Definition: A **computation machine** is a discrete dynamical system $M = (\mathcal{C}, \mathcal{E})$. The state $C \in \mathcal{C}$ is called a **configuration** of the machine and $E \in \mathcal{E}$ is called a **transition**. On an input w, the machine will be placed in an *initial configuration* $C_0^w \in \mathcal{C}$. At each unit of time, a neighbouring configuration is chosen if it exists. If no neighbouring configurations exists, then the machine stops. A computation machine M is deterministic for any $C \in \mathcal{C}$ there is at most one neighbour.

The possible configurations that may be attained by an input w, form a tree. If the computation is deterministic, then the tree is a path.

Remark 2. In computation, we usually distinguish between halting and crashing. Loosely defined, halting is an expected stopping and crashing is an unexpected stopping. To distinguish between these two scenarios, we may consider $\mathcal{C}_h \subset \mathcal{C}$ as the set of halting configuration and add it to the description of M. Therefore, if a non-halting configuration has no neighbour, the computation machine crahses.

When viewing our current definition of the computation machines as computer programs, we see certain similarities. For example, a computer program is a set of variables and instructions. At each step, an instruction is executed which will change the value of the some the variables– e.g. the instruction pointer. The input of the program is considered as one of the variables. Then, the values of the variables is the configuration of the program. The instruction are the transition between different configurations.

However, there is a major difference between computer programs and computation machines. Computer programs are finite, however, computation machines are not. As stated above, the set of configurations of any useful machine is infinite. As a result, the set of transitions is also infinite. On the other hand, a practical theory of computation requires a certain finite limitation. This finite limitation is placed on the description of the computation machine. That is, there exists a finite set symbols such that the computation machines can be described by a finite string of those symbols.

Therefore, if we can describe a machine by a string, it makes sense to assume that its configurations and transitions correspond to some strings. As a result, a computation machine can also be viewed as a string processor. We want our string processor to have a finite description.

A string processor's configurations are strings and its transition describe how these strings change. There are infinitely many possible strings. Thus, the challange is to give the correct result/appropriate action for infinitely many input by a finitely describable entity. Hence, finite description is the same as saying that configuration graph can be constructed by finitely many rules. Finitely many rules implies that configurations change locally. Local property is necessary for finite description. Basic operations are changes in bounded number of memory.

4.1.1 Problems

A problem is a query on some given data and constants. To answer the query by a computation machine, we need to encode the data of the problem into some string. This string is then given to the computation machine. The computation machine takes into consideration the constants and query of the problem so that on any valid input it returns the answer to the query.

The coding of the problem must not have any pre-processing. We enforce this by making the coding essentially trivial. The reason being that different coding may result in better or faster computation.

4.2 Formal Languages

- Suppose we want to compute a function $f : \Sigma^* \to \Pi^*$ with yes/no answer. That is, $\forall w \in \Sigma^*,\ f(w) \in \{0, 1\}$. Then, computing $f(w)$ is equivalent to determining $w \in L_f$ where $L_f = \{w \mid f(w) = 1\}$.
- Generally a problem type is described by a triple of constants, given data, and a query. Yes/no problems have queries which are either true or false.
- To have a well-defined yes/no problem, the set L must have a finite description.

4.2.1 Finite description of formal languages

1. Logical: $L = \{x \mid P(x)\}$ for some proposition P on strings.
2. Grammar: $L = \langle A, \tau_1, \ldots, \tau_n \rangle$ where A is a set of axioms and τ_i are the rules that allow us to combine the axoim, inference rules.
3. Computation Machine: $L =$ is the language of a computation machine, a program in a compiler.
4. Enumerator: $L =$ is the language of a an enumerator, a generator in a compiler.
5. Formal Game: $L =$.?
6. Transducer?

In general, the classes of languages that can be described by several of the above's methods are interesting.

A definition is given by specifying a type, the data, and the properties.

4.3 Computation Machines

Types of machines:

- Deterministic vs Nondetermninistic. Technically, any deterministic machine is also non-determninistic.

Machines may have some halting configurations. Based on that we can convene the certain halting configuration are accepting configuration and others are rejecting configuration. Based on that we can define certain languages for the machines, for example, the input that are accepted. Then, we have two further classifications.

- Acceptors vs Deciders.

Therefore, a computation is described by its finite description and its standard language. Deciders are closed under inversion of output.

The main problem in theory of computation is determining the relationships of the sub-classes of a model. For example, whether DetDec are a proper subset of DetAcc or are they equal.

4.3.1 Finite automaton

4.3.2 Pushdown automaton

4.3.3 Turing machine

4.3.4 Enumerators

4.4 Complexity of the languages

We want to measure the complexity of the problems wrt some model. One way is to reduce the problems to each other.

many-to-one reduction. B must be serializable. hardness and completeness. isomorphism conjecture. Berman-Hartmanis conjecture

4.5 Gap problems

Gap problems between NATM and DATM: No. Gap problems between DATM and DDTM: No. Gap problems between NATM and DDTM: Yes. Gödel's theorem.

Halting problem. Complete problem.

1. Fix a coding for a TMs in $\{0, 1\}$.
2. It is like numbering them.
3. Determining if a number is TM are not is a simple syntax checking.
4. Construct a language that does not have a DTM. Done by the Cantor's diagonalization.
5. If lucky, the language has a acceptor.

4.6 Constraining Turing machines

4.6.1 Time constraints

redefine the TM as binary and single taped. define a the TIME as the length of the computation. Define $f(n)$-boundedness of a machine. Now limit by choosing f.

Write the Gödel's proof for P vs NP.

$NP \subset DDTM$.

Combination of two poly TM is another poly TM.

NP is like having a poly certifier.

4.6.2 Space constraints

redefine the TM as 3-taped: input RO, worktap RW, output WO. Argue that this does not affect our computation capability and results in the same class of languages (almost).

LBAs and the context sensitive grammers.