CSCI3130 Formal Languages and Automata Theory

Ryan Chan

December 2, 2025

Abstract

This is a note for **CSCI3130 Formal Languages and Automata Theory**.
Contents are adapted from the lecture notes of CSCI3130, prepared by Tsung-Yi Ho, as well as some online resources.
This note is intended solely as a study aid. While I have done my best to ensure the accuracy of the content, I do not take responsibility for any errors or inaccuracies that may be present. Please use the material thoughtfully and at your own discretion.
If you believe any part of this content infringes on copyright, feel free to contact me, and I will address it promptly.
Mistakes might be found. So please feel free to point out any mistakes.

Contents

Chapter 1

Introduction

1.1 Theory of Computation

In the theory of computation, we study the following:

1. **Formal Languages**

This is the abstraction of the general characteristics of programming languages. It consists of a set of symbols and some rules, i.e. strings and grammar of formation, and these symbols are then combined into sentences.

2. **Automata Theory**

This is the study of the dynamic behaviours of "discrete-parameter information systems" in the form of "abstract computing devices."

Types of automata are distinguished by their temporary memory.

For finite automata, there is no temporary memory. Pushdown automata use a stack, and Turing machines use random-access memory. The computational power of finite automata is the smallest, while Turing machines have the highest, with pushdown automata in between.

There are three major models of automata:

- generator: with output only

- acceptor: with input only

- transducer: with both input and output

3. **Computability**

This is the study of the problem-solving capabilities of computational models.

We can classify problems based on resources:

- Impossible problems

- Possible with unlimited resources but impossible with limited resources

- Possible-with-limited-resources problems

Or by time:

- Undecidable problems

- Intractable problems

- Tractable problems

4. **Computational Complexity**

This is the study of the efficiency of problem-solving. To unify comparison, we use an abstract model for problem execution. A Turing machine is usually used, since although simple, it has been proved to be able to simulate any problem-solving steps designed by human beings.

Problems can be classified into:

- P: Polynomial-time problems. These are problems that can be solved quickly (in polynomial time) by a normal computer.

- NP: Non-deterministic Polynomial time. These are problems where we do not know how to solve them quickly, but if someone gives us a solution, we can verify it quickly (in polynomial time).

- NP-hard: at least as hard as every NP problem.

- NP-complete: both NP and NP-hard.

1.2 Three Basic Concepts

1.2.1 Language

A language is a set of strings, where a string is a sequence of symbols from the alphabet. For example, for the alphabet $\Sigma = \{a, b\}$ we have $\{a, ab, abba, \dots\}$.

There are several operations we can apply to strings. Consider

$$w = a_1 a_2 \cdots a_n, \quad v = b_1 b_2 \cdots b_m.$$

1. concatenation

We can concatenate strings, e.g. $wv = a_1 a_2 \cdots a_n b_1 b_2 \cdots b_m$.

2. reverse

We can reverse a string, denoted as $w^R = a_n \cdots a_2 a_1$.

Denote the string length as $|w|$, where the length of concatenation is $|uv| = |u| + |v|$.

We denote a string with no letters by λ. Therefore, $\lambda w = w\lambda = w$.

A substring of a string is a subsequence of **consecutive** characters.

In $w = uv$, we say u is the prefix and v is the suffix.

We write $w^n = \underbrace{ww \cdots w}_{n}$, where $w^0 = \lambda$.

We write $\Sigma^\star$ to represent the set of all possible strings from alphabet Σ, including λ, and Σ^+ to represent the set of all possible strings from Σ except λ.

A language is any subset of $\Sigma^\star$.

> **Note.** Note that $\varnothing = \{\} \neq \{\lambda\}$.

A language can be infinite, e.g. $L = \{a^n b^n : n \geq 0\}$.

For operations on languages, we adopt the usual set operations. For complement, we have $\overline{L} = \Sigma^\star - L$.

We define reverse as $L^R = \{w^R : w \in L\}$. For example, for $L = \{a^n b^n : n \geq 0\}$, we have $L^R = \{b^n a^n : n \geq 0\}$.

We define concatenation of languages as

$$L_1 L_2 = \{xy : x \in L_1,\, y \in L_2\}.$$

We write $L^n = \underbrace{LL \cdots L}_{n}$, where $L^0 = \{\lambda\}$.

Kleene closure is defined as

$$L^\star = L^0 \cup L^1 \cup L^2 \cup \cdots.$$

For example, we have

$$\{a, bb\}^\star = \begin{cases} \lambda, \\ a, bb, \\ aa, abb, bba, bbbb, \\ \cdots \end{cases}$$

Positive closure is defined as

$$L^+ = L^1 \cup L^2 \cup \cdots .$$

For example, we have

$$\{a, bb\}^+ = \begin{cases} a, bb, \\ aa, abb, bba, bbbb, \\ \cdots \end{cases}$$

1.2.2 Grammar

Language can be described as a system of symbols, and the grammar is the rule by which the symbols are manipulated.

Definition 1.2.1 (Grammar). A grammar G is defined as a 4-tuple

$$G = (V, T, S, P)$$

where

- V is a finite set of variables,
- T is a finite set of terminals,
- $S \in V$ is the start variable,
- P is a finite set of production rules.

Here, the production rules $x \to y$ specify how the grammar transforms one string into another. If γ can be derived from α in one step, we write $\alpha \Rightarrow \gamma$. If it can be derived using zero or more steps, we write $\alpha \overset{\star}{\Rightarrow} \gamma$.

Definition 1.2.2. Let $G = (V, T, S, P)$ be a grammar. Then the set

$$L(G) = \{w \in T^\star : S \overset{\star}{\Rightarrow} w\}$$

is the language generated by G.

If $w \in L(G)$, then the sequence $S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n \Rightarrow w$ is a derivation of the sentence w, where $S, w_1, \cdots, w_n$ are called sentential forms.

Two grammars G_1, G_2 are said to be equivalent if they generate the same language, i.e. $L(G_1) = L(G_2)$.

1.2.3 Automata

An automaton is an abstract model of a digital computer. There are input, a control unit, storage, and output.

There are deterministic and nondeterministic automata. Also, as introduced before, we have acceptors and transducers. More specifically, an automaton whose output is "yes" or "no" is an acceptor, while one whose output is a string of symbols is a transducer.

We will look at the details in the next chapter.

Chapter 2

Finite Acceptor

Before diving into the details, we take a look at the transition graph, which is an important component in learning the concepts that follow.

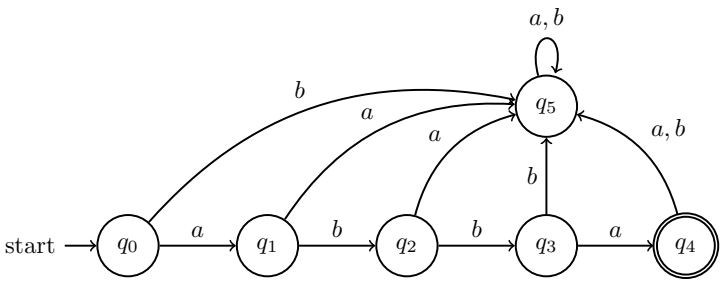

Figure 2.1: Transition Graph

For a transition graph, there is an initial state, and the internal transitions are represented by arrows, where the labels on top of them indicate valid transitions. There is also a final (accepting) state, which is a node drawn with a double circle, as shown above for q_4.

2.1 Deterministic Finite Acceptor

A Deterministic Finite Acceptor is a machine with a finite number of states; some states are accepting while others are rejecting. It reads the input string one character at a time.

After reading a character, it moves to another state depending on the current state and the input read. After reading the entire string, if the DFA is in an accepting state, the input string is accepted; otherwise, it is rejected.

Definition 2.1.1 (Deterministic Finite Acceptor (DFA)). A Deterministic Finite Acceptor is defined by the 5-tuple

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

where

- Q: a finite set of internal states,
- Σ: a finite set of symbols called the input alphabet,
- δ: $Q \times \Sigma \to Q$, called the transition function (a total function),
- q_0: $q_0 \in Q$ is the initial state,
- F: $F \subseteq Q$ is the set of final states.

From Figure 2.1, we have $\Sigma = \{a, b\}$, $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$, where q_0 is the initial state and q_4 is the

final state. The transition function is

$$\delta(q_0, a) = q_1, \quad \delta(q_0, b) = q_5, \cdots$$

We also have the extended transition function $\delta^\star$

$$\delta^\star : Q \times \Sigma^\star \to Q,$$

for example,

$$\delta^\star(q_0, ab) = q_2.$$

That being said, if we have the string $abba$ as input, it will be accepted by this DFA since it reaches the final state and outputs "accept".

Here we see that there is a walk from $q \to q'$ with label w iff. $\delta^\star(q, w) = q'$, where $w = \sigma_1\sigma_2\cdots\sigma_k$.

We also allow recursion:

$$\delta^\star(q, w\sigma) = \delta(\delta^\star(q, w), \sigma).$$

Definition 2.1.2. For a DFA M, the language $L(M)$ contains all input strings accepted by M, where

$$L(M) = \{w \in \Sigma^\star : \delta^\star(q_0, w) \in F\}.$$

Then it is intuitive that

$$\overline{L(M)} = \{w \in \Sigma^\star : \delta^\star(q_0, w) \notin F\}$$

is the language rejected by M.

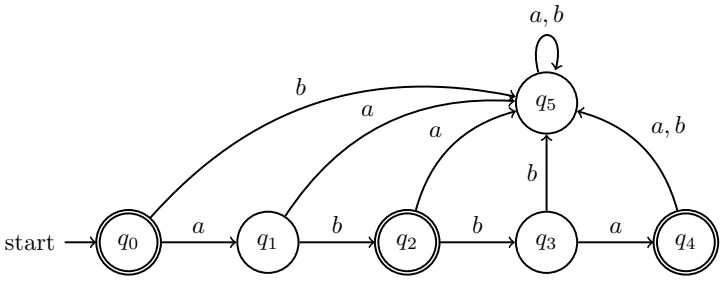

Figure 2.2: Transition Graph

For example, in the above DFA M, $L(M) = \{\lambda, ab, abba\}$.

Definition 2.1.3 (Regular Language)**.** A language L is regular iff. there exists some DFA M such that $L = L(M)$. The language $L(M)$ contains all input string accepted by a DFA M.

This means that there are also languages that are not regular, since no DFA can accept such languages.

2.2 Nondeterministic Finite Acceptor

Many deterministic algorithms require that one make a choice at some stage. Thus, we introduce nondeterminism, which is sometimes helpful in solving problems easily. It is also an effective mechanism for describing some complicated languages concisely.

Definition 2.2.1 (Nondeterministic Finite Acceptor (NFA))**.** A Nondeterministic Finite Acceptor is defined by the 5-tuple

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

where

- Q: a finite set of internal states,

- Σ: a finite set of symbols called the input alphabet,

- δ: $Q \times (\Sigma \cup \{\lambda\}) \to 2^Q$, called the transition function (a total function),
- q_0: $q_0 \in Q$ is the initial state,
- F: $F \subseteq Q$ is the set of final states.

The main difference between an NFA and a DFA is that in an NFA, the range of $\delta(q, \sigma)$ is in the powerset 2^Q. It allows λ as the second argument of δ, and the set $\delta(q_i, a)$ may be empty.

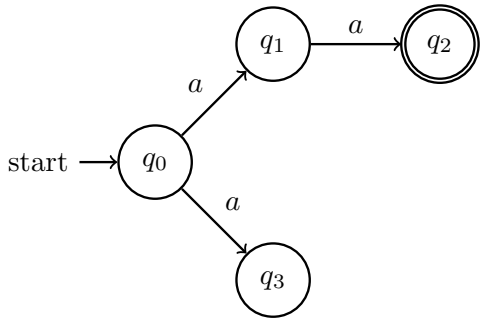

Figure 2.3: Transition Graph

For example, in Figure 2.3, when the input symbol is a, there are two choices. If we have the input string aa, all input is consumed and accepted in the upper route, while the automaton hangs in the lower route.

We say that an NFA accepts a string when there exists a computation of the NFA that accepts the string, i.e., all the input is consumed and the automaton is in a final state.

An NFA rejects a string when there is no computation of the NFA that accepts the string, i.e., either all input is consumed and the automaton is not in a final state, or the input cannot be consumed.

Therefore, in the above example, aa is accepted.

In an NFA, we also allow λ-transitions, as shown below.

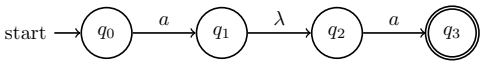

Figure 2.4: Lambda transition

Note that the λ symbol will not appear on the input tape. We can also express languages more easily using an NFA than a DFA.

Definition 2.2.2. For an NFA M, the language $L(M)$ contains all input strings accepted by M, where

$$L(M) = \{w \in \Sigma^\star : \delta^\star(q_0, w) \cap F \neq \varnothing\}.$$

Consider the following: we can have $\delta(q_1, 0) = \{q_0, q_2\}, \delta(q_0, \lambda) = \{q_0, q_2\}, \cdots$.

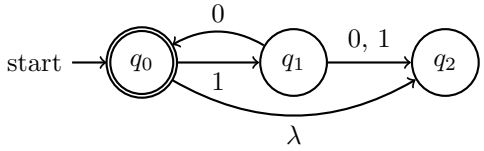

Figure 2.5: Transition Graph

The extended transition function $\delta^\star$ is the same as in the case of a DFA.

Again, there is a walk from $q_i \to q_j$ with label w iff. $q_j \in \delta^\star(q_i, w)$.

2.3 Equivalence of DFA and NFA

Definition 2.3.1. Two finite acceptors M_1, M_2 are said to be equivalent if

$$L(M_1) = L(M_2),$$

i.e., both accept the same language.

DFA and NFA have the same computational power, since a DFA is just a restricted kind of NFA. One operation we can perform is the conversion from an NFA to a DFA.

Given an NFA M, we want to convert it to an equivalent DFA M'. If the NFA has states $q_0, q_1, \cdots$, then the DFA will have states in the powerset of the NFA states, i.e., $\{\varnothing, \{q_0\}, \{q_1\}, \cdots\}$.

To do the conversion, we follow the following procedure:

1. Convert the initial state of the NFA q_0 to the initial state of the DFA $\{q_0\}$.

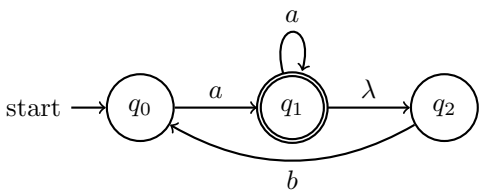

Figure 2.6: NFA

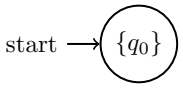

Figure 2.7: DFA - 1

2. For every DFA state, compute the transition function in the NFA, then add the corresponding transition to the DFA. Repeat this step until no more transitions can be added.

3. For any state in the DFA that contains a final NFA state, mark it as a final state in the DFA.

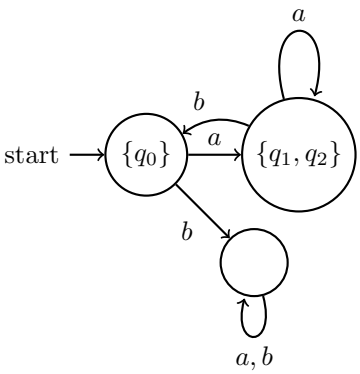

Figure 2.8: DFA - 2

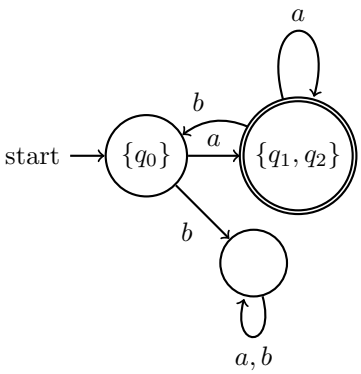

Figure 2.9: DFA - 3

As previously mentioned, the language accepted by a DFA is regular, and this also applies to an NFA. To prove this, we show the following:

1. Every DFA is trivially an NFA, i.e., any language accepted by a DFA is also accepted by an NFA.

2. Any NFA can be converted to an equivalent DFA, i.e., any language accepted by an NFA is also accepted by a DFA.

Theorem 2.3.1. For any NFA M, there exists a DFA M' constructed via the subset construction such that

$$L(M) = L(M').$$

Proof. Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA. We construct a DFA $M' = (Q', \Sigma, \delta', q_0', F')$ using the subset construction:

$$Q' = \mathcal{P}(Q), \quad q_0' = \{q_0\}, \quad F' = \{S \subseteq Q \mid S \cap F \neq \emptyset\},$$

$$\delta'(S, a) = \bigcup_{q \in S} \delta(q, a) \quad \text{for } S \subseteq Q, a \in \Sigma.$$

We want to prove that for all strings $w \in \Sigma^*$,

$$\hat{\delta}'(q_0', w) = \hat{\delta}(q_0, w),$$

where $\hat{\delta}$ and $\hat{\delta}'$ are the extended transition functions for M and M' respectively.

Base Case: $|w| = 0$, i.e., $w = \epsilon$.

$$\hat{\delta}'(q_0', \epsilon) = q_0' = \{q_0\} = \hat{\delta}(q_0, \epsilon).$$

Inductive Step: Assume that for a string $x \in \Sigma^*$ of length k,

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

Now consider a string $w = xa$ where $a \in \Sigma$. Then

$$\begin{aligned}
\hat{\delta}'(q_0', xa) &= \delta'(\hat{\delta}'(q_0', x), a) \\
&= \delta'(\hat{\delta}(q_0, x), a) \quad \text{(by inductive hypothesis)} \\
&= \bigcup_{q \in \hat{\delta}(q_0, x)} \delta(q, a) \\
&= \hat{\delta}(q_0, xa).
\end{aligned}$$

By induction, for all $w \in \Sigma^*$,

$$\hat{\delta}'(q_0', w) = \hat{\delta}(q_0, w).$$

Finally, a string w is accepted by M if $\hat{\delta}(q_0, w) \cap F \neq \emptyset$, and w is accepted by M' if $\hat{\delta}'(q_0', w) \in F'$. Since

$$\hat{\delta}'(q_0', w) = \hat{\delta}(q_0, w),$$

we have

$$w \in L(M) \Leftrightarrow w \in L(M').$$

Hence, $L(M) = L(M')$. ■

Therefore, it suffices to show that an NFA accepts a regular language.

Also, any NFA can be converted into an equivalent NFA with a single final state by using λ-transitions.

Chapter 3

Regular Language

Chapter 4

Context-Free Language

Chapter 5

Pushdown Automata

Chapter 6

Turing Machine