Algorithm Analysis and Automata

UT Knoxville, Spring 2023, COSC 312

Dr. Michael W. Berry, Alex Zhang

February 17, 2023

Contents

1 Introduction

Computer science is all about problem-solving, and as computer scientists, we have developed a method of abstracting problems into three key components: unknowns, data, and conditions. To further our understanding of this process, we have developed the Theory of Computation (TOC), which encompasses three main areas: Automata, Computability, and Complexity.

Automata are problem-solving devices that help us model and solve problems. Computability provides a framework for categorizing these devices based on their computing power, while Complexity measures the space complexity of the tools we use to solve problems.

When approaching problems, we think of the data as "words" in a given "alphabet", while conditions form a set of words known as a language. The unknowns in this equation are boolean values, which are true if a word is in the language and false if it is not.

Overall, the Theory of Computation provides a framework for understanding how we solve problems using computers. By breaking down problems into their component parts, we can use automata to model and solve them, and categorize these solutions based on their computing power and space complexity.

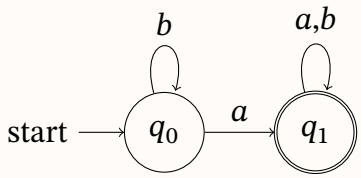
2 Finite Automata

Definition 2.0.1 ▸ Finite Automaton

A ***finite automaton*** is a theoretical device that is designed to recognize patterns in input from a set of characters. It operates through a finite set of states, including a start state and one or more final or accept states, and transitions between these states based on the input it receives.

In essence, a finite automaton is a simplified machine that helps to identify patterns within data. Finite automata can be used to generalize tons of applications, ranging from parsers for compilers, pattern recognition, speech processing, and market prediction.

Example 2.0.1 ▸ Simple Finite Automaton



Here, q_0 represents our start state, and q_1 represents our accept state. If we give the machine the character a, then it transitions to q_1 and is in an accept state. If we don't give it an a, then it will be stuck on q_0.

Definition 2.0.2 ▸ Deterministic Finite Automaton (DFA)

A ***deterministic finite automaton*** (DFA) is a type of finite automaton where every state transition corresponds to one and only one next state. In other words, a DFA is a finite-state machine that follows a single path of transitions for a given input.

We can formally define a DFA as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of all possible states
- Σ is a finite set of symbols called an ***alphabet***
- $\delta : Q \times \Sigma \to Q$ is a ***transition function***
- $q_0 \in Q$ is the ***start state***

- $F \subseteq Q$ is the set of ***accept states*** (or final states)

The determinism of a DFA ensures that, for any given state and input, there is a unique and well-defined next state. To explain the computation of a DFA, let's consider a simple example DFA called M_1. The process of running M_1 can be broken down into the following steps:

- M_1 begins at its initial state.
- We give M_1 a sequence of characters from its alphabet.
- With each input symbol, M_1 makes a state transition along the edge labeled with that symbol.
- When M_1 reads the last symbol, it outputs whether it's in an accept state

In essence, the DFA computation involves moving from state to state based on the input symbols until the final state is reached, at which point it outputs whether the input is accepted or rejected.

Definition 2.0.3 ▸ String

A ***string*** over an alphabet Σ is a (finite) sequence of symbols in Σ.

- ϵ represents the ***empty string***
- The length of a string w is denoted by $|w|$.
- The ***concatenation*** of w_1 and w_2 is a string that contains one copy of w_1 followed by one copy of w_2. We simply denote this by juxtaposition: $w_1 w_2$.

Definition 2.0.4 ▸ Acceptance

We say a DFA ***accepts*** a string if, after reading that string, the machine ends on an accept state. Otherwise, the machine rejects the string.

More formally, let $M := (Q, \Sigma, \delta, q_0, F)$ be a DFA and $w = a_1 \dots a_n$ be a string over Σ. We say M ***accepts*** w if there exists a sequence of states $r_0 \dots r_n$ in Q such that:

- $r_0 = q_0$
- $\delta(r_i, a_{i+1}) = r_{i+1}$
- $r_n \in F$

Definition 2.0.5 ▸ Language

A ***language*** is simply a set of strings. A language over an alphabet means every character of every string in the language is a character of the alphabet.

Technique 2.0.1 ▸ Constructing a DFA

Here's the general design approach for a DFA:

1. Identify the possible states of the finite automaton
2. Identify the condition to change from one state to another
3. Identify initial and final states
4. Add missing transitions

Given a finite automaton, we can deduce the language of possible inputs.

Example 2.0.2 ▸ Simple Finite Automaton

Let $M_1 := (Q, \Sigma, \delta, q_1, F)$ where $Q = \{q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$, and $F = \{q_2\}$. Define a possible transition function δ.

The transition function $\delta : Q \times \Sigma \to Q$ must map every ordered pair of a state and letter to another state.

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

Then, the language is:

$$L(M_1) = \{(w \in \Sigma^* : 1 \in w) \wedge (\text{has even number of 0s following the last 1})\}$$

Example 2.0.3 ▸ Simple Finite Automaton

Let $M_2 := (Q, \Sigma, \delta, q_1, F)$ where $Q = \{q_1, q_2\}$, $\Sigma = \{0, 1\}$, and $F = \{q_2\}$. Consider a possible transition function δ defined as such:

	0	1
q_1	q_1	q_2
q_2	q_1	q_2

This time, our language is much simpler:

$$L(M_2) = \{w \in \Sigma^* : w \text{ ends in a } 1\}$$

Now imagine if kept everything the same but made $F = \{q_1\}$. Because our finite automaton's initial state is q_1, we must now consider the possibility of the empty word. Then our language is:

$$L(M_2) = \{w \in \Sigma^* : w \text{ does not end in } 1\}$$

Example 2.0.4 ▸ Pattern Recognizing Finite Automaton

Let $M_3 := (Q, \Sigma, \delta, q_1, F)$ where $Q = \{s, q_1, q_2, r_1, r_2\}$, $\Sigma = \{a, b\}$, and $F = \{q_1, r_1\}$. Consider a possible transition function δ defined as such:

	0	1
s	q_1	r_1
q_1	q_1	q_2
q_2	q_1	q_2
r_1	r_2	r_1
r_2	r_2	r_1

Now our machine encompasses two smaller machines. s acts as a branch point where we must lock ourselves into either q states or r states.

Then our language is:

$$L(M_3) = \{w \in \Sigma^* : (w \text{ starts and ends with } a) \vee (w \text{ starts and ends with } b)\}$$

Now, let's start with a given language, then find an acceptable transition function δ.

Example 2.0.5 ▸ Finding δ from Language

Suppose $\Sigma = \{a, b\}$. Let F_1 be a finite automaton that recognizes the language $A_1 := \{w : w \text{ has at exactly two a's}\}$. Let F_2 be a finite automaton that recognizes the language $A_2 := \{w : w \text{ has at least two b's}\}$.

Finish example

3 Regular Languages

Definition 3.0.1 ▸ Regular Language

A language is ***regular*** if there exists a finite automaton that can accept every string from the language.

We can consider new languages derived from set operations on other languages, such as union and concatenation. There are also two special operations to consider: ***concatenation*** and ***Kleene closure***.

Definition 3.0.2 ▸ Concatenation

Let L_1 and L_2 be languages. The ***concatenation*** of L_1 and L_2 is defined as:

$$L_1 \circ L_2 \coloneqq \{xy : x \in L_1 \wedge y \in L_2\}$$

Definition 3.0.3 ▸ Set Power, Kleene Closure

Let L be a language, and let $n \in \mathbb{N}$. The ***set power*** operation on L is defined as:

$$\begin{aligned} L^n &\coloneqq \begin{cases} \{\epsilon\}, & \text{if } n = 0 \\ L^{n-1} \circ L, & \text{otherwise} \end{cases} \\ &= \underbrace{L \circ L \circ \cdots \circ L}_{n \text{ times}} \end{aligned}$$

The ***Kleene closure*** of L is defined as:

$$\begin{aligned} L^* &\coloneqq \bigcup_{i=0}^{\infty} L^i \\ &= L^0 \circ L^1 \circ L^2 \circ \cdots \end{aligned}$$

Theorem 3.0.1 ▸ Closure of Regular Languages

Class of regular languages is closed under intersection and closed under complementation.

Need proof here!

3.1 Nondeterminism

Designing a DFA that accepts a complex language can be a challenging task. However, to address this, we can introduce the concept of nondeterminism, which allows the machine to choose state transitions even without any input. By doing so, we create a more flexible automaton that can branch and handle complex languages. Specifically, we can create a nondeterministic finite automaton (NFA) that can recognize languages beyond the capability of a DFA.

Definition 3.1.1 ▸ Nondeterministic Finite Automaton (NFA)

A ***nondeterministic finite automaton*** is a finite automaton where each state transition can lead to an arbitrary number of states, chosen by the machine.

We can formally define an NFA as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states
- Σ is a finite set of symbols called an ***alphabet***
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$ is a ***transition function*** ($\mathcal{P}(Q)$ being the powerset of Q)
- $q_0 \in Q$ is the ***start state***
- $F \subseteq Q$ is the set of ***accept states***

In this context, ϵ represents a nondeterministic choice by the machine.

With a nondeterministic finite automaton (NFA), a state can have multiple transitions for a given input symbol, creating multiple paths the machine can follow. This makes it easier to design an NFA that accepts a more complex language, as the machine can explore different paths to find the correct one. However, this flexibility comes at a cost: NFAs are more difficult to analyze and simulate compared to DFAs.

Definition 3.1.2 ▸ Acceptance (NFA)

An NFA ***accepts*** a string if, after reading that string, the NFA ends on an accept state.

More formally, let $N := (Q, \Sigma, \delta, q_0, F)$ be an NFA and $w := y_1 \dots y_n$ be a string over $\Sigma \cup \{\epsilon\}$. We say N ***accepts*** w if there exists a sequence of states $r_0, \dots, r_m \in Q$ such that:

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

Theorem 3.1.1 ▶ Closure of Regular Languages

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

Proof sketch. □

draw proof sketch here

3.2 DFA/NFA Equivalence

An NFA can have multiple state transitions for a given input symbol, leading to ambiguity in the possible paths the machine can take. To overcome this ambiguity, we can "blow up" the set of possible states to its power set. This means creating a new state for every possible combination of states the NFA could be in at any given time.

Once we have this expanded set of states, we can create deterministic transitions between the subsets of states. This means that for every input symbol, there is only one possible state the machine can transition to. By doing this, we shift the ambiguity from the transitions between states to the states themselves. The result is a DFA that is more predictable and easier to analyze.

Technique 3.2.1 ▶ Converting NFA to DFA

To convert an NFA to a DFA, we carry out the following steps:

1. Let $N := (Q, \Sigma, \delta, q_0, F)$ be an NFA that recognizes the language A.
2. Construct the DFA M that also recognizes A. For convenience, define:

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

3. For all $R \subseteq Q$, define $E(R)$ to be the collection of all states that can be reached from R by going along ϵ transitions, including members of R themselves.
4. Modify δ' to place additional edges on all states that can be reached by going along ϵ edges after every step.
5. Set $q_0' = E(\{q_0\})$ and $F' = \{R \in Q' : R \text{ contains an accept state of } N\}$.

examples of NFA/DFA equivalence here

4 Irregular Languages

Definition 4.0.1 ▸ Regular Expression (RE)

We say that R is a ***Regular expression*** if R is one of the following:

- a for some $a \in \Sigma$ (a symbol of the alphabet)
- ϵ (language contains only the empty string)
- $\varnothing$ (language contains no strings)
- $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions
- $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions
- R_1^*, where R_1 is a regular expression

what the hell is this definition

Theorem 4.0.1

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

Proof. If a language R is described by a regular expression, then A is recognized by an NFA. Thus, A must be regular. If the language R is regular, then it is recognized by a DFA from which a regular expression can be deduced. □

Example 4.0.1 ▸ Irregular Languages

Consider the following languages:

- $B = \{0^n 1^n : n \geq 0\}$ is not regular.
- $C = \{w : w$ has an equal number of 0s and 1s$\}$ is not regular. We would need infinite states to account for all possible inputs.
- $D = \{w : w$ has an equal number of 01 and 10 substrings$\}$ is regular. We can think of this as recording transitions between 0s and 1s, and vice versa. In this sense, every 01 must be eventually followed by a 10, and every 10 must be eventually followed by a 01.

4.1 Pumping Lemma

The pumping lemma is a powerful tool in the theory of computation that helps us reason about the limitations of regular languages. Regular languages are those that can be recognized by a finite automaton, but not all languages are regular. The pumping lemma provides a way to prove that certain languages are not regular by demonstrating that they cannot satisfy certain conditions.

The pumping lemma works by breaking a long string in a language into smaller parts in a way that exposes its structure. If the language is regular, then we should be able to divide the string into parts in such a way that each part can be repeated or removed to create a new string that is still in the language. However, if the language is not regular, then at some point we will reach a part that cannot be repeated or removed without breaking the rules of the language.

Theorem 4.1.1 ▸ Pumping Lemma

Let L be a regular language. There exists a pumping length $p \in \mathbb{N}$ such that, for any string s of length p or more, s may be divided into three substrings, $s = xyz$ where all the following hold:

1. $\forall (n \geq 0)\,(xy^n z \in L)$
2. $|y| > 0$
3. $|xy| \leq p$

Proof sketch. As a guide, the proof follows this general form:

- Let $M := (Q, \Sigma, \delta, q_1, F)$ be a DFA that recognizes A. Let p be the number of states.
- Show that any string $s \in A$ where $|s| \geq p$ can be broken into xyz satisfying the three conditions.
- If there are no strings in A of length p or more, then the lemma is true because all three conditions hold for all strings of length p or more.

□

A formal proof is as follows:

Proof. Let $M := (Q, \Sigma, \delta, q_1, F)$ be a DFA that recognizes A. Let $p := |Q|$. Let $s := s_1 s_2 \ldots s_n$ be a string over Σ with $n \geq p$ and $r = r_1 r_2 \ldots r_{n+1}$ be the sequence of states encountered while processing s.

TODO: finish proof

□

Technique 4.1.1 ▸ Proving a Language is Irregular

To prove that a language A is not regular, we:

1. Suppose for contradiction A is regular
2. Find $s \in A$ where $|s| \geq p$ that cannot be pumped; consider all the ways of dividing s into xyz, and show that for each division, at least one of the conditions fail.

Example 4.1.1 ▸ Simple Pumping Lemma Example

Prove that the language $B \coloneqq \{0^n 1^n : n \geq 0\}$ is not regular.

We can show that any substring of B cannot be pumped by contradicting the first condition of the Pumping Lemma.

Proof. Suppose for contradiction B is regular. Let p be the pumping length for B. Let $s \coloneqq 0^p 1^p \in B$. Then $|s| = 2p > p$. By the Pumping Lemma, we can partition s into three substrings, say x, y, z, such that $xy^*z \in B$. Let $s' \coloneqq xyyz$. WE will show $s' \notin B$. Consider the three possible cases for the contents of y:

1. $y \in 0^+$. Then. s' has more 0s than 1s. Thus, $s' \notin B$, contradicting the first condition of the Pumping Lemma.
2. $y \in 1^+$. Then, following the logic from the first case, this is not possible.
3. y consists of 0s and 1s. Then $yy \notin 0^n 1^n$, so $s' \notin B$.

□

We can simplify the above proof by targeting condition 3 instead:

Proof. By the third condition of the Pumping Lemma, we have $|xy| \leq p$. Hence, y could only contain 0s. □

Example 4.1.2 ▸ Another Pumping Lemma Example

Prove that the language $E \coloneqq \{0^i 1^j : i > j\}$ is not regular.

Proof. Suppose for contradiction E is regular. Let p be the pumping length for E, and let $s \coloneqq 0^{p+1}1^p \in E$. Then $|s| = (p+1) + p > p$. By the Pumping Lemma, we can partition s into three substrings, say x, y, z, such that $xy^*z \in E$. Then, by the third condition, we have $|xy| \leq p$. Note that there are $p+1$ number of 0s at the beginning of s. Thus, $x \in 0^*$ and $y \in 0^+$. Consider the case when $i = 0$. Then $xy^+z \notin E$, leaving only xz. However, $|y| > 0$ by the second condition, so xz would lose at least one 0. Thus, $xz \notin E$, so $xy^*z \notin E$. Therefore, E is not regular. ▢

5 Context-Free Languages

Certain languages like $L := \{0^n 1^n : n \geq 0\}$ cannot be specified by neither a finite automaton nor a regular expression. To address this, we can use context-free grammars to specify a much larger class of languages.

Definition 5.0.1 ▸ Context-Free Grammar (CFG)

A context-free grammar consists of a set of production rules that describe how to generate strings. The rules consist of a left-hand side symbol and a right-hand side string.

More formally, a ***context-free grammar*** can be defined as 4-tuple (V, Σ, R, S) where:

- V is a finite set of symbols called ***variables*** or ***non-terminals***
- Σ is a finite set of symbols disjoint from V called ***terminals***
- R is a finite relation from V to $(V \cup \Sigma)^*$ (i.e. a set of ***production rules***)
- $S \in V$ is the ***start variable***

Instead of writing $(a, b) \in R$, we write $a \to b$.

To generate a string using a CFG, we begin with the start variable S. Then, we repeatedly apply the production rules in R to replace a variable with a sequence of symbols until no variables remain in the string. In each step, we choose a variable in the string and use a production rule that has that variable on the left-hand side to generate a new sequence of symbols to replace that variable on the right-hand side. We repeat this process until we have a string consisting only of terminal symbols from T.

Example 5.0.1 ▸ Simple CFG

Suppose CFG G_1 has the following specification rules:

$$A \mapsto 0A1$$
$$A \mapsto B$$
$$B \mapsto \#$$

The start variable for G_1 is A. The non-terminals are A and B. The terminals are 0, 1, and

#. An example output may look like:

$$0A1, 00A11, 000A111, 0000B111, 000\#111$$

We can add a rule like $B \mapsto \epsilon$ to erase strings. Then the final output would be 000111.

Derivation:

$$A \to 0A1 \to 00A11 \to 000A1111 \to 000B111 \to 000\#111$$

Definition 5.0.2 ▸ Direct Derivation

If $uv, w, \in (V \cup \mathrm{E})^*$ and $A \mapsto w \in R$ is a grammar rule, then we say that uvw is ***directly derived*** from uAv using the rule $A \to w$.

Example 5.0.2 ▸ Language of Simple CFG

Let $G_3 := (\{S\}, \{a, b\}, \{S \to aSb|SS|\epsilon\}, S)$ be a CFG. $L(G_3)$ is the language of all strings of properly-nested pair-delimiters (e.g. parentheses or brackets).

Example 5.0.3 ▸ Another Language of Simple CFG

Let $G_3 := (\{E, T, F\}, \{a, +, *, (,)\}, R, E)$ where R is given by:

$$E \mapsto E + T|T, \quad T \mapsto T * F|F, \quad F \mapsto (E)|a$$

$L(G_4)$ is the language of some arithmetic expressions.

Definition 5.0.3 ▸ Ambiguous Grammar

A grammar is considered ***ambiguous*** if it can generate the same string in different ways.

5.1 Design Techniques

Many CFGs are unions of simpler CFGs. Combination involves putting all the rules together and adding the new rules.

Index

Definitions

Examples

Techniques

Theorems