Algorithm Analysis and Automata

UT Knoxville, Spring 2023, COSC 312

Dr. Michael W. Berry, Alex Zhang

March 6, 2023

Contents

Preface

These are my notes for the **Algorithm Analysis and Automata** course at UT Knoxville. They are a compilation of lecture notes by Dr. Michael W. Berry, lecture notes from the University of Illinois' CS 373[1], and Michael Sisper's *Introduction to the Theory of Compution*, as well as online resources like the CS Stack Exchange and Wikipedia.

[1]`https://courses.engr.illinois.edu/cs373/fa2010/lectures/`

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.

We can define these terms more formally:

Definition 1.0.1 ▶ Alphabet, Symbols

An ***alphabet*** is any nonempty, finite set. Members of the alphabet are called the ***symbols*** of the alphabet.

We denote the alphabet using the Greek letter Σ.

Definition 1.0.2 ▶ String

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

- The length of a string w is denoted by $|w|$.
- ϵ represents the ***empty string***, the string of length 0
- The ***concatenation*** of w_1 and w_2 is the string obtained from appending w_2 to the end of w_1. We denote concatenation either by juxtaposition $w_1 w_2$, or with superscript notation w_1^n.

For example, if $w_1 = \text{hello}$ and $w_2 = \text{world}$, then $w_1 w_2 = \text{helloworld}$.

2

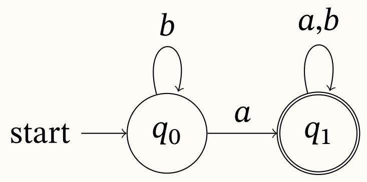
Regular Languages

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 2.0.2 ▶ 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.

2.1 Deterministic Finite Automata

Definition 2.1.1 ▶ 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 the ***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 2.1.2 ▸ String Acceptance, String Rejection

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

Definition 2.1.3 ▸ Language

A ***language*** is simply a set of strings. We say L is a language over an alphabet Σ if every word in L is a string over Σ. We say L is the language of a machine M if L contains every string that M accepts, denoted by $L(M)$.

Technique 2.1.4 ▸ 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 2.1.5 ▶ 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 2.1.6 ▶ 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 2.1.7 ▶ 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 2.1.8 ▸ 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

2.2 Regular Languages

Definition 2.2.1 ▸ Language Recognition, Regular Language

We say that a finite automaton M ***recognizes*** a language L if $L = \{w : M \text{ accepts } w\}$. We say a language is ***regular*** if there exists some finite automaton that recognizes it.

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

Definition 2.2.2 ▸ Concatenation

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

$$L_1 \circ L_2 := \{xy : (x \in L_1) \wedge (y \in L_2)\}$$

For example, if $L_1 = \{\text{bob}\}$ and $L_2 = \{\text{cat}\}$, then there is only one string in $L_1 \circ L_2$: bobcat.

Definition 2.2.3 ▸ Exponent, Kleene Star, Kleene Plus

Let L be a language, and let $n \in \mathbb{N}$. We define a language raised to some ***exponent*** as:

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

The ***Kleene star*** (or Kleene closure) on L is defined as:

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

The ***Kleene plus*** on L is defined as:

$$\begin{aligned} L^+ &:= \bigcup_{i=1}^{\infty} L^i \\ &= L^1 \circ L^2 \circ \cdots \end{aligned}$$

For example, if we had a language like $L := \{a, b, c\}$:

- L^5 contains any combination of a, b, and c of length 5.
- L^* contains any combination of a, b, and c, regardless of length, including the empty string.
- L^+ contains any combination of a, b, and c whose length is 1 or more.

The key difference between the L^* and L^+ is that L^* includes the empty string.

Theorem 2.2.4 ▸ Closure of Regular Languages

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

Need proof here!

2.3 Nondeterminism

Designing a DFA that accepts a complex language can be a challenging task. To address this, we can introduce the concept of ***nondeterminism***: allowing 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.

Definition 2.3.1 ▸ Nondeterministic Finite Automaton (NFA)

A ***nondeterministic finite automaton*** (NFA) is a finite automaton that can transition states without any input.

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***
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ 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, the empty string $\epsilon \in \Sigma$ represents a state transition that was chosen by the machine.

With an 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 than DFAs.

Theorem 2.3.2 ▸ Closure of Regular Languages

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

Proof sketch. ▢

draw proof sketch here

2.4 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 2.4.1 ▸ 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, also this explanation sucks

3 Irregular Languages

Definition 3.0.1 ▸ Regular Expression (RE)

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

- $\{a\}$ for some $a \in \Sigma$ (language of one symbol)
- $\{\epsilon\}$ (language of only the empty string)
- $\varnothing$ (language of 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

Theorem 3.0.2

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 3.0.3 ▸ 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.

3.1 Pumping Lemma

The pumping lemma is a powerful tool that helps us reason about the limitations of regular languages. Recall that a language is ***regular*** if there exists a DFA that recognizes the language. The pumping lemma provides a way to prove that certain languages are not regular by demonstrating that they cannot satisfy certain conditions.

In this context, ***pumping*** refers to repeating a string. If we had the string "abc", we can pump "b" to be "abbbbc", or ***pump down*** to get "ac".

The pumping lemma works by partitioning a long string in such a way that exposes its structure. If the language is regular, then there should be a way to partition the string into three substrings such that the middle substring can be pumped up or down and still be in the language. However, if the language is not regular, then at some point we will reach a part that cannot be pumped without breaking the rules of the language.

Theorem 3.1.1 ▸ Pumping Lemma

If L is a regular language, then there exists some $p \in \mathbb{N}$ (the pumping length) such that for all $s \in L$ where $|s| \geq p$, s may be divided into three substrings, $s = xyz$, satisfying the three following conditions:

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

Proof. Let $M \coloneqq (Q, \Sigma, \delta, q_1, F)$ be a DFA that recognizes L, and let p be the number of states in M (i.e. $p \coloneqq |Q|$).

Let $s \coloneqq s_1 s_2 \cdots s_n \in L$ where $n \geq p$. Let $r_1, \dots, r_{n+1}$ be the sequence of states that M enters when processing s (i.e. $r_1 = q_1$, and $r_{i+1} = \delta(r_i, s_i)$ for $1 \leq i \leq n$). By the Pigeonhole Principle, then two of the first $p + 1$ elements of the sequence must be the same. Let r_a be the first of these, and let r_b be the second of these.

Now, let $x \coloneqq s_1 \cdots s_{a-1}$, $y \coloneqq s_a \cdots s_{b-1}$, and $z \coloneqq s_b \cdots s_n$. Then x takes M from r_1 to r_a, y takes M from r_a to r_a, and z takes M from r_a to r_{n+1} (the accept state).

1. Thus, M must accept $xy^i z$ for any $i \in \mathbb{Z}$ where $i \geq 0$.

2. Since $a \neq b$, then $|y| > 0$.

3. Since r_a occurs within the first $p+1$ places in the sequence of states, then $b \leq p+1$. Thus, $|xy| \leq p$.

Therefore, these conditions hold for any $s \in L$ where $|s| \geq p$. ▢

Technique 3.1.2 ▸ Proving a Language is Irregular

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

1. Suppose for contradiction L is regular, and let p be the pumping length for L.
2. Find some string $s \in L$ 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 3.1.3 ▸ 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 3.1.4 ▸ Another Pumping Lemma Example

Prove that the language $E \coloneqq \left\{0^i 1^j \,:\, i > j\right\}$ 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. ▢

4

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 4.0.1 ▸ Context-Free Grammar (CFG), Context-Free Language

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. We say a language is ***context-free*** if it can be generated by a context-free grammar.

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 4.0.2 ▸ 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 4.0.3 ▶ 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 4.0.4 ▶ 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 4.0.5 ▶ 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 4.0.6 ▶ Ambiguous Grammar

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

4.1 Design Techniques

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

Derivations, parse trees

4.2 Chomsky Normal Form

Definition 4.2.1 ▶ Chomsky Normal Form (CNF)

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

- $A \to BC$ (a variable produces two variables)
- $A \to a$, (a variable produces a terminal)

where a is a terminal, A, B, C are any variables, and B, C aren't the start variable. We also allow the rule $S \to \epsilon$ where S is the start variable.

CNF can simplify any context-free grammar to a (usually very unbalanced) binary tree.

Theorem 4.2.2 ▶ ASdf

Any context-free language can be generated by a context-free grammar in Chomsky normal form.

Intuition: We want to remove any possible recursive rules involving the start variable on the right-hand side. Thus, we simply create a new start variable that maps to the old start variable.

Proof. ▢

Technique 4.2.3

1. Add a new start variable S_0 and rule $S_0 \to S$, where S was the original start variable.
2. Eliminate all ϵ rules. (Repeat the following steps until done)
   (a) Eliminate the rule $A \to \epsilon$ where A is not the start variable.
   (b) For each occurrence of A on the right-hand side of a rule, add a new rule with that occurrence of A deleted.
   (c) If there exists a rule $B \to A$, replace it with $B \to A|\epsilon$ unless the rule $B \to \epsilon$ has not been eliminated.
3. Remove all unit rules. (Repeat the following steps until done)
   (a) Remove any unit rule $A \to B$ (i.e. any rule that replaces one character with

strictly one other character)

(b) For each rule $B \to u$, add the rule $A \to u$, unless it was a previously removed unit rule

4. Convert all remaining rules (Repeat until no rules of the form $A \to u_1 u_2 \ldots u_k$ with $k \geq 3$ remain)

   (a) Re

5

Pushdown Automata

Definition 5.0.1 ▸ Pushdown Automata

A ***pushdown automaton*** is like a NFA, except it can store memory in a stack.

Formally, a ***pushdown automaton*** is defined as a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where:

- Q is a set of states
- Σ is the input alphabet
- Γ is the stack alphabet
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q \times (\Gamma \cup \epsilon))$ is a transition function
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is the set of accept states

We write $a, b \to c$ to mean whenever the machine reads a, it can choose to pop the top element if it is a b, and replace it with c.

In this context, the stack is just some memory where we can only access the most recently added element. We can choose to remove or ***pop*** the top element, or we could ***push*** a new element onto the stack.

A PDA computes as follows:

1. We input some string over the alphabet $\Sigma \cup \{\epsilon\}$.
2. The machine

finish this

A language specified by a context-free grammar can be recognized by a pushdown automaton.

Theorem 5.0.2 ▸ Context-Free Regularity

A langauage is context free if and only if some pushdown automaton recognizes it.

Proof. $\square$

Index