CS 310: Automata Theory

Amit Rajaraman

Spring 2022

Contents

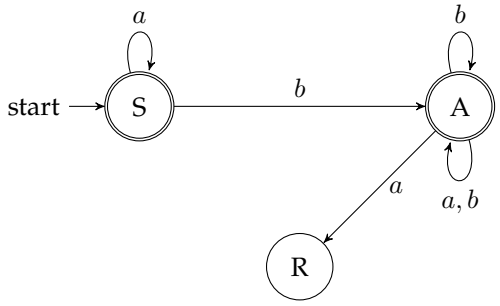
§1. Introduction

1.1. Overview

Consider the problem of determining whether a given multivariate polynomial P with integer coefficients has integer roots. While this may seem simple, this problem is in fact *undecidable*. That is, one cannot write a program on a computer that correctly outputs the answer 'yes' or 'no' to the problem (for any polynomial P). The focus of this course is to study such fundamental limits to computers and computation.
Let us look at the problem of determining whether a certain word is present in a 'language'. For example, consider

$$L_1 = \{a^n b^m : n, m \geq 0\}.$$

Now, we must write a program that given a string s over the alphabet $\{a, b\}$, determines if $s \in L$. This program is in the form of a *discrete finite automaton*.



How would we do this if we instead of have the language

$$L_2 = \{a^n b^n : n \geq 0\}?$$

It may be shown that such a language cannot be recognized by a deterministic finite automaton. To create an automaton that does recognize it, we require an additional *stack*, which gives rise to the *pushdown automaton*. The automaton itself is almost identical to the above automaton, except that we 'push' an a onto the stack when we read an a, and 'pop' an a when we read a b. Finally, we further require that the stack is empty when the string has been read.

We also have what is known as a *context-free grammar*. We have a bunch of rules, and generate strings in the language by repetitively performing a replacement using one of the rules. It turns out that these are equivalent to pushdown automata.

Finally, consider the language

$$L_3 = \{a^n b^n c^n : n \geq 0\}.$$

This cannot be recognized by even a pushdown automaton. This leads to the *Turing machine*, where instead of a stack we have a 'tape'. This represents the 'ultimate' computer that can do anything a computer can do. We shall look at each of these in detail over the next few sections.

The focus of our study shall be each of the following.

Machine	Language
Discrete finite automaton (DFA/FSA)	Regular expressions
Pushdown automaton (PDA)	Context-free grammars
Turing machine (TM)	Unrestricted grammars

We can further introduce non-determinism in each of the three automata. We shall see that the expressive power of DFAs and TMs do not change on allowing non-determinism, while that of PDAs does.

First, before explaining anything, let us set up some notation and definitions for the rest of this course.

Definition 1.1. An *alphabet* Σ is a non-empty set. Its elements are referred to as *letters* or *terminals*.
The set Σ^* is the set of all finite strings over Σ. In particular, Σ^* contains the empty string denoted ϵ. The set Σ^+ is equal to $\Sigma^* \setminus \{\epsilon\}$.
A *language* is a subset of Σ^*.

The *Chomsky hierarchy* represents the increasing complexity of languages. At the lowest level, we have *regular languages* that are recognized by *finite state automata*. This is a subset of *context-free grammars*, which are recognized by *pushdown automata*. This in turn is a subset of *unrestricted grammars*, which are recognized by *Turing machines*.

A finite state automaton is a tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite non-empty set of states, Σ is an alphabet, $\delta : Q \times \delta \to Q$ is the transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states.

Next, let us consider the language $\{a^n b^n : n \geq 0\}$ we mentioned earlier. This can be recognized by the context-free grammar

$$S \to \epsilon$$
$$S \to aSb$$

What this means is that beginning with the string S, we keep performing replacements using one of the two rules above until our current string is composed of only terminals.
Another example is that of the set of non-empty strings with matched parentheses:

$$S \to ()$$
$$S \to (S)$$
$$S \to SS$$

A context-free grammar is a tuple (V, Σ, R, S), where V is a set of *non-terminals* or *variables*, Σ is the alphabet, $R : V \to (V \cup \Sigma)^*$ is the finite set of *rules*, and $S \in V$ is the *start symbol*. The language of this grammar is the set of all strings (over terminals) derivable using the rules.

Finally, let us look at unrestricted grammars. Consider the language $\{a^n b^n c^n : n \geq 1\}$. This can be represented by the unrestricted grammar

$$S \to abc$$
$$S \to aAbc$$
$$Ab \to bA$$
$$Ac \to Bbcc$$
$$bB \to Bb$$
$$aB \to aa$$
$$aB \to aaA$$

The only difference between unrestricted and context-free grammars is that the former allow the left-hand side of the rules to be strings as well. That is, an unrestricted grammar is a tuple (V, Σ, R, S), where V is a set of non-terminals, Σ is the alphabet, $R : (V \cup \Sigma)^* \to (V \cup \Sigma)^*$ is the finite set of rules, and $S \in V$ is the start symbol. As before, the language of this grammar is the set of all strings (over terminals) derivable using the rules.

Similar to how context-free grammars are equivalent to pushdown automata, *regular expressions* are equivalent to ordinary finite state automata. These are defined as follows:

1. $\varnothing$, $\{\epsilon\}$, and $\{a\}$ (for any $a \in \Sigma$) are regular expressions. The second and third sets are often represented as just ϵ or a.

2. if E_1, E_2 are regular expressions,
   (a) $E_1 + E_2$, where the $+$ represents union, is a regular expression.
   (b) E_1E_2, the concatenation of the two expressions, is a regular expression.

$$E_1E_2 = \{uv : u \in E_1, v \in E_2\}.$$

   (c) E_1^* is a regular expression, where * is the *Kleene star* operation defined by

$$E_1^* = \{u_1u_2 \cdots u_n : n \geq 0, u_i \in E_1 \text{ for each } i\}.$$

   (d) (E_1) is a regular expression.

So, for example, the language L comprised of the set of all strings with an even number of as over the alphabet $\Sigma = \{a, b\}$ can be defined using the regular expression $b^*(ab^*ab^*)^*$. As a slightly harder example, the language L comprised of the set of all strings with an even number of as and odd number of bs over the alphabet $\Sigma = \{a, b\}$ can be defined as.

In pushdown automata, we have a normal FSM along with a stack. The stack has a stack alphabet (that need not be the same as the actual alphabet of the automaton). On each letter of input, we can move to another state and either push a particular symbol onto the stack or pop the topmost symbol. This is usually drawn in the usual FSM way by changing the label of each arrow from just something like a (the read letter) to $a, X|aX$ or $b, aX|X$, which represents that we transition to another state and also push a onto the stack/pop a from the stack. The stack begins with a single symbol Z_0, which represents the bottom of the stack. Note that this means that Z_0 is always a part of the stack alphabet. Finally, we accept the input iff the stack is empty.