Theory of Computation

Claire Driedger

Feb-June, 2025

References: Lectures by Stephen Cranefield in COSC341 at the University of Otago, New Zealand; Lecture Slides mostly by Michael Albert; tikz tutorial by Satyaki Sikdar

Contents

1 Deterministic Finite State Automaton (DFAs)

1.1 Introduction to DFAs

DEF: A *deterministic finite state automaton (DFA)*, **A**, consists of the following:

- A finite set Σ called its alphabet,
  the *alphabet* can be thought of as keys on the keyboard, or events
- A finite set $\mathcal{S}$ called its states,
  the *states* can be thought of as symbols without meaning
- A function $T : \mathcal{S} \times \Sigma \to \mathcal{S}$ called its transition function,
  the *transition function* takes as input the current state and a letter from the alphabet
- A single element $s \in \mathcal{S}$ called its start state,
- A subset $A \subseteq \mathcal{S}$ called its final states or accepting states.

Notes:

- Changes to the state of any model always occur sequentially, not in parallel.
- The DFA has a limited, finite memory, and no history

We begin with an example. Consider a light with two switches. Flipping either switch changes the state of the light.

Ex:

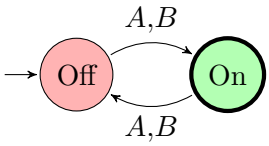


Figure 1: Two buttons, one light

- The light starts in the **Off** state.
- In either state, making an input of either A or B switches to the other state.
- We consider the **On** state to be *accepting* – any sequence of inputs that leads to this state is considered successful
- The successful inputs are all strings consisting of characters from A, B that have an odd number of characters.

Ex:

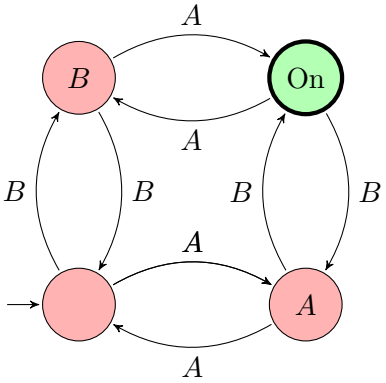
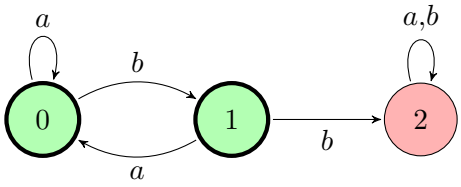


Figure 2: Two different buttons, one light

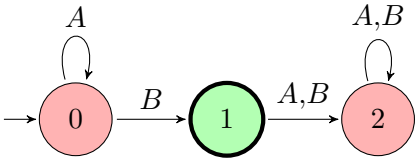
- The light starts in the lower left state.
- The successful inputs are all strings consisting of characters from $\{A, B\}$ that have an odd number of each letter.

Ex: What does this machine do?



- The machine starts in the state labelled 0.
- There are two buttons to press - a and b. These define the alphabet of the machine.
- Each button press causes a transition according to the labelled arrows.
- The sequence of button presses leaving us in an accepting state (coloured green) are the language accepted by the machine.

Ex:



In this example, the accepted language is exactly "zero or more copies of A followed by exactly one B".

1.2 Automata and Grammars

DEF: A *sequence of length n over Σ* is a function:

$$\sigma : \{0, 1, 2, \ldots, n-1\} \to \Sigma.$$

the function takes as input the subscript and outputs an element of the alphabet

When $n = 0$, we denote the empty string ϵ.
The set of all strings over Σ is denoted Σ^*.

DEF: A *string of length n over Σ* is a sequence $\sigma_0\sigma_1 \ldots \sigma_{n-1}$ where each $\sigma_i \in \Sigma$.
This is an *array-like* definition of a string. With this definition, we can consider the concatenation of two strings, for example,

$$\text{“}black\text{”} + \text{“}dog\text{”} \to \text{“}blackdog\text{”}$$

With the indexing of "*blackdog*" as $w_0 \ldots w_{n-1}v_0 \ldots v_{k-1}$ given by the following σ function:

$$c(i) = \begin{cases} w(i) & i < n \\ v(i-n) & i \geq n \end{cases}$$

On the other hand, the recursive definition of a string involves composing an element with the previous string. Formally, this involves:

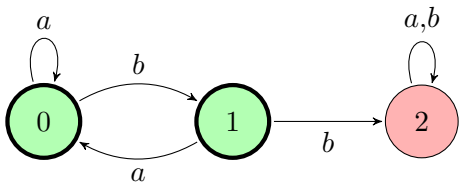
1. Base case: The empty string, ϵ
2. Recursive step: A pair (x, s), where $x \in \Sigma$ and s is a string over Σ

We can similarly define the length of a string recursively by

1. Base case: $len(\epsilon) = 0$
2. Recursive step: $len((x, s)) = 1 + len(s)$

DEF: The *language accepted by* $\boldsymbol{A}$, $L(\mathbf{A})$ is the set of all strings over Σ such that "when you press the buttons defined by the string you wind up in an accepting state".

Ex: What is the language accepted by the following machine?



We begin by listing all strings using formal grammar production rules in text form: start state $\rightarrow$ accepting strings:

$$S_0 \rightarrow \epsilon \mid aS_0 \mid bS_1$$
$$S_1 \rightarrow \epsilon \mid aS_0 \mid bS_2$$
$$S_2 \rightarrow aS_2 \mid bS_2$$

ϵ is included as a valid transition for S_0 and S_1 because these are already accepting states. However, S_2 is not an accepting state, so ϵ is not included as a transition state.

The idea is to think about for each state X (and not only the DFA's start state) all the strings that you could accept starting from X. A transition $T(S, a) = B$ in the DFA tells us that one way to accept a string (if we were allowed to start with state S) is to receive an a and then accept a string *starting from* B. That leads to the grammar production rule $S \rightarrow aB$.
Each state in the DFA corresponds to a non-terminal symbol in the grammar.

The language accepted by the machine is:

$$S \rightarrow \epsilon \mid aS \mid bB$$
$$B \rightarrow \epsilon \mid aS$$

1.2.1 Regular grammars

DEF: A right-regular grammar, G, consists of:

- An alphabet, Σ, $(\{a, b\})$.
- A set N of non-terminal symbols, $(\{S, B\})$ (the *states*)
- A designated start symbol, $S \in N$
- A list of production rules $(S \rightarrow aS)$ each of one of the forms:
  - $X \rightarrow \epsilon$ $(X \in N)$

- $X \to a$ $(X \in N,\, a \in \Sigma)$
- $X \to aY$ $(X, Y \in N,\, a \in \Sigma)$

Note the recursive nature of the production rules. Events from the language are 'terminal' and states are 'non-terminal'. To generate the language, we continue applying production rules until there are no non-terminal symbols left.
We might ask: "are the accepting states of the DFA the same as the language generated by a right-regular grammar?"

1.2.2 From grammar to language

Each non-terminal, X of a regular grammar, G, has an associated language, $L_G(X)$ (or $L(X)$) which is a subset of Σ^* defined recursively as follows:

1. If $X \to \epsilon \in G$, then $\epsilon \in L(X)$, and if $X \to a \in G$, then $a \in L(X)$.
2. For every production $X \to aY$ in G, $L(X)$ contains the set $aL(Y)$, meaning whatever can be generated from the new state Y.

When we say "defined recursively", we implicitely say "and no other strings except those which must belong to $L(X)$ according to these rules are included".

The language of the grammar itself is the language of its start state, S.

Ex: Consider the right-regular grammar G defined as follows:

$$S \to \epsilon \mid aS \mid bB$$
$$B \to \epsilon \mid aS$$

This means, for example, that 'a' followed by anything in the language for S is in the language.
We can build the language $L(S)$ (generated by S) and $L(B)$ (generated by B) from the bottom up:

- $L(S) = \epsilon,\ a\ (a + \epsilon),\ b\ \ (b + \epsilon),\ aa\ \ (a + a),\ ab\ (a + b),\ ba,\ \ldots$
- $L(B) = \epsilon,\ a,\ (a + \epsilon),\ aa,\ ab,\ a + aa,\ aab,\ aba,\ \ldots$

Characters are added on the left, rather than the right.

Things which might be allowed in a regular grammar that are not allowed in a DFA:

1. Having multiple production rules with the same initial symbol

   $$A \to aB \mid aC \mid bD$$

   because at most one arrow leaves a node for each element of the alphabet

2. Have no production rule with some initial symbol

   $$B \to \epsilon \mid aS$$

   because we need to consider at least one state for each letter of the alphabet.

In a DFA, each state is supposed to have exactly one outgoing arrow corresponding to each letter of the alphabet.
Do regular grammars potentially allow for *more languages* than are represented by DFAs?

1.3 Uncomputability

Thm: No reasonable model of computation allows **every** language to be recognized.

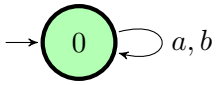
Proof. Programs: $p_1, p_2, p_3, \ldots$ (a sequence of programs)
Languages: $l1, l2, l3, \ldots$ (a sequence of languages)
We will show that there exists a language $l = \ldots$ s.t. $L \neq l_i$ for any i Diagonalization proof: s1 s2 s3 l1 $\in \notin$ l2 $\in \notin$
many languages will be the empty languages because it is syntactially incorrect string 1 is either in L1 or not. Construct our new language with the diagonal: this language is new, and does not correspond to any language produced by our machines we enumerated all the possible machines: index all the languages they recognize

2 Tutorials

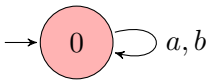
2.1 Tutorial 01

For this tutorial, the language $\Sigma = \{a, b\}$ is to be used throughout.

1. *How many different DFAs are there having only one state? What languages do they accept?*
   → There are exactly two DFAs having only one state (such a state is the start state and both transitions must loop on that state.) The only choice is whether or not it is an accepting state. If the start is an accepting state, it accepts all strings (i.e. its language is Σ^*), while the one where it is not accepting accepts no strings at all (i.e. it's language is $\{\}$, which does not even include the empty string).



$L(A) = \Sigma^*$



$L(A) = \{\}$

2. *A state in a DFA is unreachable if there is no input sequence that leads to that DFA. Note that the start state is always reachable since the empty sequence ϵ leads to it. If we remove the unreachable states from a DFA how does that affect the language it accepts?*
   → Since a DFA can never enter an unreachable state, removing unreachable states does not affect the strings accepted by the DFA.

3. *How many different DFAs are there having only two states? What languages do they accept? Are they all different?*
   → Let the two states of such a DFA be 0 and 1. Which states are accepting and what are the transitions?

   (a) There are two choices of accepting/not-accepting for each of the two states.

   (b) For each transition, there are two choices for its endpoint. There are two transitions for each of the two states.

   Therefore, we have six binary choices to make and the number of two-state DFAs is

$$2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^6 = 64$$

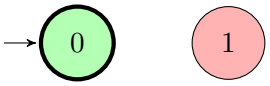
In general, a DFA with k states and an alphabet of size n has k binary state choices (accepting or not) and kn transitions to choose endpoints for, each one having k possibilities, generating

$$2^k \times k^{kn}$$

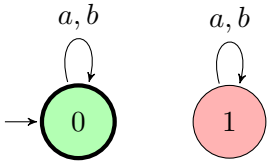
possible $k - state$ DFAs.
For the accepting lanauge for the the two-state DFA:

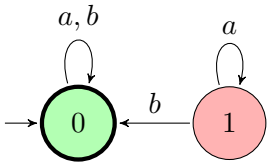
- If both states are accepting, the language is Σ^*
- If both states are non-accepting, the language is {}

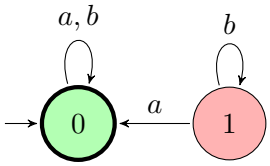
This accounts for 32 DFAs. For the remaining 32 DFAs, one state is accepting and the other is not, and interchanging the states complements the accepting language. Consider the DFAs in which the start state is accepting and the other state isn't:

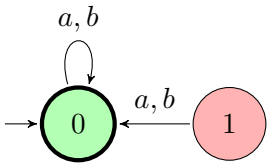


- There are four DFAs in which a and b loop on the start state (the four are generated by the four transition states for a and b from 1). The accepting language is Σ^*.

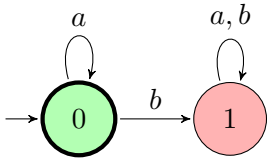




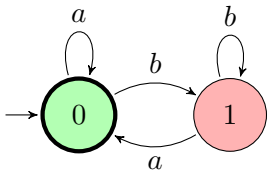




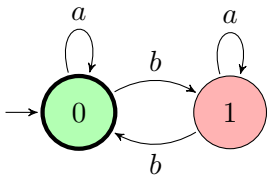
- If a loops and b transitions, it remains to consider the transitions from state 1:



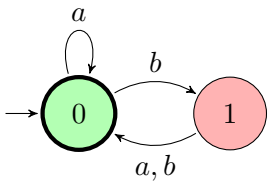
All repetitions of a
0 or more times.



The empty word, and all words ending with a.

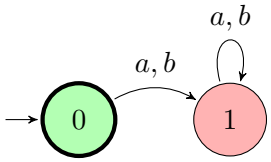


Words with an even number of b's

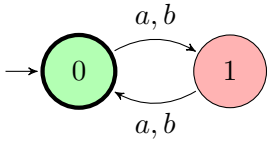


Words ending with a or with an even number of b's

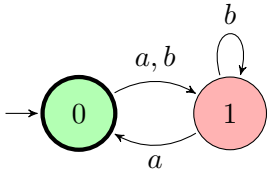
- If both a and b transition:



The accepted language is ϵ



Any word with an even number of letters



ϵ and any word where every second letter is an a???

4. *A state in a DFA that is not accepting and all of those whose transitions are loops back to itself is sometimes called a "garbage state". Why?*
   → Because we can never leave that state to reach an accepting state.

5. *How many three state DFAs are there all of whose states are reachable and exactly one of which is a garbage state? What languages do they accept?* →

2.1.1 Tutorial 02

1. *If Σ has k elements how many different strings of length n are there over Σ?*
   → There are n^k possible strings.

2. *If $w \in \Sigma^*$ and $m, i + j \leq |w|$ how are:*

(a) *the prefix of w of length m,*

(b) *the suffix of w of length m, and*

(c) *the substring of w starting at position i of length j*

defined?
Recall that w is a function:

$$w : \{0, 1, 2, \ldots, n-1\} \to \Sigma$$

(a) Prefix:

$$p : \{0, 1, \ldots, m-1\} \to \Sigma$$
$$p(i) = w(i)$$

(b) Suffix:

$$s : \{0, 1, \ldots, m-1\} \to \Sigma$$
$$s(i) = w(i + n - m)$$

(c) Substring:

$$m : \{0, 1, \ldots, j-1\} \to \Sigma$$
$$m(t) = w(i + t)$$

3. *See S02.*

4. *Determine a regular grammar for the language of all strings containing an even number of a's (over $\Sigma = \{a, b\}$*

$$S \to \epsilon \mid aO \mid bS$$
$$O \to aS \mid bO$$

Basically, S stands for "even number of a's so far" and O for "odd number of a's so far".

5. *Let $\boldsymbol{A}$ and $\boldsymbol{B}$ be DFAs over the same alphabet, Σ. Can you describe DFAs that accept:*

(a) *The complement of the language $L(\boldsymbol{A})$, i.e. the set of all strings not belonging to $L(\boldsymbol{A})$.*

(b) *The intersection of $L(\boldsymbol{A})$ and $L(\boldsymbol{B})$, i.e. the set of all strings belonging to both of $L(\boldsymbol{A})$ and $L(\boldsymbol{B})$.*

(c) *The union of $L(\boldsymbol{A})$ and $L(\boldsymbol{B})$, i.e. the set of all strings belonging to at least one of $L(\boldsymbol{A})$ and $L(\boldsymbol{B})$.*

$\to$

(a) Interchange all the accepting and non-accepting states.

(b) For example, consider two automata: one that accepts strings with an even number of a's and any number of b's, and one that accepts strings with any number of a's and an even number of b's. Their intersection is words with an even number of a's and b's. The "cross product" machine is given on the website www.geeksforgeeks.org/cross-product-operation-in-dfa.com. The new machine will have 2 x 2 = 4 states.

(c) see the website.

6. Repeat the previous question for the languages $L(\mathbf{G})$ and $L(\mathbf{H})$ associated to two *right-regular grammars* over Σ. $\rightarrow$ In general, a right-regular grammar does not align exactly with a DFA (notably, the condition that state needs exactly one transition defined for each letter of the alphabet). In the case of the union, just list out the production rules. If any production rule leads back to the start symbol, we will be required to rename the start symbol for both machines.

$$s \rightarrow S_1 \mid S_2$$

Then include the production rules from both grammars.
Because of the inherent non-determinism of grammars (where we might have multiple rules associated with a particular letter) it's not clear at all how one could realize the intersection of two languages or the complement of one.