13 Nondeterministic Finite Automata

13.1 Introduction

So far we have been able to show that regular languages are closed under union. We still need to show that they are also closed under concatenation and Kleene *.

To show that regular languages are closed under concatenation we would need to do the following.

- Given two regular languages \( L_1 \) and \( L_2 \)
- We know that there are two DFA’s \( M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1) \) and \( M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2) \) such that \( L(M_1) = L_1 \) and \( L(M_2) = L_2 \).
- We must create a new DFA \( M = (Q, \Sigma, \delta, q_0, F) \) that accepts those strings that are in \( L_1 L_2 \).

For example, if \( L_1 = \{ w \in \Sigma^* | w \) has two consecutive b's \} \), and \( L_2 = \{ w \in \Sigma^* | w \) starts and ends with a b \} \), our machine \( M \) should reject the string: \( w = aabaabaaabaaa \), because there is no way to partition it into two consecutive parts \( x \) and \( y \) such that \( x \) has two consecutive b's and \( y \) starts and ends with a b. In order for \( M \) to be able to do this, \( M \) should have to explore ALL possible ways in which the input string \( w \) should be partitioned, and as far as we have seen, a DFA cannot rewind the input tape and start over again. In addition, to determine that the string \( z = aabaabaaabaaa \), is in \( L_1 L_2 \) the machine would also have to explore all possible ways to divide \( z \) so that it will find the two parts: \( x = aabaabaaabaaa \) and \( y = b \), where \( x \in L_1 \) because it has two consecutive b's, and \( y \in L_2 \) because it starts and ends with a b.

What we will do is build a special kind of DFA that can explore many options simultaneously (in parallel), so that it can make the right choice. These machines are called Nondeterministic Finite Automata, or NFA for short.

13.2 Tracing NFA Computations

A deterministic machine always goes through the same computation when given the same input. A nondeterministic machine may behave in different ways when given the same input. The idea is that when the machine is in a given state, reading an input, the machine might have a choice of several states to move to after reading the symbol.
The following diagram is an NFA because when in state 0, there are two transitions labeled $a$. One transition goes to state 1, and the other one stays in state 0, and when in state 2, there are also two transitions labeled $b$. One goes to state 1 and the other one remains in state 2.

To get an idea of how NFAs compute, let us trace the computation of the NFA given above on input $ababb$. Since there are instantaneous configurations that yield more than one instantaneous configuration, we will draw the configurations as a tree, starting with the initial configuration as its root.

(a)

```
             (a)
            [0,aabb]
               / \
  a -------------- a
         [0,abb]   [1,abb]
               / \
  a -------------- a
         [0,bb]   [1,bb]
               / \
   b -------------- b
      Reject   [2,b]
               / \
   b -------------- b
         [1,ε]   [2,ε]
               / \\
      Reject   Accept
```

The computation of the NFA has several computation paths, some of those do not represent an accepting computation, while other paths may represent accepting computation. In the example there is one computation path that
ends up in an accepting state, and so we say that the NFA accepts the input string \textit{ababb}.

**Definition 13.1** An NFA accepts a string \(w\) if there is at least one computation path that ends in an accepting state with no more input to read.

### 13.3 Formal Definition of an NFA

As we have seen, the only difference between an NFA and a DFA is that the transition function of the NFA produces several possible states given a state and a symbol.

**Definition 13.2** An Nondeterministic Finite Automaton (NFA) \(M\) is a five-tuple:

\[ M = (Q, \Sigma, \delta, q_0, F) \]

where

- \(Q\) is a finite set of states.
- \(\Sigma\) is the alphabet over which the transitions are defined.
- \(\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)\)
- \(q_0\) is the start state.
- \(F\) is the set of final states.

We say that an NFA \(M = (Q, \Sigma, \delta, q_0, F)\) accepts a string \(w\), if the computation of \(M\) on input \(w\) ends in a final state. Formally

**Definition 13.3** An NFA \(M = (Q, \Sigma, \delta, q_0, F)\) accepts a string \(w \in \Sigma^*\), if there is a computation such that \([q_0, w] \vdash [q_f, \epsilon]\), and \(q_f \in F\).

In the same way as we did in the case of DFAs, we can define the language of the machine as the set of all those strings that are accepted by it.

**Definition 13.4** Given an NFA \(M = (Q, \Sigma, \delta, q_0, F)\), the language of \(M\) is \(L(M) = \{w \in \Sigma^* | M\text{ accepts } w\}\)

### 13.4 Examples of NFAs

**Example.** For this example we will design an NFA that accepts those strings that end with three consecutive \textit{a}'s. The idea is that the NFA will read \textit{a}'s and \textit{b}'s initially, until it “decides” that it is time to read the last three symbols, and they should be \textit{a}'s. Rephrasing in terms of states. The NFA will be in state 0
reading either a’s or b’s, and will have the choice of moving to state 1, when it expects the last three symbols. The following is the state diagram of our NFA.

If we input the string \( w = ababaa \) to this machine, there might be several computation paths that do not end in a final state, but there is one computation path that ends in a final state:

\[
[0, ababaa] \vdash [0, babaa] \vdash [0, abaaa] \vdash [0, baaa] \vdash [0, aaa] \vdash [1, aa] \vdash [2, a] \vdash [3, \epsilon]
\]

Since there is at least one computation that ends in a final state, then the string is accepted by the machine.

Notice that we could trace all possible computations of the machine on the input string \( x =aabaa \) and there is no computation path that accepts, therefore, it will be rejected. The tracing of all these computation paths is left as an exercise.

---

**Example.** Let us now build an NFA that accepts the language of those strings that have \( abb \) and \( bba \) as disjoint substrings. For example, the string \( aababbba \) should be accepted because \( aababbba \).

Our machine will be in a state reading a’s and b’s until it “guesses” that the substring \( abb \) is next. It will then read \( abb \) and go to a state where it is just reading a’s and b’s, until it “guesses” that the substring \( bba \) is next. It will then read \( bba \), and go to a final state where it will read the remaining section of the string.

The machine we have so far should look like this:

However, this machine will fail to accept the string \( bbaaaabb \), which clearly is in the language \( (bbaaabb) \).

This problem is easily fixed by adding another transition from the start state to take care of this situation, as illustrated next.