Lecture 4: Nondeterminism

built on 2021/01/13 at 11:33:01

Last time, we saw a union theorem and an intersection theorem: The union of two regular languages is also regular. Similarly, the intersection of two regular languages is also regular. For this, we usually say regular languages are *closed* under union, and regular languages are closed under intersection.

1 More Operations

There are a few other operations that we want to consider.

1.1 Complement

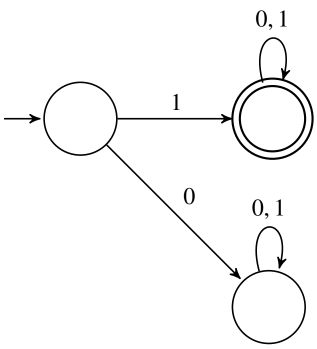
For a language, let the complement of L, denoted by $\neg L$, be defined as

$$\neg L = \{w \in \Sigma^* \mid w \notin L\}.$$

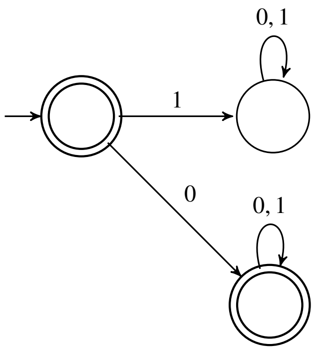
Q: If L is regular, is $\neg L$ necessarily regular?

To answer this question, let's play with a few examples. Consider the langauge $L_1 = \{0, 1\}^*$, the language of all binary strings. The complement of L_1 is the empty language, which is regular.

Another example: let L_2 be the language $\{w : \{0, 1\}^* \mid w \text{ begins with a } 1\}$. Is L_2 regular? How about $\neg L_2$?



A DFA for L_2



A DFA for $\neg L_2$

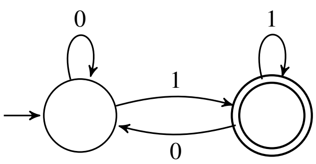
It is not hard to see that both L_2 and $\neg L_2$ are regular, providing further evidence that perhaps *regular languages are closed under complement*. Indeed, we have the following theorem:

Theorem 1.1 (Complement Theorem) *If L is a regular language, then $\neg L$ is also regular.*

How do we prove this? Idea: change all accepting states to nonaccepting and vice versa.

1.2 Reverse?

Let's continue with the language L_2 (above). We know that L_2 is regular because there's a DFA that recognizes it. We'll call the machine above M_2 as $L(M_2) = L_2$. If the input to M_2 is read *right to left*—instead of the usual left to right—then M_2 will recognize a totally different language: $L_2' = \{w \in \Sigma^* \mid w \text{ ends with } 1\}$. Is L_2' regular? You can show that it is by giving, perhaps, the following DFA:



In general, we'll define the reverse operation as follows: for a language L,

$$L^{\mathsf{R}} = \{w \in \Sigma^* \mid \mathsf{rev}(w) \in L\},$$

where rev is the string reverse function.

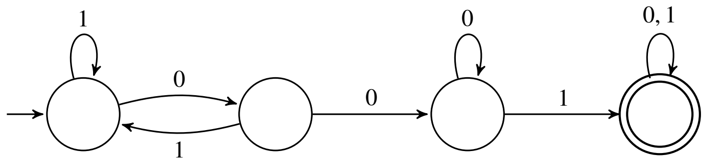
The question is, if L is regular, *is it always the case that L^R is regular?* In other words, can every "right-to-left" DFA be turned into a normal DFA?

Indeed, this is true:

Theorem 1.2 *If L is a regular language, then L^{R} is also regular.*

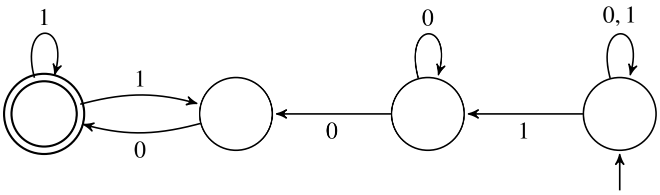
Attempt #1: To prove this, we will follow the same recipe as before. We know that because L is regular, there is a DFA M that recognizes it. So, our job has become to build a DFA M^{R} that accepts L^{R}. But how?

To experiment with some ideas, let's work with a slightly more complex DFA (remember it recognizes the existence of consecutive "`001`".):



Here's an idea:

- we'll reverse all the arrows;
- turn the start state into an accepting state;
- turn the accepting states into starting states.



The problem is, the resulting state-transition diagram isn't even a DFA: there could be many starting states. Worse yet, some states may have multiple outgoing edges for the same symbol—or none at all.

In other words, there are many places to start from. And you may end up being in multiple states at the same time because there are more than one possible action for each symbol. Nondeterminism is born!

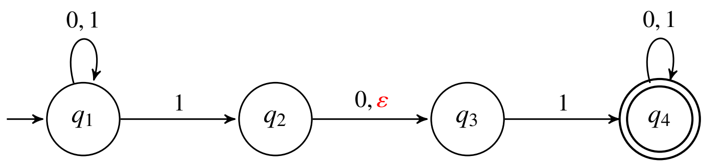
2 Nondeterministic Finite Automata (NFAs)

Our discussion of finite-state machines thus far has the flavor: every step of a computation is uniquely determined by which previous state it was and which symbol it received. Knowing which state it is in and knowing the next symbol, you can determine with 100% certainty what state it is going to be in next—it is completely deterministic. This is why such a computation is called *deterministic* computation. By contrast, in a *nondeterministic* machine, several choices may present themselves at any point.

As a side note: nondeterminism is a generalization of determinism: every deterministic computation is presented with a choice of one.

2.1 NFA Example

We begin our discussion of nondeterministic finite automata with an example. The figure below shows a nondeterministic figure automaton (NFA), called N_1.



Differences from a DFA are easy to spot:

1. In a DFA, every state has one outgoing arrow for each symbol. In an NFA, there can be zero, one, or more outgoing arrows for a symbol.
2. In a DFA, every outgoing arrow is labeled with a symbol in the alphabet. In an NFA, a special symbol ε (the Greek's letter epsilon) is also allowed. See below for how we compute with it. Furthermore, the number of outgoing arrows for a state could be zero, one, or more.

How do we compute with this? The idea is mostly the same as with a DFA. A few things to note:

- Instead of being in a single state at a time, you're omnipresent—you are in multiple states at the same time.
- If there are multiple ways to proceed, you follow all of them (cloning yourself as necessary). If, for example, you are in state q_1 and receive a 1, you're forked into q_1 and q_2.
- If you have no way to proceed, you die out. If, for example, you're at q_3 and receive a 0, there is no way to go, so that copy dies.

The Story of ε: ε is a wormhole that teleports you to places without any symbol at all. Say you're at q_1 and you consume the symbol 1. Two things happen:

- Because q_1 has two outgoing arrows for 1, there will be two copies of you—one in q_1 and one in q_2.
- When a copy of you enters q_2, because q_2 has an outgoing arrow marked with ε—the teleport symbol—this copy of you is furthe split into two: one remains at q_2 and the other is at q_3.

Hence, consuming a 1 when you're in q_1 leads to a total of 3 copies of yourself in q_1, q_2 and q_3.

For intuition, think of nondeterminism as a parallel computing scheme with many threads running at the same time, each working independently. Each thread upon receiving a symbol decides what to do next. If there are multiple states to be, it creates new threads. If this thread has nowhere to go next, it kills itself. In the end, the machine accepts if at least one of the threads is in an accepting state.

2.2 Formality

Definition 2.1 *A* nondeterministic finite automaton *(NFA) is a* 5*-tuple* $N = (Q, \Sigma, \delta, Q_0, F)$ *where*

1. *Q is a set of states;*
2. *Σ is the alphabet;*
3. *$\delta : Q \times \Sigma_\varepsilon \to 2^Q$ is the transition function*
4. *$Q_0 \subseteq Q$ is the set of starting states; and*
5. *$F \subseteq Q$ is the set of accepting states.*

Keep in mind that Σ_ε is the alphabet set augmented with a special symbol ε (denoting the empty string), and 2^Q is the "power set" of Q—that is, the set of all possible subsets of Q.

Therefore, the NFA N_1 (above) has the following formal description: Let $N = (Q, \Sigma, \delta, Q_0, F)$ where

- $Q = \{q_1, q_2, q_3, q_4\}$
- $\Sigma = \{0, 1\}$.
- δ is given as

δ	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\varnothing$
q_2	$\{q_3\}$	$\varnothing$	$\{q_3\}$
q_3	$\varnothing$	$\{q_4\}$	$\varnothing$
q_4	$\{q_4\}$	$\{q_4\}$	$\varnothing$

In this view, N accepts a string $w \in \Sigma^*$ if w can be written as a sequence of symbols $w_1 w_2 \dots w_n$, $w_i \in \Sigma_\varepsilon$ (that is, some of them are empty) and there are $r_0, r_1, \dots, r_n \in Q$ such that

- $r_0 \in Q_0$,
- $r_{i+1} \in \delta(r_i, w_{i+1})$ for $i = 0, 1, \dots, n-1$, and
- $r_n \in F$

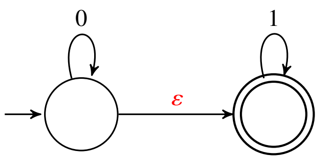
This definition of accept takes a different view than the paralell-computation view discussed earlier. This definition focuses on one thread (the lucky thread) that when presented with multiple ways, picks the "right" choice and ends up with an accepting state.

Does N_1 accept 010110? One way to get accepted is the following:

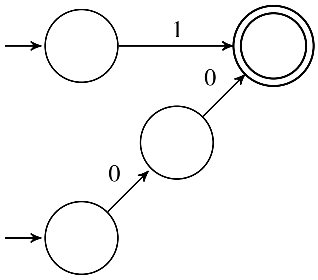
r_0	r_1	r_1	r_3	r_4	r_5	r_6	r_7
	0	1	0	1	ε	1	0
q_1	q_1	q_1	q_1	q_2	q_3	q_4	q_4

By the way, N_1 accepts all binary strings that contain either 101 or 11 as a substring.

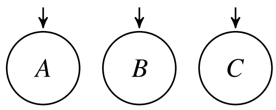
2.3 What do they accept?

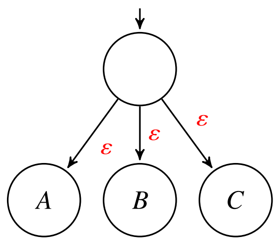


A: binary strings that start with a bunch of 0s followed by a bunch of 1s. That is, $0^i 1^j$.



A:$\{1, 00\}$

Converting Multiple Start States to One: Convert from 

to 

Moral of the story: it's equally good to have only one starting state.

3 Practice

Design an NFA that recognizes all binary strings whose 3rd position from the end is a 1. (*Hint:* You don't need as many states as a DFA.)