Nondeterminism

Michael Sipser

Introduction to the Theory of Computation

This concludes the construction of the finite automaton M that recognizes the union of A_1 and A_2. This construction is fairly simple, and thus its correctness is evident from the strategy described in the proof idea. More complicated constructions require additional discussion to prove correctness. A formal correctness proof for a construction of this type usually proceeds by induction. For an example of a construction proved correct, see the proof of Theorem 1.54. Most of the constructions that you will encounter in this course are fairly simple and so do not require a formal correctness proof.

We have just shown that the union of two regular languages is regular, thereby proving that the class of regular languages is closed under the union operation. We now turn to the concatenation operation and attempt to show that the class of regular languages is closed under that operation, too.

THEOREM 1.26

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

In other words, if A_1 and A_2 are regular languages then so is $A_1 \circ A_2$.

To prove this theorem, let's try something along the lines of the proof of the union case. As before, we can start with finite automata M_1 and M_2 recognizing the regular languages A_1 and A_2. But now, instead of constructing automaton M to accept its input if either M_1 or M_2 accept, it must accept if its input can be broken into two pieces, where M_1 accepts the first piece and M_2 accepts the second piece. The problem is that M doesn't know where to break its input (i.e., where the first part ends and the second begins). To solve this problem, we introduce a new technique called nondeterminism.

1.2

NONDETERMINISM

Nondeterminism is a useful concept that has had great impact on the theory of computation. So far in our discussion, every step of a computation follows in a unique way from the preceding step. When the machine is in a given state and reads the next input symbol, we know what the next state will be—it is determined. We call this ***deterministic*** computation. In a ***nondeterministic*** machine, several choices may exist for the next state at any point.

Nondeterminism is a generalization of determinism, so every deterministic finite automaton is automatically a nondeterministic finite automaton. As Figure 1.27 shows, nondeterministic finite automata may have additional features.

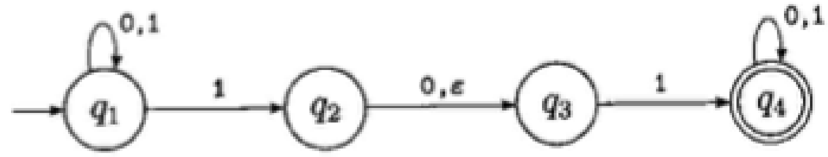

FIGURE 1.27
The nondeterministic finite automaton N_1

The difference between a deterministic finite automaton, abbreviated DFA, and a nondeterministic finite automaton, abbreviated NFA, is immediately apparent. First, every state of a DFA always has exactly one exiting transition arrow for each symbol in the alphabet. The NFA shown in Figure 1.27 violates that rule. State q_1 has one exiting arrow for 0, but it has two for 1; q_2 has one arrow for 0, but it has none for 1. In an NFA, a state may have zero, one, or many exiting arrows for each alphabet symbol.

Second, in a DFA, labels on the transition arrows are symbols from the alphabet. This NFA has an arrow with the label ε. In general, an NFA may have arrows labeled with members of the alphabet or ε. Zero, one, or many arrows may exit from each state with the label ε.

How does an NFA compute? Suppose that we are running an NFA on an input string and come to a state with multiple ways to proceed. For example, say that we are in state q_1 in NFA N_1 and that the next input symbol is a 1. After reading that symbol, the machine splits into multiple copies of itself and follows *all* the possibilities in parallel. Each copy of the machine takes one of the possible ways to proceed and continues as before. If there are subsequent choices, the machine splits again. If the next input symbol doesn't appear on any of the arrows exiting the state occupied by a copy of the machine, that copy of the machine dies, along with the branch of the computation associated with it. Finally, if *any one* of these copies of the machine is in an accept state at the end of the input, the NFA accepts the input string.

If a state with an ε symbol on an exiting arrow is encountered, something similar happens. Without reading any input, the machine splits into multiple copies, one following each of the exiting ε-labeled arrows and one staying at the current state. Then the machine proceeds nondeterministically as before.

Nondeterminism may be viewed as a kind of parallel computation wherein multiple independent "processes" or "threads" can be running concurrently. When the NFA splits to follow several choices, that corresponds to a process "forking" into several children, each proceeding separately. If at least one of these processes accepts, then the entire computation accepts.

Another way to think of a nondeterministic computation is as a tree of possibilities. The root of the tree corresponds to the start of the computation. Every branching point in the tree corresponds to a point in the computation at which the machine has multiple choices. The machine accepts if at least one of the computation branches ends in an accept state, as shown in Figure 1.28.

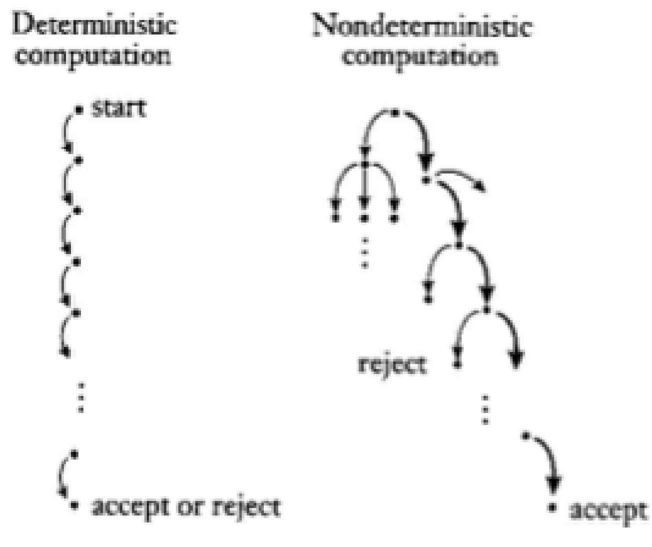

FIGURE 1.28
Deterministic and nondeterministic computations with an accepting branch

Let's consider some sample runs of the NFA N_1 shown in Figure 1.27. The computation of N_1 on input 010110 is depicted in the following figure.

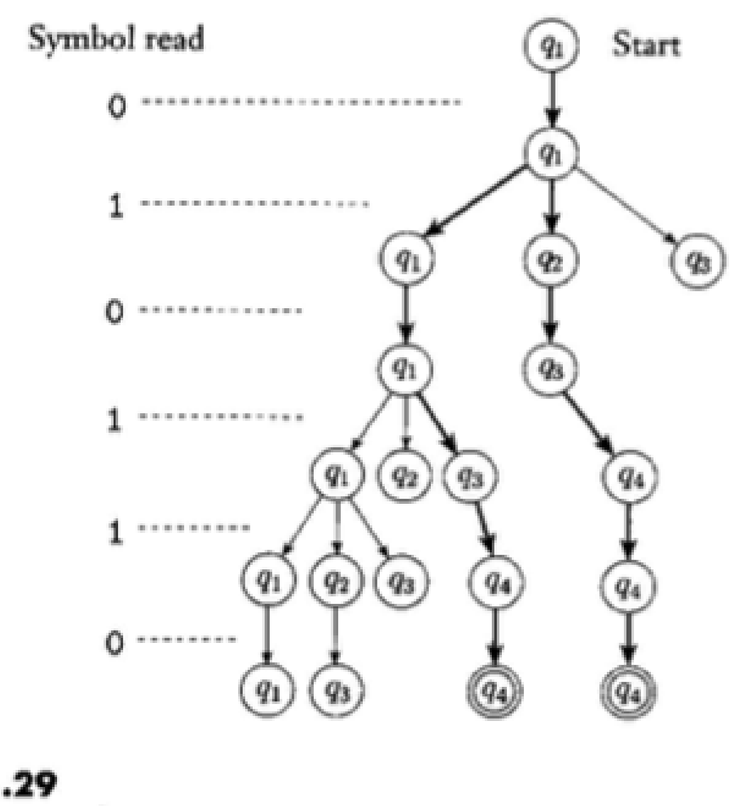

FIGURE 1.29
The computation of N_1 on input 010110

On input 010110, start in the start state q_1 and read the first symbol 0. From q_1 there is only one place to go on a 0—namely, back to q_1—so remain there. Next, read the second symbol 1. In q_1 on a 1 there are two choices: either stay in q_1 or move to q_2. Nondeterministically, the machine splits in two to follow each choice. Keep track of the possibilities by placing a finger on each state where a machine could be. So you now have fingers on states q_1 and q_2. An ε arrow exits state q_2 so the machine splits again; keep one finger on q_2, and move the other to q_3. You now have fingers on q_1, q_2, and q_3.

When the third symbol 0 is read, take each finger in turn. Keep the finger on q_1 in place, move the finger on q_2 to q_3, and remove the finger that has been on q_3. That last finger had no 0 arrow to follow and corresponds to a process that simply "dies." At this point, you have fingers on states q_1 and q_3.

When the fourth symbol 1 is read, split the finger on q_1 into fingers on states q_1 and q_2, then further split the finger on q_2 to follow the ε arrow to q_3, and move the finger that was on q_3 to q_4. You now have a finger on each of the four states.

When the fifth symbol 1 is read, the fingers on q_1 and q_3 result in fingers on states q_1, q_2, q_3, and q_4, as you saw with the fourth symbol. The finger on state q_2 is removed. The finger that was on q_4 stays on q_4. Now you have two fingers on q_4, so remove one because you only need to remember that q_4 is a possible state at this point, not that it is possible for multiple reasons.

When the sixth and final symbol 0 is read, keep the finger on q_1 in place, move the one on q_2 to q_3, remove the one that was on q_3, and leave the one on q_4 in place. You are now at the end of the string, and you accept if some finger is on an accept state. You have fingers on states q_1, q_3, and q_4; and as q_4 is an accept state, N_1 accepts this string.

What does N_1 do on input 010? Start with a finger on q_1. After reading the 0, you still have a finger only on q_1; but after the 1 there are fingers on q_1, q_2, and q_3 (don't forget the ε arrow). After the third symbol 0, remove the finger on q_3, move the finger on q_2 to q_3, and leave the finger on q_1 where it is. At this point you are at the end of the input; and as no finger is on an accept state, N_1 rejects this input.

By continuing to experiment in this way, you will see that N_1 accepts all strings that contain either 101 or 11 as a substring.

Nondeterministic finite automata are useful in several respects. As we will show, every NFA can be converted into an equivalent DFA, and constructing NFAs is sometimes easier than directly constructing DFAs. An NFA may be much smaller than its deterministic counterpart, or its functioning may be easier to understand. Nondeterminism in finite automata is also a good introduction to nondeterminism in more powerful computational models because finite automata are especially easy to understand. Now we turn to several examples of NFAs.

EXAMPLE 1.30

Let A be the language consisting of all strings over {0,1} containing a 1 in the third position from the end (e.g., 000100 is in A but 0011 is not). The following four-state NFA N_2 recognizes A.

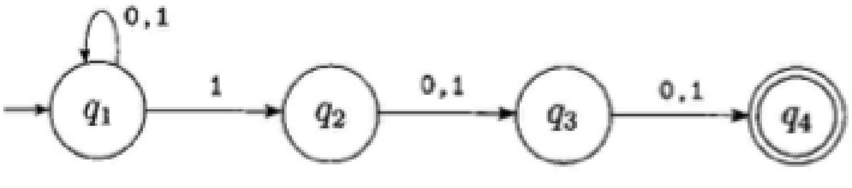

FIGURE 1.31
The NFA N_2 recognizing A

One good way to view the computation of this NFA is to say that it stays in the start state q_1 until it "guesses" that it is three places from the end. At that point, if the input symbol is a 1, it branches to state q_2 and uses q_3 and q_4 to "check" on whether its guess was correct.

As mentioned, every NFA can be converted into an equivalent DFA; but sometimes that DFA may have many more states. The smallest DFA for A contains eight states. Furthermore, understanding the functioning of the NFA is much easier, as you may see by examining the following figure for the DFA.

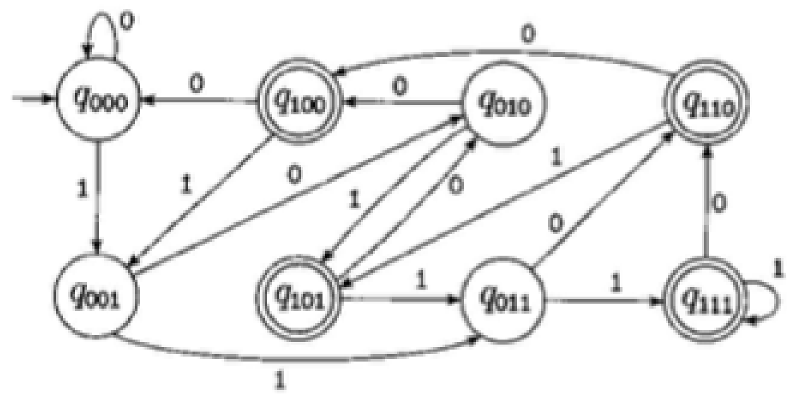

FIGURE 1.32
A DFA recognizing A

Suppose that we added ε to the labels on the arrows going from q_2 to q_3 and from q_3 to q_4 in machine N_2 in Figure 1.31. So both arrows would then have the label 0, 1, ε instead of just 0, 1. What language would N_2 recognize with this modification? Try modifying the DFA in Figure 1.32 to recognize that language.

EXAMPLE 1.33

The following NFA N_3 has an input alphabet $\{0\}$ consisting of a single symbol. An alphabet containing only one symbol is called a ***unary alphabet***.

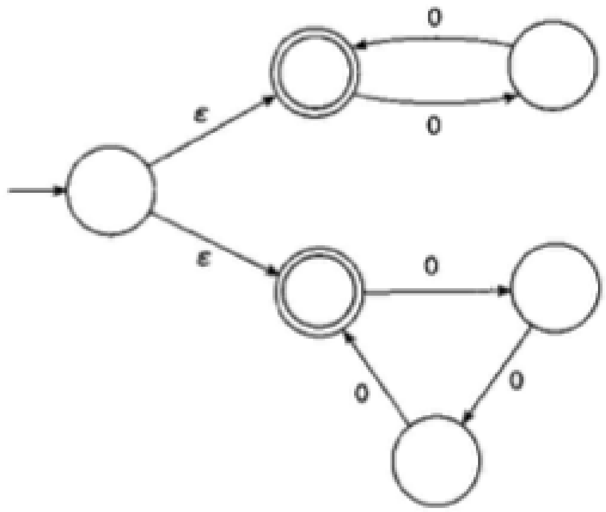

FIGURE 1.34
The NFA N_3

This machine demonstrates the convenience of having ε arrows. It accepts all strings of the form 0^k where k is a multiple of 2 or 3. (Remember that the superscript denotes repetition, not numerical exponentiation.) For example, N_3 accepts the strings ε, 00, 000, 0000, and 000000, but not 0 or 00000.

Think of the machine operating by initially guessing whether to test for a multiple of 2 or a multiple of 3 by branching into either the top loop or the bottom loop and then checking whether its guess was correct. Of course, we could replace this machine by one that doesn't have ε arrows or even any nondeterminism at all, but the machine shown is the easiest one to understand for this language.

EXAMPLE 1.35

We give another example of an NFA in Figure 1.36. Practice with it to satisfy yourself that it accepts the strings ε, a, baba, and baa, but that it doesn't accept the strings b, bb, and babba. Later we use this machine to illustrate the procedure for converting NFAs to DFAs.

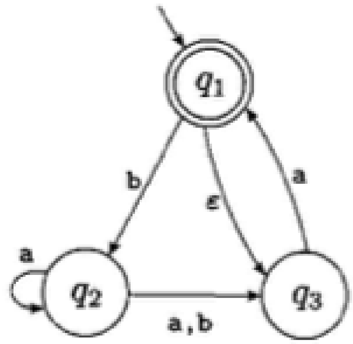

FIGURE 1.36
The NFA N_4

FORMAL DEFINITION OF A NONDETERMINISTIC FINITE AUTOMATON

The formal definition of a nondeterministic finite automaton is similar to that of a deterministic finite automaton. Both have states, an input alphabet, a transition function, a start state, and a collection of accept states. However, they differ in one essential way: in the type of transition function. In a DFA, the transition function takes a state and an input symbol and produces the next state. In an NFA, the transition function takes a state and an input symbol *or the empty string* and produces *the set of possible next states.* In order to write the formal definition, we need to set up some additional notation. For any set Q we write $\mathcal{P}(Q)$ to be the collection of all subsets of Q. Here $\mathcal{P}(Q)$ is called the ***power set*** of Q. For any alphabet Σ we write Σ_ε to be $\Sigma \cup \{\varepsilon\}$. Now we can write the formal description of the type of the transition function in an NFA as $\delta\colon Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$.

DEFINITION 1.37

A ***nondeterministic finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta\colon Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

EXAMPLE 1.38

Recall the NFA N_1:

The formal description of N_1 is $(Q, \Sigma, \delta, q_1, F)$, where

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

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

4. q_1 is the start state, and
5. $F = \{q_4\}$.

The formal definition of computation for an NFA is similar to that for a DFA. Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and w a string over the alphabet Σ. Then we say that N ***accepts*** w if we can write w as $w = y_1 y_2 \cdots y_m$, where each y_i is a member of Σ_ε and a sequence of states $r_0, r_1, \ldots, r_m$ exists in Q with three conditions:

1. $r_0 = q_0$,
2. $r_{i+1} \in \delta(r_i, y_{i+1})$, for $i = 0, \ldots, m-1$, and
3. $r_m \in F$.

Condition 1 says that the machine starts out in the start state. Condition 2 says that state r_{i+1} is one of the allowable next states when N is in state r_i and reading y_{i+1}. Observe that $\delta(r_i, y_{i+1})$ is the *set* of allowable next states and so we say that r_{i+1} is a member of that set. Finally, condition 3 says that the machine accepts its input if the last state is an accept state.

EQUIVALENCE OF NFAS AND DFAS

Deterministic and nondeterministic finite automata recognize the same class of languages. Such equivalence is both surprising and useful. It is surprising because NFAs appear to have more power than DFAs, so we might expect that NFAs recognize more languages. It is useful because describing an NFA for a given language sometimes is much easier than describing a DFA for that language.

Say that two machines are ***equivalent*** if they recognize the same language.