CSE 322 - Introduction to Formal Methods in Computer Science
Nondeterministic Finite Automata

Dave Bacon
Department of Computer Science & Engineering, University of Washington

Today we are going to talk about nondeterministic finite automata. Our motivation will be, at first, for proving that the concatenation of two regular languages is regular. Recall that the definition of the concatenation of two languages A and B:

$$A \circ B = \{wv | w \in A \text{ and } v \in B\} \tag{1}$$

Okay, so what is the difficulty in showing that the concatenation of two regular languages is regular? Well one way we might think about solving this is to dream up a DFA which accepts strings from $A \circ B$. Our first thoughts along these lines are probably: well take a machine which accepts A and then.... Well you can see the problem: somehow the machine which recognizes strings in $A \circ B$ needs to somehow know when it can break the input into first coming from A in order to start looking if the rest of the string is from B. So this seems to be a bit of a problem.

How to get around this problem? Well suppose that every time that the machine which recognizes A gets itself into an accept state it "branches off a process" which starts to check whether the rest of the string is recognized by B. Of course every time that A reaches an accept state, it should do this, which, as you can imagine, might lead to a huge number of branching off processes. This idea: that we can "branch off" a new process at some step in a finite automata's computation is the basic idea behind nondeterministic finite automata (NFA).

I. AN EXAMPLE

Consider the following state diagram Now first of all you should not that this is not a state diagram for a DFA.

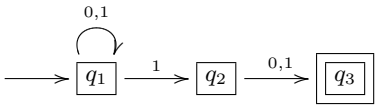

FIG. 1: This isn't a DFA!

Why? Well first of all note that state q_0 has two arrows leaving it which are labeled 1. Second, the state q_3 has *no* arrows leaving it! Indeed this is not a state diagram for a DFA but instead a state diagram for a nondeterministic finite automaton (NFA).

Okay, so how could we possibly think about what the state diagram in Figure 1 really represents? Well lets begin to evaluate it like it was a DFA and see what happens. To begin with, there is only one start state, so we can assume that we start in this state, q_1. Now if we are evaluating the action of the machine on 0, there is only one transition from q_1, the one that goes back to itself. But if the input to the machine is 1, there are *two* outgoing arrows. Oh noes! What to do? Well you can imagine that you follow both paths...at the same time! Now that's just crazy talk you say. And I say yes, crazy talk, but as long as it is consistent crazy talk, that is fine! So let's imagine that every time our machine encounters more than one outgoing arrow for a given input, the machine "splits" into different incarnations which follow each of the arrows to new states.

Another complication, comes along when we end up in state q_3. At q_3 there are no arrows leaving this state. Therefore we have to ask ourselves: what happens when we encounter a state and input such that we have no where to go? Well above we sort of fudge-ly said that our machine split into multiple instantiations when we encountered too many arrows. So what to do when we encounter no arrows? Well end that particular computation of course!

So now we have some informal idea about how we can conceive of a strange machine which computes using the above example. Let's go through a simple example just to illustrate how this might work. Suppose that we ask the machine we are conceiving to evaluate 110. What happens? Well first of all we start in state q_1. Now we read the first input character, which is a 1. Now, informally, we can imagine that the machine splits into two state separate incarnations following each of the two 1 labelled arrows from q_1. Thus we image that the machine is at q_1 (by following the loop) and at q_2. The next input is 1. So now we need to evaluate what happens to both q_1 and q_2 when we

receive as input a 1. q_1 again will branch into two separate states, q_1 and q_2. q_2 on the other hand, will transition to q_3. Thus we now imagine that our machine has split into three copies, one in each of the three states, q_1, q_2, q_3. The final input is 0. q_1 on 0 transitions to q_1. q_2 on 0 transitions to q_3. But what happens to q_3? Well there is no arrow going away from q_3, so q_3 simply ends. Thus we see that after the input 110 is read, the machine is in the state q_2 and q_3. Now since one of these is an accept state (q_3) we say that the machine accepts 110. A helpful way to picture the above process is to draw a tree which includes the bifurcation and death of processes like the one in Figure 2.

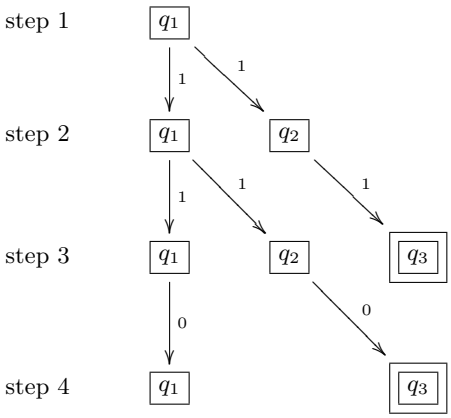

FIG. 2: A NFA Tree

Another twist in NFAs is that we now allow edges to be labeled by the empty string ε (note that this is not the empty set!) When a NFA encounters a state with the empty string coming from it, the machine simply makes a copy of itself For example, consider this NFA:

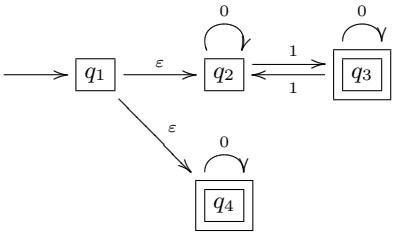

FIG. 3: Another NFA

Now, what happens when this NFA starts computing? Well we see that the first state q_1 has two arrows branching from it which are both εs. Well, the NFA starts in q_1, and then immediately creates copies of itself which follow the ε arrows to q_2 and q_3. Thus before any input, the state of the machine is the three states q_1, q_2, and q_3. Now when the first input of the string is read, the machine which was spawned at q_1 dies (there are no arrows with the labels 0 or 1 coming from q_1.) Then the two machines at the top proceed on their marry way, as if they had been started in q_2 and q_4. Thus, examining the top machine, we see that it accepts when the string has an odd number of 1s. The bottom machine accepts only strings of the form 0^k. Thus the above NFA has a language made up of all string which have an odd number of 1s or are of the form 0^k.

II. FORMAL DEFINITION OF A NONDETERMINISTIC FINITE AUTOMATA

The formal definition of a DFA was given by the five tuple $(Q, \Sigma, \delta, q_0, F)$. For a NFA all of these will be exactly the same as before except one! Now the transition function will be a much more interesting function. In particular it will be defined as

$$\delta : Q \times \Sigma_\varepsilon \to P(Q) \tag{2}$$

But what are these newfangled thingamabobs Σ_ε and $P(Q)$. Well the first is rather simple, $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$. In other words we now allow transitions to be labeled not just by elements of the alphabet, but also by the empty string ε. What is that second thing, $P(Q)$? $P(Q)$ is the power set of Q. That is $P(Q)$ is the set of subsets of Q. This is the part which takes care of the fact that NFAs can, for a given alphabet letter, go to no new state, one new state, or any number of new states. Thus the transition function now takes as input a state and an element of the alphabet or the empty string, and maps this onto a set of new states.

For example, the formal definition of the NFA given above,

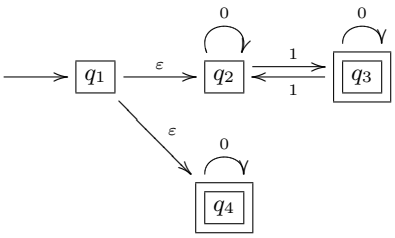

FIG. 4: The NFA whose formal definition is given before

is given by

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

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

4. The start state is $q_0 = q_1$.
5. The accept states are $F = \{q_3, q_4\}$.

The formal definition of computation for a NFA is, as you could surely have guessed, also slightly different. Let $N = (Q, \Sigma, \delta, q_0, F)$ be a NFA and w a string over Σ. We say that N accepts w if we can write w as $w = y_1 y_2 \dots y_m$ where each y_i is a member of Σ_ε (note that ε!) and a sequence of states $r_0, r_1, \dots, r_m$ exists in Q which satisfies

1. $r_0 = q_0$ (The start state is q_0.)
2. $r_{i+1} \in \delta(r_i, y_{i+1})$. Note here, in particular the $\in$. This $\in$ implies that the next state only be in the set of possible next states.
3. $r_m \in F$. (The last state is an accept state.)

Of course the most important difference here is that the transition function now returns a set. Thus any set of states which satisfies the proper update conditions, sampled from the proper sets along the way, will allow for an NFA to accept. Note that this does not preclude multiple states being accepting states and the machine having a path to many of these as well.