#1 Sipser §1.2: Nondeterminism

Nondeterminism has two common interpretations. Consider a fork in a road. One idea is to consider what happens if we take both paths. That's the notion of nondeterminism we will consider in this class. The other interpretation is to consider tossing a coin and picking a result based on the toss's outcome. We will not consider that notion in this class.

Nondeterminism can sound and feel scary because it makes things more complicated. This is true. But I hope I do not need to argue to you why it is and remains a popular idea. It is everywhere in practical computation: parallel computing, distributed computing, and so forth. One could argue parallelism is the first tool many grab in the make-it-run-faster shed. For this reason, you may be surprised to hear that deterministic finite automata and nondeterministic finite automata are equivalent in computational power. That is to say, they recognize the same class of languages (regular languages).

There's a bit of a trick to the above: they are equivalent in *what they can express*. We do not have yet (in this course) even the vocabulary to think on what it means for them to be equivalent in *how quickly* or *easily* they compute. That will come in the last part of this course.

#1.1 NFAs—a formal definition

Definition 1 (Nondeterministic Finite Automaton)**.** *A* Nondeterministic Finite Automata*, or* DFA*, is formally a 5-tuple* $(Q, \Sigma, \delta, q_0, F)$ *where*

1. Q *is a finite set of states;*
2. Σ *is a finite set called the* alphabet*;*
3. $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ *is called* transition function*;*
4. $q_0 \in Q$ *is the* start state*;*
5. $F \subseteq Q$ *is a set of* accept states*.*

Notice the difference is only in δ. Let's discuss each difference now.

- Instead of accepting the input $Q \times \Sigma$, δ now accepts inputs from $Q \times \Sigma_\epsilon$, where $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$. This is another way of saying that we have added the empty string to our list of valid characters on which we transition. .This is also why NFAs will have edges labeled with ϵ.
- Instead of outputting elements from Q we now output sets in $\mathcal{P}(Q)$. This is math-speak for "we return a set of different options." The main aspect of nondeterminism is here: when I come to a fork in a road, I may specify *many* paths, not just one.

#1.1.1 nondeterministic computation: a formal definition

We modify the definition of computation slightly.

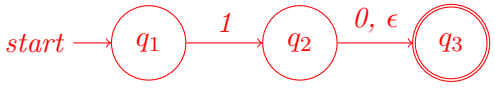
Definition 2 (Computation (NFAs)). *Let M be a finite automaton and let $w = w_1 w_2 ... w_n$ be a string where each $w_i \in \Sigma_\epsilon$. We say that M* accepts *w if a sequence of states $r_0, r_1, ... r_n$ exists in Q with three conditions:*

1. $r_0 = q_0$,
2. $r_{i+1} \in \delta(r_i, w_{i+1})$ *for* $i = 0, ..., n-1$, *and*
3. $r_n \in F$

Observe that this is the same definition as for DFAs except that

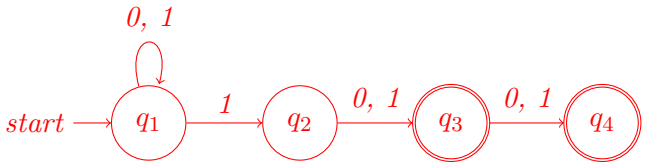
1. strings may contain the empty string ϵ as a component,
2. we check if the transitioned state r_{i+1} is in the *set of allowed transitions* $\delta(r_i, w_{i+1})$.

Example 1. *Let's consider the NFA from the quiz.*



On the quiz, I asked you if 1 is accepted by this NFA. The answer is yes, even though q_1 does not lead directly to q_2. As hinted, this is because the definition of computation above allows $w = 1\epsilon$ as input.

Example 2.



Observe that q_1 has a 1-labeled transition to both *the states q_1 and q_2. (This is the nondeterministic bit.) What strings are accepted by this NFA? We see that the string must have length at least 3.*

Try:

1. 110,
2. 111,
3. 000000011.

Observe that only the second one works. In general, this NFA accepts only strings containing a 1 *in the third position to last.*

#1.2 NFA and DFA equivalence

As I hinted at before, NFAs and DFAs are equivalent. In particular, every DFA is trivially an NFA. So consider the harder claim:

Theorem 1. *Every nondeterministic finite automaton has an equivalent deterministic finite automaton.*

And this necessarily implies

Theorem 2. *A language is regular if and only if some nondeterministic finite automaton recognizes it.*

Proof.

Given NFA $N = (Q, \Sigma, \delta, q_0, F)$ that recognizes language L, we will show a DFA $D = (Q', \Sigma, \delta', q_0', F')$ that too recognizes L. Let's consider each component.

1. $Q' = \mathcal{P}(Q)$, or: Each state in Q' is a collection of states in Q.
2. $q_0' = \{q_0\}$.
3. $F' = \{R \in Q' \mid \exists r \in R \text{ s.t. } r \in N\}$. In other words, we accept any *collection of states* R that has at least one of those states accepted by N.

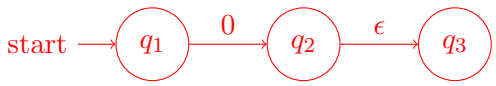
Now, to transform δ to δ' we have to define $\delta'(R, s)$ on each input $R \in Q'$ (that is, $R \subseteq Q$) to equal some state $S \subseteq Q$. But this is pretty much what δ does already. Thus define δ' as

$$\delta'(R, s) = \bigcup_{r \in R} \delta(r, s)$$

In other words, $\delta'(R, s)$ takes you to *any* state that the states $r \in R$ are taken to by input s.

Finally, we have to consider funny-business with ϵ arrows.

Suppose, for purpose of illustration, that $\delta(q_1, 0) = \{q_2\}$ for some NFA. When we try to build a DFA from these pieces, it is not enough to stop here, because q_1 might reach q_3 along a path like this:



(Recall Quiz 2.) So, instead of defining $\delta'(q_1, 0) = \{q_2\}$ for deterministic transition function δ', we need to be a bit more clever. In particular, we map the pair $(q_1, 0)$ not just to what is reached by q_1 along 0-arrows but also what can be reached by 0 or more ϵ-arrows. The definition below captures this intuition.

Definition 3 (Epsilon Closure)**.** *Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and define*

$$E(R) = \{q \mid q \text{ can be reached from } R \text{ by traveling along 0 or more } \epsilon\text{-arrows}\},$$

for some subset R of Q. Then we call $\delta' : \mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$ the Epsilon Closure *of δ if*

$$\delta'(R, a) = \{q \in Q \mid q \in E(\delta(r, a)) \text{ for some } r \in R\}$$

□

#1.3 Regular Operations

An important part of this class is asking what benefit a formalization gives us. Our first results will be over what are called *regular operations* and *closure properties*. The premise is: we start with regular languages, A, B, C, and so forth. What can we do to them such that they stay regular? This will look familiar to what you can do with regular expressions, which is no coincidence.

The following operations are called *regular*:

1. Union: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.
2. Concatenation: $A \circ B : \{xy \mid x \in A \text{ and } y \in B$.
3. Star (aka the *Kleene Closure*): $A^* = \{x_1 x_2 ... x_k \mid k \geq 0, x_i \in A\}$.

Note that the star operator above is over *languages*, not *alphabets*. This means that we are taking all finite strings formed from the *concatenation* of strings, not the sequencing of characters. A subtle difference.

Example 3. *Let* $A = \{\text{"good"}, \text{"bad"}\}$ *and* $B = \{\text{"boy"}, \text{"girl"}\}$. *Then*

$$A \cup B = \{\text{"good"}, \text{"bad"}, \text{"boy"}, \text{"girl"}\}$$
$$A \circ B = \{\text{"goodboy"}, \text{"badboy"}, \text{"goodgirl"}, \text{"badgirl"}\}$$
$$A^* = \{\epsilon, \text{"good"}, \text{"bad"}, \text{"goodgood"}, \text{"goodbad"}, \text{"badbad"}, ...\}$$

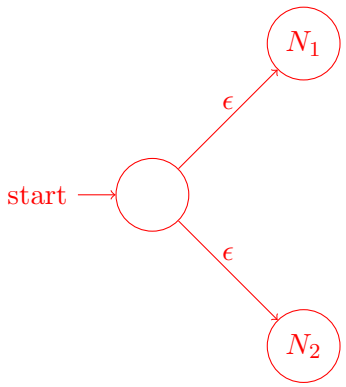
Finally, we want to show that these operations preserve regularity. This style of claim (and proof) will be exceedingly common in this class.

Claim 1 (Union Regularity)**.** *Let* L_1 *and* L_2 *be regular languages. Then their union* $A \cup B$ *is regular.*

Sipser §1.1 gives a proof of this claim using DFAs. That proof has the following structure:

1. Observe that, because L_1 and L_2 are regular, we must have DFAs D_1 and D_2 that recognize L_1 and L_2.
2. Using these DFAs, construct a *new* DFA D'. Prove that D' recognizes $L_1 \cup L_2$.
3. Conclude: as $L_1 \cup L_2$ is accepted by some DFA, it is by definition regular.

However, it's a bajillion times easier using NFAs. By the above, the languages L_1 and L_2 each have DFAs D_1 and D_2 such that are equivalent to NFAs N_1 and N_2, respectively. Now, do this:



where N_1 and N_2 are placeholders for their respective NFAs. We are simply letting input strings run on *both* N_1 and N_2 and accepting whenever *any* path is accepted. Kinda neat.