12.1 Nondeterminism

The idea of "nondeterministic" computations is to allow our algorithms to make "guesses", and only require that they accept when the guesses are "correct". For example, a simple nondeterministic polynomial-time algorithm to decide whether a number N is composite would nondeterministically guess a factorization L, M of the number, and then verify that $L \cdot M = N$. (It turns out that there is also a deterministic polynomial-time algorithm for deciding compositeness, discovered in 2002, but it is much more complicated.)

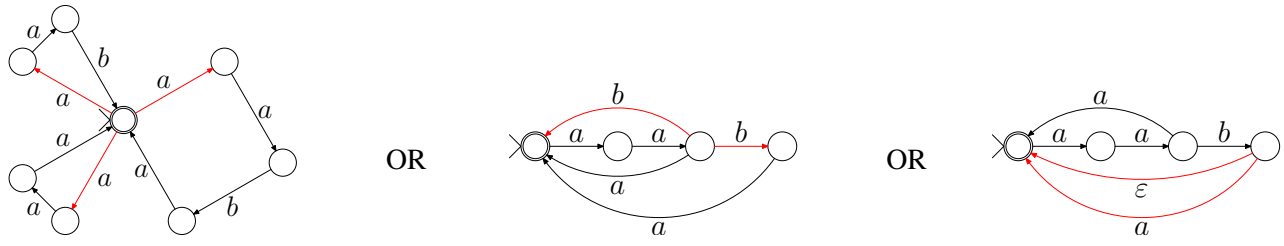
Nondeterminism is not a realistic or physical computational resource, but turns out to be very useful for capturing many computational problems of interest and better-understanding realistic deterministic models of computation. Just like introducing the imaginary number $i = \sqrt{-1}$ turns out to be very useful in answering questions about the ordinary real numbers.

12.2 Nondeterministic Finite Automata

A language for which it is hard to design a DFA:

$$L = \{aab, aaba, aaa\}^* = \{x_1 x_2 \cdots x_k : k \geq 0 \text{ and each } x_i \in \{aab, aaba, aaa\}\}.$$

But it is easy to imagine a "device" to recognize this language if there sometimes can be several possible transitions!



Def: An NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q, Σ, q_0, F are as for DFAs
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to P(Q)$.

When in state p reading symbol σ, can go to any state q in the set $\delta(p,\sigma)$.

- there may be more than one such q, or
- there may be none (in case $\delta(p,\sigma) = \emptyset$).

Can "jump" from p to any state in $\delta(p,\varepsilon)$ without moving the input head.

Computations by an NFA

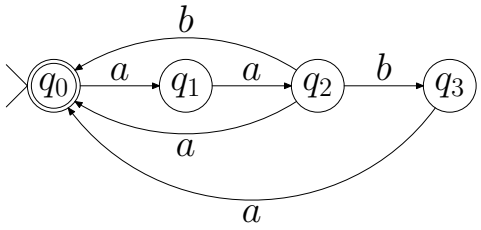
$N = (Q,\Sigma,\delta,q_0,F)$ accepts $w \in \Sigma^*$ if we can write $w = y_1 y_2 \cdots y_m$ where each $y_i \in \Sigma \cup \{\varepsilon\}$ and there exist $r_0,\ldots,r_m \in Q$ such that

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

Nondeterminism: Given N and w, the states $r_0,\ldots,r_m$ are not necessarily determined.

Example of an NFA

N :

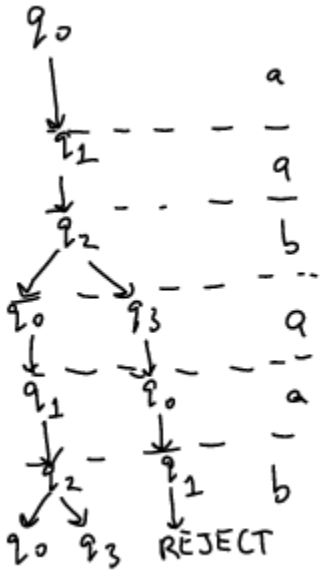


$N = (\{q_0,q_1,q_2,q_3\},\{a,b\},\delta,q_0,\{q_0\})$, where δ is given by:

	a	b	ε
q_0	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\{q_2\}$	$\emptyset$	$\emptyset$
q_2	$\{q_0\}$	$\{q_0,q_3\}$	$\emptyset$
q_3	$\{q_0\}$	$\emptyset$	$\emptyset$

Tree of computations

Tree of computations of NFA N on string *aabaab*:



An NFA N accepts w if there is at least one accepting computation path on input w, so we could check all computation paths to determine whether N accepts w. But the number of paths may grow exponentially with the length of w! Can the exponential search be avoided?

NFAs vs. DFAs

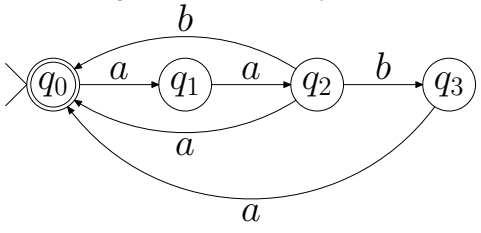
NFAs seem more "powerful" than DFAs. Are they?

Theorem 12.1 *For every NFA N, there exists a DFA M such that* $L(M) = L(N)$.

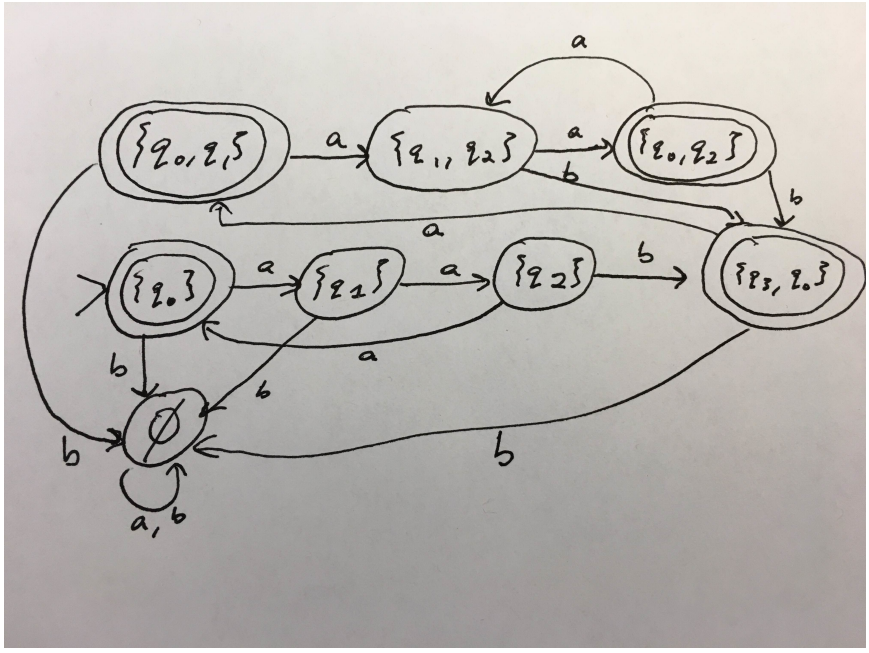
Proof by Construction: Given any NFA N, we construct a DFA M such that $L(M) = L(N)$. The idea is to have the DFA M keep track of the *set* of states that N could be in after having read the input string so far.

Before writing it down formally, we illustrate with an example. Recall our NFA N for $L = \{aab, aaba, aaa\}^*$.

N :



N starts in state q_0 so we will construct a DFA M starting in state $\{q_0\}$:



Formal Description of the Subset Construction

Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$, we construct a DFA $M = (Q', \Sigma, \delta', q_0', F')$ where

$$\begin{aligned} Q' &= P(Q) \\ q_0' &= E(\{q_0\}) \\ F' &= \{R \subseteq Q : R \cap F \neq \emptyset\} \text{ (that is, } R \in Q') \\ \delta'(R, \sigma) &= E(\{q \in Q : q \in \delta(r, \sigma) \text{ for some } r \in R\}) \\ &= \bigcup_{r \in R} E(\delta(r, \sigma)), \end{aligned}$$

where for a set $S \subset Q$, $E(S)$ is the set of states that can be reached starting from a state in S and following 0 or more ε transitions.

It can be shown by induction on $|w|$ that for every string w, running M on input w ends in the state $\{q \in Q :$ some computation of N on input w ends in state $q\}$.

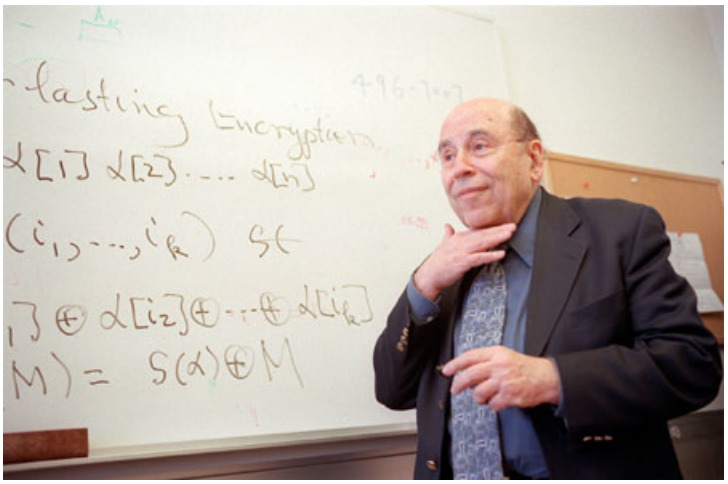
Rabin & Scott, "Finite Automata and Their Decision Problems," 1959

1976 – Michael O. Rabin ***See the ACM Author Profile in the Digital Library***

Citation
For their joint paper "Finite Automata and Their Decision Problem," which introduced the idea of nondeterministic machines, which has proved to be an enormously valuable concept. Their (Scott & Rabin) classic paper has been a continuous source of inspiration for subsequent work in this field.

Biographical Information

Michael O. Rabin (born 1931 in Breslau, Germany) is a noted computer scientist and a recipient of the Turing Award, the most prestigious award in the field.

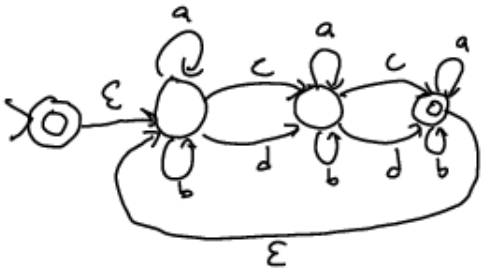


Using NFAs for Pattern Recognition

NFAs can express quite complicated pattern-recognition problems. Indeed, it is easy to construct an NFA N that accepts exactly the strings generated by any given *regular expression*, such as

$$R = ((a \cup b)^*(c \cup d)(a \cup b)^*(c \cup d)(a \cup b)^*)^*.$$

This regular expression R generates the set $L(R)$ of strings over alphabet $\Sigma = \{a, b, c, d\}$ that have an even number of occurrences of "c" or "d". We can easily convert R (or any regular expression, for that matter) into an NFA N such $L(N) = L(R)$:



Formally, a "regular expression" is a language defined inductively via the following rules:

(1) $\{\sigma\}$ is a regular expression for any $\sigma \in \Sigma$

(2) $\{\varepsilon\}$ is a regular expression, where ε is the empty string

(3) $\emptyset$ is a regular expression

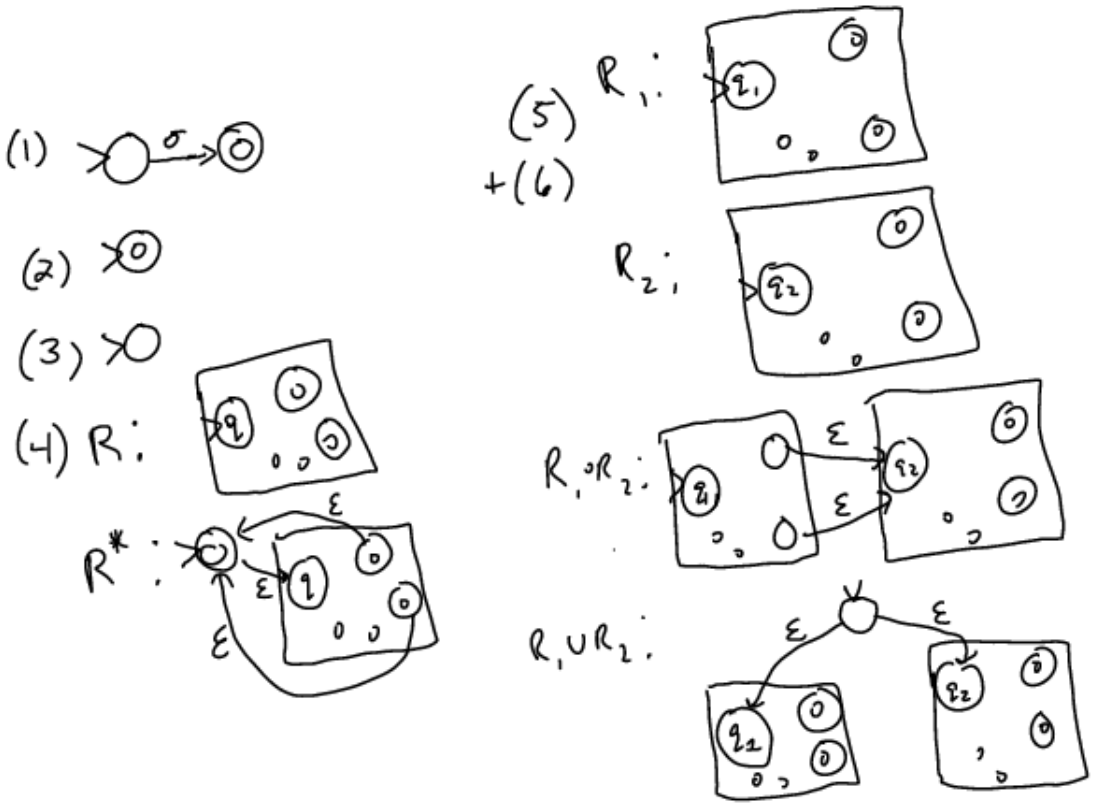
(4) If R is a regular expression, then so is $R^* = \{x_1 \dots x_k : k \geq 0, \forall i\ x_i \in R\}$

(5) If R_1, R_2 are regular expression, then so is $R_1 \circ R_2 = \{x_1 x_2 : x_1 \in R_1, x_2 \in R_2\}$

(6) If R_1, R_2 are regular expressions, then so is $R_1 \cup R_2 = \{x : x \in R_1, \text{ or } x \in R_2\}$

It turns out that L is a regular expression iff L is regular (i.e. some DFA accepts it). Since DFAs can simulate NFAs, it is equivalent to say that L is a regular expression iff some NFA accepts it.

One direction of the proof is more straightforward: namely to show that any regular expression is the language accepted by some NFA N. Those NFAs can be created as in the image below (for (4)-(6), we show how to use NFAs for R, R_1, R_2 to construct a new NFA after applying some rule).



The converse is also true (but harder to prove): for every DFA M, one can construct a regular expression R such that $L(R) = L(M)$. So DFAs, NFAs, and Regular Expressions all describe exactly the same set of languages! If you're interested in the full proof, see the recommended text by Sipser. The basic idea is to define something called

a *GNFA (generalized nondeterministic finite automaton)*, which is like an NFA except that edges can be labeled with arbitrary regexps and not just $\Sigma \cup \{\varepsilon\}$. We insist upon a GNFA of a specific format:

- There is only one start and one accept state.
- The start state has no incoming edges, and has outgoing edges to every other state.
- The accept state has no outgoing edges, and has incoming edges from every other start state.
- All states other than the start and accept states have all the possible $(|Q|-2)^2$ edges to each other, in addition to each one of them having a self loop.

Given a DFA M, we can convert it to this format quite easily. First, we make a new accept state such that every other accept state has an ε-transition to it, and all other non-accept states have $\emptyset$ transitions to it. We also make a new start state with an ε-transition to the original start state, and with $\emptyset$ transitions to all other states. Then for each other state, we add a self-loop with a ε-transition. Also if states q, q' other than the accept/start states had no edge from q to q' before, then we add one with edge label $\emptyset$. Now we obtain a GNFA M' accepting exactly $L(M)$. The main idea is this: if M' has exactly $k = 2$ states, then we are done! This is because there is a single edge from the start to accept state, and we can simply read off the regexp written as its label. Otherwise, if $k > 2$, there is some state q^* which is neither the accept nor the start state. Then we will remove q^* from the GNFA to obtain one with one less state. We have to alter existing edges to make sure the language accepted by the GNFA doesn't change. Suppose q, q' are two other states (we may have $q = q'$). Suppose the edge from q to q' is labeled R_4, and q to q^* is labeled R_1, q^* has a self-loop labeled R^2 and q^* to q' is labeled R_3. Then we replcae the edge from q to q' with a new edge labeled $R_1 R_2^* R_3 \cup R_4$, since we could either get from q to q' through q^*, or not using q^*. Induction shows that this works.

So to decide whether a given string $w \in \Sigma^*$ matches a given regular expression R, we can convert R to an NFA N, convert N to a DFA M, and then run M on R.

Q: What's the problem with this approach? How can we do better?

Theorem 12.2 *Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$ and a string w, we can decide whether $w \in L(N)$ in time $O(|Q|^2 \cdot |w|)$.*

Proof: For a subset $R \subseteq Q$ of states, recall the definition of $E(R)$ above as the set of states reachable from those in R using only ε-transitions. Note $E(R)$ can be computed in time $O(|Q|^2)$ using, say, breadth-first search, since the underlying graph has $|Q|$ vertices and at most $|Q|^2$ edges. We initialize $R_0 = E(\{q_0\})$. Then for each $i = 1, \ldots, |w|$,

we can compute $R_i = \cup_{q \in R_{i-1}} E(\delta(q, w_i))$, which is the set of states our NFA could possibly be in after processing the prefix $w_1 \ldots w_i$. Then N accepts w iff $R_{|w|} \cap F$ is non-empty.