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 \( L \cdot M \) 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!

\[
\begin{align*}
&\text{Def: An} \text{ NFA is a 5-tuple } (Q, \Sigma, \delta, q_0, F), \text{ where} \\
&\quad \cdot Q, \Sigma, q_0, F \text{ are as for DFAs} \\
&\quad \cdot \delta : Q \times (\Sigma \cup \{\varepsilon\}) \to P(Q).
\end{align*}
\]
When in state $p$ reading symbol $\sigma$, 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 $w = y_1y_2\cdots y_m$ 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**

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

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
<th>$\varepsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>${q_1}$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$q_1$</td>
<td>${q_2}$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$q_2$</td>
<td>${q_0}$</td>
<td>${q_0, q_3}$</td>
<td>0</td>
</tr>
<tr>
<td>$q_3$</td>
<td>${q_0}$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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 $L = \{aab, aaba, aaa\}^*$.

$N$ 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

$$Q' = P(Q)$$
$$q'_0 = E(\{q_0\})$$
$$F' = \{ R \subseteq Q : R \cap F \neq \emptyset \} \text{ (that is, } R \in Q')$$
$$\delta'(R, \sigma) = E(\{ q \in Q : q \in \delta(r, \sigma) \text{ for some } r \in R \})$$
$$\quad = \bigcup_{r \in R} E(\delta(r, \sigma)),$$

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

$$R = ((a \cup b \cup c \cup \cdots \cup z)^* (foo \cup bar)(a \cup b \cup c \cup \cdots \cup z)^* (foo \cup bar))^* \cup (a \cup b \cup c \cup \cdots \cup z)^*.$$ 

This regular expression $R$ generates the set $L(R)$ of strings over alphabet $\Sigma = \{a, b, \ldots, z\}$ that have an even number of non-overlapping occurrences of “foo” or “bar”. We can easily convert $R$ (or any regular expression, for that matter) into an NFA $N$ such $L(N) = L(R)$:

[The converse is also true (but harder to prove): for every NFA $N$, one can construct a regular expression $R$ such that $L(R) = L(N)$. 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.]

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:**