Nondeterministic Finite Automata

Mahesh Viswanathan

1 Introducing Nondeterminism

Consider the machine shown in Figure 1. Like a DFA it has finitely many states and transitions labeled by symbols from an input alphabet (in this case $\{0, 1\}$). However, it has important differences when compared with the DFA model we have seen.

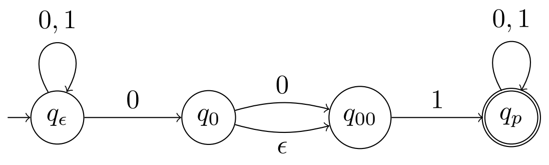

Figure 1: Nondeterministic automaton N

- State q_ϵ has two outgoing transitions labeled 0.
- States q_0, and q_{00} have missing transitions. q_0 has no transition labeled 1, while q_{00} has no transition labeled 0.
- State q_0 has a transition that is labeled not by an input symbol in $\{0, 1\}$ but by ϵ.

This is an example of what is called a *nondeterministic finite automaton (NFA)*. Intuitvely, such a machine could have many possible computations on a given input. For example, on an input of the form $u001v$, it is possible for the machine to reach the accepting state q_p by transitioning from q_ϵ to q_0 after reading u. Similarly, it is possible for the machine to reach q_p also on the input $u01v$ — for this to happen, the machine stays in q_ϵ as it reads u, transitions to q_0 on reading 0 after u, transitions to q_{00} *without* reading an input symbol by following the transition labeled ϵ, goes to q_p on reading 1, and stays in q_p while reading v. On the other hand, the machine also have other possible computations on both $u001v$ and $u01v$ — it may stay in q_ϵ and never transition out of it; or it may transition to q_{00} on reading $u0$ by following the ϵ-transition from q_0 and *die* attempting to take a transition labeled 0 (that doesn't exist) out of q_{00}. The fact that the machines behavior is *not determined* by the input string, is the reason these machines are called *nondeterministic*.

1.1 Nondeterministic Finite Automata (NFA)

NFAs differ from DFAs in that (a) on an input symbol a, a given state may of 0, 1, or more than 1 transition labeled a, and (b) they can take transitions without reading any symbol from the input; these are the ϵ-transitions [1]. These features are captured in the following formal definition of an NFA.

Definition 1. *A* nondeterministic finite automaton *(NFA) is a* $M = (Q, \Sigma, \delta, s, A)$ *where*

- *Q is a finite set whose elements are called states,*

[1]**Beware:** ϵ-transitions are not transitions taken on the symbol "ϵ". ϵ is *not* a symbol! They are transitions that are taken *without* reading any symbol from the input.

- *Σ is a finite set whose elements are the symbols used to encode the input,*
- *$\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$ is the transition function, where $\mathcal{P}(Q)$ is the powerset of Q,*
- *$s \in Q$ is the start state,*
- *$A \subseteq Q$ is the set of accepting states.*

For example, the NFA in Figure 1 can be described in the above formal notation as follows.

- Set of states $Q = \{q_\epsilon, q_0, q_{00}, q_p\}$
- Input alphabet $\Sigma = \{0, 1\}$
- Start state $s = q_\epsilon$
- Accepting states $A = \{q_p\}$, and
- The transition function δ given as

$$\begin{array}{lll} \delta(q_\epsilon, 0) = \{q_\epsilon, q_0\} & \delta(q_\epsilon, 1) = \{q_\epsilon\} & \delta(q_0, 0) = \{q_{00}\} \\ \delta(q_0, \epsilon) = \{q_{00}\} & \delta(q_{00}, 1) = \{q_p\} & \delta(q_p, 0) = \{q_p\} \\ \delta(q_p, 1) = \{q_p\} & & \end{array}$$

  with $\delta(p, a) = \emptyset$ in all other cases, not listed above.

1.2 Computation and Acceptance

There are two useful ways to think about nondeterministic computation. Both these views are mathematically equivalent, and in certain contexts, one view maybe more convenient than the other.

Parallel Computation View. At each step, the machine "forks" a thread corresponding to one of the possible next states. If a state has an ϵ-transition, then you fork a new process for each of the possible ϵ-transitions, without reading any input symbol. If the state has multiple transitions on the current input symbol read, then fork a process for each possibility. If from current state of a thread, there is no transition on the current input symbol then the thread dies. After reading all input symbols, if there is some active, live thread of the machine that is an accepting state, the input is accepted. If none of the active threads (or if there are no active threads) is in an accept state, the input is rejected. This view is shown in Figure 2 which describes the computation of the NFA in Figure 1 on input 0100.

Guessing View. An alternate view of nondeterministic computation is that the machine *magically* chooses the next state in manner that leads to the NFA accepting the input, unless there is no such choice possible. For example, again for the NFA in Figure 1 and input 0100, the machine (in this view) will magically choose the following sequence of steps that leads to acceptance.

$$q_\epsilon \xrightarrow{0} q_0 \xrightarrow{\epsilon} q_{00} \xrightarrow{1} q_p \xrightarrow{0} q_p \xrightarrow{0} q_p$$

These different views of nondeterministic computation can be captured mathematically through the notion of computation and acceptance. Central to this enterprise will be to define when an NFA $M = (Q, \Sigma, \delta, s, A)$ has an active thread of computation that starts in state p and ends in state q on input w; we denote this (like in the case of DFAs) as $p \xrightarrow{w}_M q$.

Definition 2. *Formally, $p \xrightarrow{w}_M q$ if there is a sequence of states $r_0, r_1, \ldots r_k$ and a sequence $x_1, x_2, \ldots x_k$, where for each i, $x_i \in \Sigma \cup \{\epsilon\}$, such that*

- *$r_0 = p$,*

$$\begin{tikzcd}
 & & q_\epsilon \arrow[dl, "0"'] \arrow[d, "0"'] & & \\
q_{00} \arrow[d, "1"'] & q_0 \arrow[l, "\epsilon"'] \arrow[d, "1"'] & q_\epsilon \arrow[d, "1"'] & & \\
q_p \arrow[d, "0"'] & \text{X} & q_\epsilon \arrow[d, "0"'] \arrow[dr, "0"] & & \\
q_p \arrow[d, "0"'] & & q_\epsilon \arrow[dl, "0"'] \arrow[d, "0"'] & q_0 \arrow[r, "\epsilon"] \arrow[d, "0"'] & q_{00} \arrow[d, "0"'] \\
q_p & q_0 \arrow[d, "\epsilon"'] & q_\epsilon & q_{00} & \text{X} \\
 & q_{00} & & &
\end{tikzcd}$$

Figure 2: Computation of NFA in Figure 1 on 0100. X denotes a thread dying because of the absence of any transtions. The input 0100 is accepted because there is an active thread (namely the leftmost one) that is in the accepting state q_p.

- *for each i, $r_{i+1} \in \delta(r_i, x_{i+1})$,*
- *$r_k = q$, and*
- *$w = x_1x_2x_3 \cdots x_k$*

Thus, a computation from p to q on input w is a sequence of states $r_0, \ldots r_k$ and transition labels $x_1 \ldots x_k$ such that the first state r_0 is p (condition 1), the last state r_k is q (condition 3), r_i is that state reached after i steps, while x_i is the label of the ith transition taken in this computation. Some of the steps in the computation could be taken without reading any input symbol (i.e., x_i maybe ϵ) but the requirement is that if concatenate all the symbols read during the computation in order then it constitutes the input string w (condition 4).

To see how this definition captures the intuition of an NFA computation, consider the example NFA shown in Figure 1. We can show that $q_\epsilon \xrightarrow{0100}_N q_p$ by taking $r_0 = q_\epsilon$, $r_1 = q_0$, $r_2 = q_{00}$, $r_3 = q_p$, $r_4 = q_p$, $r_5 = q_p$, and $x_1 = 0$, $x_2 = \epsilon$, $x_3 = 1$, $x_4 = 0$, $x_5 = 0$. This choice of states (r_is) and transition labels (x_is) satisfies the 4 conditions of a computation; in particular $x_1x_2 \cdots x_5 = 0\epsilon100 = 0100$. We can also show that $q_\epsilon \xrightarrow{0100}_N q_\epsilon$ by choosing $r_0 = r_1 = r_2 = r_3 = r_4 = q_\epsilon$, and $x_1 = 0$, $x_2 = 1$, $x_3 = 0$, and $x_4 = 0$; in this second computation we do not take any ϵ-transitions.

Though the definition of $p \xrightarrow{w}_M q$ for NFAs is remarkably similar to that for DFAs there are some differences. The number of steps taken by a DFA is equal to the length of w as in each step one symbol is read from the input. This is reflected in the fact that the sequence of states constituting the DFA computation is one more than the length of w. For NFAs, however, since there can be steps that don't read any input symbol, the length of the computation (k = number of transitions) is *at least* $|w|$ (rather than equal to $|w|$).

Like in the case of DFAs, it will be useful to introduce the function $\delta^*_M(p, w)$ that captures the behavior of machine M on input string w when started in state p. It will informally be the states of all active threads (in the parallel view) of M after reading all symbols in w. More precisely, the definition looks very similar to that for DFAs.

Definition 3. *$\delta^*_M(p, w) = \{q \in Q \mid p \xrightarrow{w}_M q\}$.*

It is useful to contrast Definition 3 with the corresponding one for DFAs.

1. In a DFA, because computation was completely determined by the input, there is a unique state the machine could be in after an input. This is reflected in the fact that δ^*_M for DFA returns a single state.

In contrast, for NFAs δ^*_M returns a set of possible states. In general, for a state p and input w, for an NFA, $\delta^*_M(p, w)$ could be $\emptyset$ (no active threads), $\delta^*_M(p, w)$ could be a set of size 1 (all active threads in the same state), or have size more than 1 (active threads in more than 1 state). To see this, consider the NFA N from Figure 1. $\delta^*_N(q_{00}, 0) = \emptyset$, $\delta^*_N(q_{00}, 1) = \{q_p\}$, and $\delta^*_N(q_\epsilon, 0) = \{q_\epsilon, q_0, q_{00}\}$.

2. Next, because a DFA cannot take any steps without reading a symbol from the input, we had $\delta^*_M(p, \epsilon) = p$ for every p. In contrast, because an NFA can take steps without reading an input symbol, $\delta^*_M(p, \epsilon)$ may contain more than just p. For example, for NFA N (Figure 1), $\delta^*_N(q_0, \epsilon) = \{q_0, q_{00}\}$

3. Finally, in a DFA, when $|w| = 1$, $\delta^*_M(p, w) = \delta(p, w)$. This also does not hold in NFAs because of the presence of ϵ-transitions. Again consider NFA N in Figure 1. $\delta^*_N(q_\epsilon, 0) = \{q_\epsilon, q_0, q_{00}\}$ while $\delta(q_\epsilon, 0) = \{q_\epsilon, q_0\}$. Similarly, $\delta^*_N(q_0, 1) = \{q_p\}$ whereas $\delta(q_0, 1) = \emptyset$.

Despite these differences, the following result about computations on the concatenation of strings u, v holds.

Proposition 1. *Let M be an NFA. For every $u, v \in \Sigma^*$, and $p \in Q$,*

$$\delta^*_M(p, uv) = \bigcup_{q \in \delta^*_M(p,u)} \delta^*_M(q, v)$$

Recall that an NFA accepts input w, if there is some active thread in an accepting state after w is read from the initial state s. Using the fact $\delta^*_M(p, w)$ is the states of all active threads of M after input w when started from p, we can define acceptance as follows.

Definition 4. *For an NFA $M = (Q, \Sigma, \delta, s, A)$ and string $w \in \Sigma^*$, we say M* accepts *w iff $\delta^*_M(s, w) \cap A \neq \emptyset$.*
The language accepted or recognized *by NFA M over alphabet Σ is $\mathbf{L}(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$. A language L is said to be* accepted/recognized *by M if $L = \mathbf{L}(M)$.*

2 Examples

Recall that one view of an NFAs computation is that it can magically *guess* an accepting computation on an input (if one exists). This power of "guessing" can help in designing remarkably simple solutions to solving certain decision problems, because the NFA can guess what may happen in the future, so to speak. We illustrate this through a some examples.

Example 1. For $\Sigma = \{0, 1, 2\}$, let

$$L = \{w\#c \mid w \in \Sigma^*,\ c \in \Sigma, \text{ and } c \text{ occurs in } w\}$$

So $1011\#0 \in L$ but $1011\#2 \notin L$. The input alphabet for language L is $\Sigma \cup \{\#\}$.

An NFA recognizing L is shown in Figure 3. It works as follows.

- Read symbols of w, i.e., portion of input before # is seen
- Guess at some point that current symbol in w is going to be the same as 'c'; store this symbol in the state
- Read the rest of w
- On reading #, check that the symbol immediately after is the one stored, and that the input ends immediately after that.

Example 2. For alphabet Σ and $u \in \Sigma^*$, let

$$L_u = \{w \in \Sigma^* \mid u \text{ is a substring of } w\}$$

An NFA recognizing L_u could work as follows.

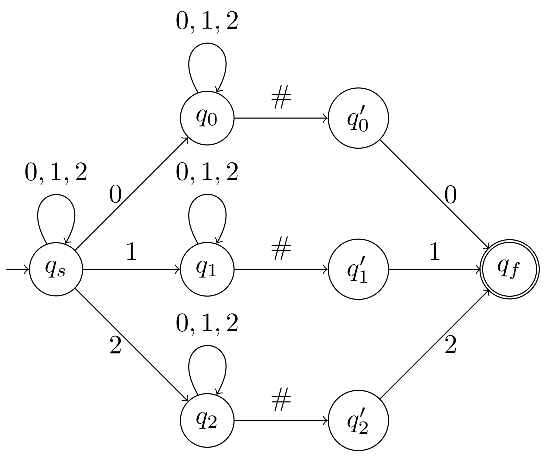

Figure 3: NFA recognizing $\{w\#c \mid c \text{ occurs in } w\}$.

- Read symbols of w
- Guess at some point that the string u is going to be seen
- Check that u is indeed read
- After reading u, read the rest of w

To do this, the automaton will remember in its state what prefix of u it has seen so far; the initial state will assume that it has not seen any of u, and the final state is one where all the symbols of u have been observed.

Formally, we can define this automaton as follows. Let $u = a_1a_2\cdots a_n$. The NFA $M_u = (Q, \Sigma, \delta, s, A)$ where

- $Q = \{\epsilon, a_1, a_1a_2, a_1a_2a_3, \ldots, a_1a_2\cdots a_n = u\}$. Thus, every prefix of u is a state of NFA M_u.
- $s = \epsilon$,
- $A = \{u\}$,
- And δ is given as follows

$$\delta(q,a) = \begin{cases} \{\epsilon\} & \text{if } q = \epsilon,\ a \neq a_1 \\ \{\epsilon, a_1\} & \text{if } q = \epsilon,\ a = a_1 \\ \{a_1a_2\cdots a_{i+1}\} & \text{if } q = a_1\cdots a_i\ (1 \leq i < n),\ a = a_{i+1} \\ \{u\} & \text{if } q = u \\ \emptyset & \text{otherwise} \end{cases}$$

Example 3. Recall that for a string $w \in \{0,1\}^*$, lastk(w) denotes the last k symbols in w. That is,

$$\text{lastk}(w) = \begin{cases} w & \text{if } |w| < k \\ v & \text{if } w = uv \text{ where } u \in \Sigma^* \text{ and } v \in \Sigma^k \end{cases}$$

Recall the language L_k is the set of all binary strings that have a 1 k-positions from the end. That is,

$$L_k = \{w \in \{0,1\}^* \mid \text{lastk}(w) = 1u \text{ where } u \in \{0,1\}^{k-1}\}$$

An NFA for L_k will work as follows.

- Read symbols of w
- Guess at some point that there are only going to be k more symbols in the input
- Check that the first symbol after this point is a 1, and that we see $k-1$ symbols after that
- Halt and accept no more input symbols

The states need to remember that how far we are from the end of the input; either very far (initial state), or less that k symbols from end.

Formally, $N_k = (Q_k, \{0,1\}, \delta_k, s_k, A_k)$ where

- $Q_k = \{q_i \mid 0 \leq i \leq k\}$. The subscript of the state counts how far we are from the end of the input; q_0 means that there can be many symbols left before the end, and q_i $(i > 1)$ means there are $k-i$ symbols left to read.
- $s_k = q_0$
- $A = \{q_k\}$,
- And δ is given as follows

$$\delta(q,a) = \begin{cases} \{q_0\} & \text{if } q = q_0,\ a = 0 \\ \{q_0, q_1\} & \text{if } q = q_0,\ a = 1 \\ \{q_{i+1}\} & \text{if } q = q_i (1 \leq i < k) \\ \emptyset & \text{otherwise} \end{cases}$$

Observe that this automaton has only $k+1$ states, whereas we proved in lecture 3 that any DFA recognizing this language must have size at least 2^k. Thus, NFAs can be exponentially smaller than DFAs.

Proposition 2. *There is a family of languages L_k (for $k \in \mathbb{N}$) such that the smallest DFA recognizing L_k has at least 2^k states, whereas there is an NFA with only $O(k)$ recognizing L_k.*

Proof. Follows from the observations above. □

3 Equivalence of NFAs and DFAs

DFAs and NFAs are computational models that solve decision problems. A natural question to ask is if there are problems that can solved on one model but not the other. In other words, we would like to ask

1. Is there a language that is recognized by a DFA but not by any NFAs?
2. Is there a language that is recognized by an NFA but not by any DFAs?

The answer to the first question is clearly no. Every DFA is a special kind of NFA. Thus if a language L is recognized by a DFA D, then since D is a special NFA, L is also recognized by an NFA. Surprisingly, it turns out the that answer to the second question is also no. Eventhough, NFAs have the ability to take steps without reading symbols, and magically guess the future, they cannot solve any problem that cannot be solved on a DFA. Hence, the main result we will prove in this section is the following.

Theorem 3. *For every NFA N, there is a DFA $\det(N)$ such that $\mathbf{L}(N) = \mathbf{L}(\det(N))$.*

In order to construct a DFA D that is equivalent to an NFA N, the DFA will "simulate" the NFA N on the given input. The computation of N on input w is completely determined by the threads that are active at each step. While the number of active threads can grow exponentially as the N reads more of the input, since the behavior of two active threads in the same state will be the same in the future, the DFA does not need to keep track of how many active threads are in a particular state, only whether there is an active thread in a particular state. Thus, to simulate the NFA, the DFA only needs to maintain the current set of states of the NFA.

The formal construction based on the above idea is as follows. Consider an NFA $N = (Q, \Sigma, \delta, s, A)$. Define the DFA $\det(N) = (Q', \Sigma, \delta', s', A')$ as follows.

- $Q' = \mathcal{P}(Q)$
- $s' = \delta_N^*(s, \epsilon)$
- $A' = \{X \subseteq Q \mid X \cap A \neq \emptyset\}$
- $\delta'(\{q_1, q_2, \ldots q_k\}, a) = \delta_N^*(q_1, a) \cup \delta_N^*(q_2, a) \cup \cdots \cup \delta_N^*(q_k, a)$ or more concisely,

$$\delta'(X, a) = \bigcup_{q \in X} \delta_N^*(q, a)$$

An example NFA is shown in Figure 4 along with the DFA $\det(N)$ in Figure 5.

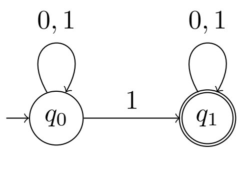

Figure 4: NFA N

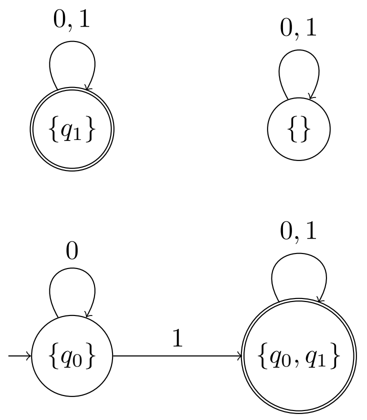

Figure 5: DFA $\det(N)$ equivalent to N

We will now prove that the DFA defined above is correct. That is

Lemma 4. $\mathbf{L}(N) = \mathbf{L}(\det(N))$

Proof. Need to show

$$\begin{array}{l} \forall w \in \Sigma^*.\ \det(N) \text{ accepts } w \text{ iff } N \text{ accepts } w \\ \forall w \in \Sigma^*.\ \delta_{\det(N)}^*(s', w) \in A' \text{ iff } \delta_N^*(s, w) \cap A \neq \emptyset \\ \forall w \in \Sigma^*.\ \delta_{\det(N)}^*(s', w) \cap A \neq \emptyset \text{ iff } \delta_N^*(s, w) \cap A \neq \emptyset \end{array}$$

Again for the induction proof to go through we need to strengthen the claim as follows.

$$\forall w \in \Sigma^*.\ \delta_{\det(N)}^*(s', w) = \delta_N^*(s, w)$$

In other words, this says that the state of the DFA after reading some string is exactly the set of states the NFA could be in after reading the same string.

The proof of the strengthened statement is by induction on $|w|$.

Base Case If $|w| = 0$ then $w = \epsilon$. Now

$$\delta_{\det(N)}^*(s', \epsilon) = s' = \delta_N^*(s, \epsilon) \text{ by the defn. of } \delta_{\det(N)}^* \text{ and defn. of } s'$$

Induction Hypothesis Assume inductively that the statement holds $\forall w.\ |w| < n$.

Induction Step Let w be such that $|w| = n$ (for $n > 0$). Without loss of generality w is of the form ua with $|u| = n-1$ and $a \in \Sigma$.

$$\begin{aligned}
\delta^*_{\det(N)}(s', ua) &= \delta^*_{\det(N)}(\delta^*_{\det(N)}(s,u), a) && \text{property of } \delta^* \text{ of DFAs}\\
&= \delta'(\delta^*_{\det(N)}(s,u), a) && \text{property of } \delta^* \text{ of DFAs}\\
&= \textstyle\bigcup_{q \in \delta^*_{\det(N)}(s,u)} \delta^*_N(q,a) && \text{definition of } \delta'\\
&= \textstyle\bigcup_{q \in \delta^*_N(s,u)} \delta^*_N(q,a) && \text{induction hypothesis on } u\\
&= \delta^*_N(s, ua) && \text{Proposition 1}
\end{aligned}$$

□

3.1 Relevant States

The formal definition of the DFA has many states, several of which are unreachable from the initial state (see Figure 5). To make the algorithm simpler for a human to implement (and for the resulting DFA to be readable), one can include only the reachable states. To do this,

1. Start by drawing the initial state of the DFA.
2. While there are states with missing transitions, draw the missing transitions creating any new states that maybe needed.
3. Step 2 is repeated until transitions from every state has been drawn.
4. Figure out which states are final, and mark them appropriately.

The method of adding only relevant states is shown in Figures 6 and 7.

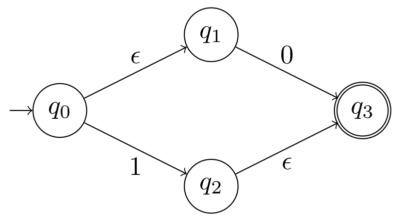

Figure 6: NFA N_ϵ

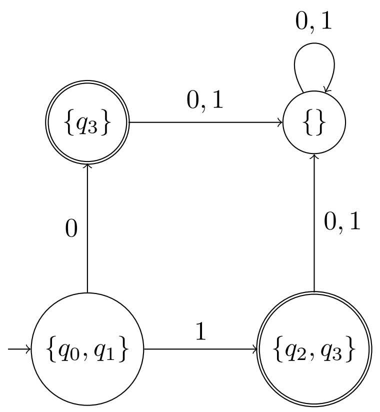

Figure 7: DFA $\det(N_\epsilon)$ with only relevant states