Chapter 2

Non-deterministic automata

The automata we have studied so far are called Deterministic Finite-state Automata (DFA) because when the DFA receive an input it is able to precisely determine its next state. *Deterministic behaviour is a highly desirable characteristic*, because it guarantees that we can implement a DFA as a program on a computer.

A Non-deterministic Finite state Automaton (NFA), has the property that it can be in more than one state at any one time. This (apparently weird) property allows us to express many automata more compactly than as a DFA.

Non-determinism appears to be an undesirable property (if the NFA cannot determine which state to enter next, what is it to do?) but it is not. It turns out that every NFA can be converted to an equivalent (though sometimes much larger) DFA, so NFAs can *always* be implemented, though with a bit more effort.

The attraction of the NFA is that we can more-easily *specify* the desired behaviour. Then, using a mechanical procedure, we can convert the NFA to a DFA, and thus implement it. This two-step approach neatly simplifies the overall task.

2.1 An informal view

In most respects, an NFA is the same as a DFA: It has a finite number of states; it accepts a finite set of input symbols; it begins in an initial state; and it has a set of accepting states. There is also a transition function, δ. Recall that in a DFA, $\delta(q, s)$ takes the current state, q, the current input symbol, s, and returns the next *state* to enter.

In an NFA, $\delta(q, s)$ takes a current state, q, the current input symbol s, and returns a *state-set* (a set of states) that the NFA enters. The set can contain zero, one, or more states.

Consider the $zeroOne$ recogniser described earlier. We can express this as an NFA as shown in fig. 2.1.

There are several things to notice about this diagram:

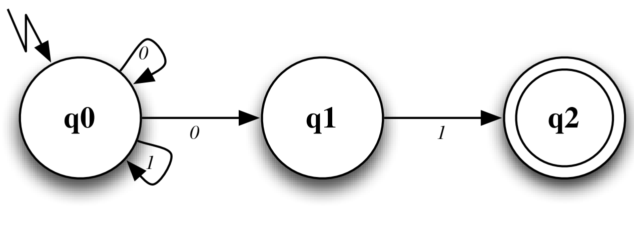


Figure 2.1: The zeroOne recogniser as an NFA

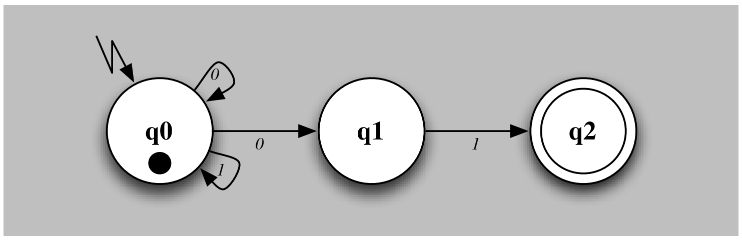
- The diagram has fewer transitions than the earlier DFA, so it appears simpler.
- There are *two* transitions from state q_0, labelled 0. Thus when a 0 is received, the NFA enters *both* state q_0 *and* state q_1. We will see how to think about this situation shortly.
- There is no transition corresponding to 0 from state q_1, and no transitions at all from state q_2. If these situations occur, the thread of the NFA's existence that corresponds to these states simply "dies". Other threads may continue to exist.

The central idea of this NFA is to try to "guess" when the final 01 has begun. Whenever it is in state q_0, and it sees a zero, it guesses that the 0 is the beginning of the final 01, so it enters state q_1. (If it guessed correctly, a subsequent 1 will cause it to enter state q_2.) However, just in case it makes a bad guess, and the 0 is *not* the beginning of a 01 sequence, the NFA "hedges its bets" and also remains in state q_0.

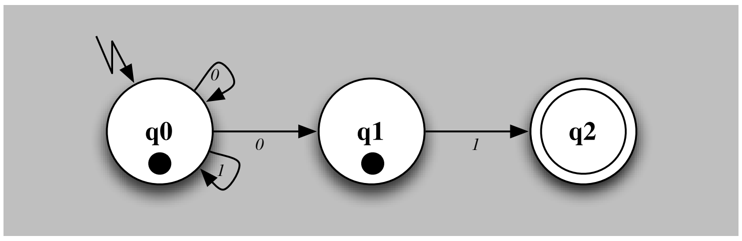
2.1.1 Example — the zeroOne NFA processing a string

To see how this works, let us process the same string 011001 that was handled by the previous DFA.

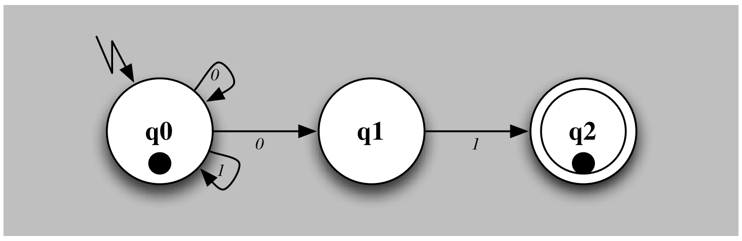
Initially, the NFA is in state q_0.



After processing the input symbol 0, the NFA enters states q_0 and q_1. There are thus *two* dots showing the current state.

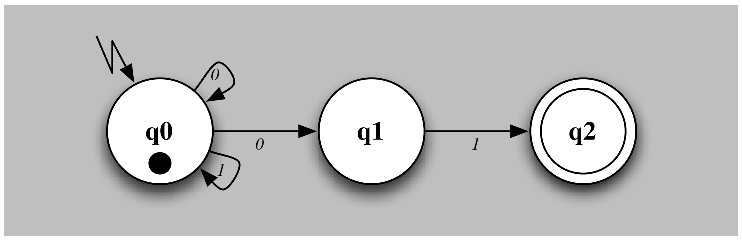


To process the input symbol 1, we must handle two cases. The dot in q_0 results in the next state being q_0. The dot in q_1 results in the next state being q_2. There are two current states q_0 and q_2, and the diagram now looks like this:

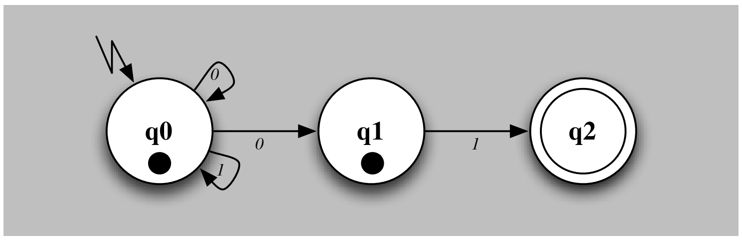


Since q_2 is an accepting state, the NFA has recognised a string that ends in 01.

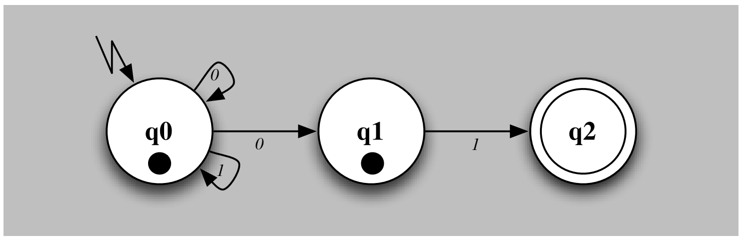
To process the input symbol 1, again we must handle two cases. The dot in q_0 results in the next state being q_0. q_2 has no outgoing transitions, so there is no next state. There is only one current state, and the diagram is:



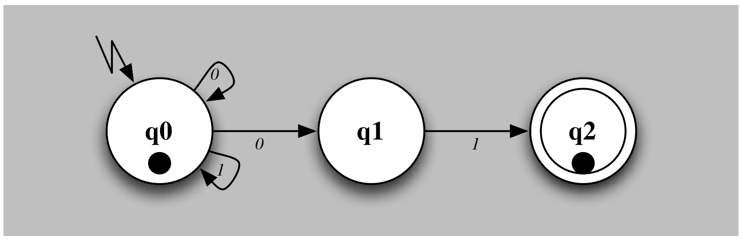
When we process the input symbol 0, the NFA makes two transitions to q_0 and q_1. There are again two current states, and the diagram is:



When we process the input symbol 0, there are two cases to handle: There is no outgoing transition from state q_1, so it is unable to handle the 0. State q_0 has two outgoing transitions labelled with 0, so the NFA enters states q_0 and q_1. The overall effect is for the NFA diagram to not change:



To process the input symbol 1, we again must handle two cases. The dot in q_0 results in the next state being q_0. The dot in q_1 results in the next state being q_2. There are two current states q_0 and q_2, and the diagram now looks like this:



Since q_2 is an accepting state, the NFA has recognised a string that ends in 01.

2.2 Formal definitions of an NFA

2.2.1 Five-tuple definition of an NFA

A *non-deterministic finite automaton* (NFA) consist of:

1. A finite set of *states*, usually denoted by Q
2. A finite set of *input symbols*, usually denoted by Σ.
3. A *transition function* $Q \times \Sigma \to \{Q\}$, that takes a state and an input symbol, and returns a *set* of new states. The transition function is usually denoted by δ.
4. A set of *start-states* $Q_0 \subseteq Q$. Q_0 is a set containing one or more of the the states in Q.
5. A set of *final* or *accepting* states F ⊆Q, usually denoted by F. Clearly, F is a subset of Q.

Notice that the there are two minor differences between a DFA and an NFA. The first difference is that the transition function for a DFA returns a *single* state, whereas for an NFA it returns a *set* of states.

The second difference follows from the first, and is that a DFA has a *single* start state, whereas an NFA has a *set* of start states.

A non-determinsitic finite automaton named N can be represented by a 5-tuple:

$$N = (Q, \Sigma, \delta, Q_0, F)$$

where Q is the set of states, Σ is the set of input symbols, δ is the transition function, Q_0 is the set of initial states, and F the set of final states.

2.2.2 Transition table definition of an NFA

Not surprisingly, we can represent the transition function for our example in tabular form, like this:

zeroOne

δ	**0**	**1**
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

The only difference between this table and the one for a DFA is that the *next-state* entries in the table are a *set* of states, instead of being a single state. For example, the entry for state q_0 given the symbol 0 is the set $\{q_0, q_1\}$. When there is no next state, we show the next state as an empty set, $\emptyset$.

2.3 How an NFA processes a string

2.3.1 The extended transition function, $Delta$

An NFA can simultaneously be in multiple states but, by convention, the transition function, δ, is specified in terms of a *single* state and *single* symbol, and returns a *set* of states. We will find it convenient to define a new function Δ that accepts as parameters a *set* of states, and a symbol, and returns a *set* of states.

If an NFA is in the set of states P, and receives an input symbol s, then the next state is determined by evaluating $delta$ for each state in that set (which will return a *set* of states), and then computing the union of those sets.

Formally:

$$Delta(P, s) = \bigcup_{p \in P} \delta(p, s)$$

2.3.2 Extending the transition function to strings

To understand how an NFA decides whether to "accept" a string of symbols, we need to see how it processes a string. The set of all strings that an NFA accepts is called its "language".

Suppose $s_1 s_2 s_3 \cdots s_n$ is a sequence of input symbols drawn from Σ, (the set of input symbols that this NFA can process).

We start with the NFA in its initial state-set, $R_0 = Q_0$, and use the extended transition function Δ to process the first input symbol to get the next the next state-set:

$$R_1 = \Delta(R_0, s_1)$$

R_1 is the set of new states that the NFA enters after receiving the input symbol s_1. We now take this state-set, and process the next symbol s_2, to find the next state-set, R_2:

$$R_2 = \Delta(R_1, s_2)$$

Continuing in this manner until all input symbols have been processed, we succesively enter sets of states $R_3, R_4, \ldots, R_n$. At each step,

$$R_i = \Delta(R_{i-1}, s_i)$$

If one of the states in R_n is an accepting state (i.e. if $R_n \cap F \neq \emptyset$), then the string of input symbols $s_1 s_2 s_3 \cdots s_n$ is "accepted". If not, it is "rejected".

2.4 The string transition function for an NFA

The extended transition function, Δ allows us to specify how an NFA in state-set Q, will move to state-set R upon receipt of a symbol s. We show this by writing $R = \Delta(Q, s)$.

We are interested in knowing what state-set an NFA will reach, starting at Q_0, if it is presented with a *string* of symbols $w = s_1 s_2 s_3 \cdots s_n$.

We can specify $\hat{\Delta}$, the string transition function, that takes a current *state-set* P, and a string of input symbols $w = s_1 s_2 s_3 \cdots s_n$, and generates the state-set R that the NFA will reach after processing those symbols: $R = \hat{\Delta}(P, w)$, like this:

$$\hat{\Delta} : \{Q\} \times \Sigma^* \to \{Q\}$$

(Note that $\{Q\}$ is a *set of sets of states* — the set of all subsets that can be created by selecting states from Q.)

We can define $\hat{\Delta}$ recursively, like this:

Base case If there is no input, the NFA stays in the current state-set, P. Thus the rule is:

$$\hat{\Delta}(P, \epsilon) = P$$

Recurrence case If we are given a string of symbols $s_1 s_2 s_3 \cdots s_n$, we “chop” the first symbol s_1 from the string, and use Δ to compute the state-set, T, that would be reached, starting at each state p in P:

$$T = \Delta(P, s_1)$$

We then use the $\hat{\Delta}$ function to process the remainder of the string, starting at T: $\hat{\Delta}(T, s_2 s_3 \cdots s_n)$. Combining the pieces, we get:

$$\hat{\Delta}(P, s_1 s_2 s_3 \cdots s_n) = \hat{\Delta}(\Delta(P, s_1)), s_2 s_3 \cdots s_n)$$

We can see that a recursive evaluation according to these rules *must* terminate in a finite number of steps, because each recursive invocation of $\hat{\delta}$ occurs with a shorter string of symbols. Eventually, we will reach the base case: $\hat{\Delta}(R, \epsilon)$.

2.5 The language of an NFA

We can define the language of an NFA in a very similar way to a DFA. The language is the set of all input strings that take the NFA from its initial state to an accepting state. Thus, for the NFA N, defined by:

$$N = (Q, \Sigma, \delta, Q_0, F)$$

The language $L(N)$ of this NFA is defined by:

$$L(N) = \{w \in \Sigma^* \mid \hat{\Delta}(Q_0, w) \cap F \neq \emptyset\}$$

Note that we must define the acceptance test slightly differently from a DFA. In a DFA, the $\hat{\delta}$ function returns a single state, and we check to see if that state is in F. For an NFA, $\hat{\Delta}$ returns a *set* of states. If one or more of those states is in F, then the NFA is in an accepting state. We express this condition by computing the intersection between the result of $\hat{\Delta}$ and F, $\hat{\Delta}(\cdots) \cap F$. If the intersection is *non-null*, we know that the NFA has reached an accepting state.

2.6 Equivalence of NFAs and DFAs

An NFA is often easier to construct than a DFA for a given language, which is sufficient justification for studying NFAs. You may be surprised to know that *every* NFA can be converted into an equivalent DFA, so it is always possible to implement an NFA.

In the worst case, an NFA with n states will require a DFA with $2^n - 1$ states. Fortunately, this case arises only rarely in practice, and a typical DFA has about the same number of states as the original NFA.

We can construct a DFA from an NFA by first creating all the non-empty *subsets* of the states of the NFA (there will be $2^n - 1$ such subsets, for an NFA with n states). Let us call each subset a *d-state*. We then build a DFA with a separate state for each d-state.

The goal is to take an NFA $N = (Q_N, \Sigma, \delta_N, Q_{0N}, F_N)$, and generate a corresponding DFA $D = (Q_D, \Sigma, \delta_D, q_{0D}, F_D)$ such that $L(N) = L(D)$.

We notice that the two automata have the same input alphabet, Σ, and the initial state of D (q_{0D}) is the d-state containing all the initial states of N (Q_{0N}).

The remaining parts of the automaton D can be constructed like this:

- Q_D is the set of all non-empty subsets of Q_N. We refer to each of these as a *d-state*. Since Q_N has n states, there will be 2^n subsets. After eliminating the empty subset, Q_D will have $2^n - 1$ d-states. Fortunately, in practical cases, many of these d-states will be unreachable from the initial d-state, and so can be thrown away. This can substantially reduce the complexity of the final DFA.

- The final-set of D ($= F_D$), is the set of d-states in Q_D, such that each d-state contains at least one accepting state of N. We can specify this as:

$$F_D = \{d \in Q_D \mid d \cap F_N \neq \emptyset\}$$

- For each d-state d in Q_D, and for each symbol t in Σ, we can define $\delta_D(d,t)$ by looking at each of the corresponding NFA states $s \in d$, and then seeing how N would handle each of those states for an input symbol t. Then we compute the union of those states, to get the actual d-state. Formally:

$$\forall d \in Q_D, \delta_D(d,t) = \Delta_N(d,t)$$

2.6.1 Converting our example NFA to a DFA

Taking our example of the $zeroOne$ NFA, we have: $Q_N = \{q_0, q_1, q_2\}$. Now Q_D is the set of all non-empty subsets of Q_N, thus the d-states in Q_D will be: $\{q_0\}$, $\{q_1\}$, $\{q_2\}$, $\{q_0, q_1\}$, $\{q_0, q_2\}$, $\{q_1, q_2\}$, and $\{q_0, q_1, q_2\}$. We can write:

$$Q_D = \{\{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$$

(Since N has 3 states, there are $2^3 - 1 = 7$ non-empty subsets, as we have just seen.)

The final states of the DFA are $\{q_2\}$, $\{q_0, q_2\}$, $\{q_1, q_2\}$, and $\{q_0, q_1, q_2\}$, since these d-states all include the final state q_2 of the original NFA.

When we construct the transition function, we find:

$zeroOne$

δ_D	**0**	**1**
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

To show how this table was derived, let us generate the entries for the d-state $\{q_0, q_2\}$. Consider first the case when the next input symbol is a 0. Within the d-state $\{q_0, q_2\}$ there are two states to consider, when we evaluate the Δ_N transition function:

$$\Delta_N(\{q_0, q_2\}, 0) = \delta_N(q_0, 0) \cup \delta_N(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$$

We thus see that $\delta_D(\{q_0, q_2\}, 0) = \{q_0, q_1\}$, so $\{q_0, q_1\}$ goes into the 0 column.

Now consider the case when the input symbol is a 1. Once again, within the d-state there are two states:

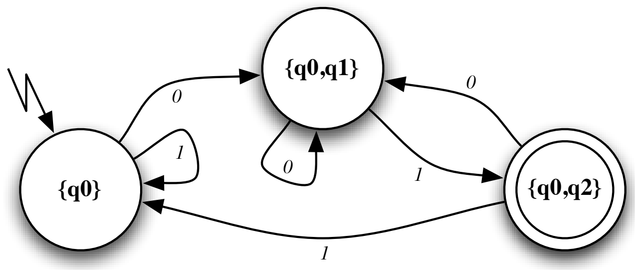
$$\Delta_N(\{q_0, q_2\}, 1) = \delta_N(q_0, 1) \cup \delta_N(q_2, 1) = \{q_0\} \cup \emptyset = \{q_0\}$$

We thus see that $\delta_D(\{q_0, q_2\}, 1) = \{q_0\}$. Which now goes into the 1 column.

The remaining entries in the table are computed in a similar way.

If we examine the table carefully, starting at the initial d-state $\{q_0\}$, we find we can reach only the d-states $\{q_0, q_1\}$, $\{q_0\}$, and $\{q_0, q_2\}$. All the other d-states are unreachable, and can safely be ignored. Notice that the DFA now has only three states, exactly the same as the original NFA! This is *much* better than the (potential) worst case of 7 states.

If we draw the DFA transition diagram that corresponds to this table, we get this diagram:



With the exception of the labels on the states, which are unimportant, because they are just names, this diagram is *identical* to the DFA we derived in chapter 1. This is, of course, exactly what we expected!