5.2. Nondeterministic FSMs (NDFSM)

A Nondeterministic FSM (or NDFSM) M is a quintuple (K, Σ, Δ, s, A). where:

K is a finite set of states

Σ is an alphabet

$s \in K$ is the initial state

$A \subseteq K$ is the set of accepting states, and

Δ is the transition relation is a finite subset of (K×(Σ U {ε}) x K)

In other words, each element of A contains a (state, input symbol or ε) pair and a new state. We define configuration, initial configuration, accepting configuration, yield-in-one-step and computation analogously to the way that we de tined them fur DFSMs.

Let *w* be an element of Σ*, then we will say that:

- M accepts w iff at least one of its computations accepts.
- M rejects w iff none or its computations accepts.

The language accepted by M, denoted L(M), is the set of all strings accepted by M.

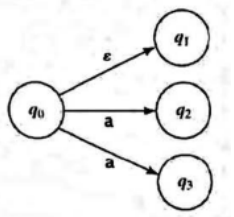
Difference between DFSM and FSM

There are two key differences between DFSMs and NDFSMs. ln every configuration, a DFSM can make exactly one move. However, because Δ can be an arbitrary relation (that may not be a function) that is not necessarily true for an NDFSM. Instead:

- An NDFSM M may enter a configuration in which there are still input symbols left to read but from which ***no moves are available***. This situation is possible because Δ is not a function. So there can be (state, input) pairs for which no next state is defined.
- An NDFSM M may enter a configuration from which **two or more competing moves** are possible. The competition can come from either or both of the following properties of the transition relation of an NDFSM:
  - An NDFSM M may have ***one or more transitions*** that are labeled ε, rather than being labeled with a character from Σ. An ε -transition out of state q may (but need not) be followed, without consuming any input, whenever M is in state q. So, an ε-transition from a state q competes with all other transitions out of q.
  - Out of some state q, there may be more than one transition with a given label. These competing transitions give M another way to guess at a correct path.

Consider the fragment, shown in figure, of an NDFSM M. lf M is in state q and the next input character is an a, then there are three moves that M could make:

1. It can take thee-transition to q before it reads the next input character.
2. It can read the next input character and take the transition to q_2.
3. It can read the next input character and take the transition to q_3



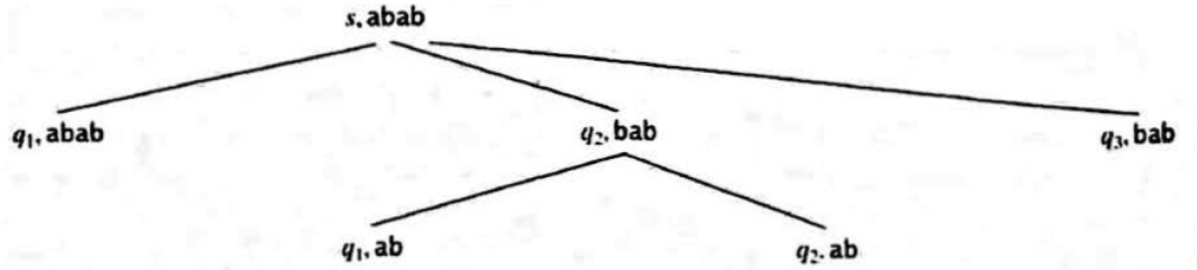


FIGURE 5.2 Viewing nondeterminism as search through a space of computation paths.

One way to envision the operation of M is as a tree, as shown in Figure 5.2. Each node in the tree corresponds to a configuration of M. Each path from the root corresponds to a sequence of moves that M might make. Each path that leads to a configuration in which the entire input string has been read corresponds to a computation of M.

An alternative is to imagine following all paths through M in parallel. Think of M as being in a set of states at each step of its computation. If, when M runs out of input, the set of states that it is in contains at least one accepting state, then M will accept.

Given an NDFSM M, such as any of the ones we have just considered. how can we analyze it to determine **what strings it accepts?** One way is to do a depth-first search of the paths through the machine. Another is to imagine tracing the execution of the original NDFSM M by following all paths in parallel.

Handling ε-Transitions

we introduce the function eps(q), where q is some state in M. **The function gives the set of states of M that are reachable from q by following zero or more ε-transitions.**

$$eps(q) = \{p \in K : (q, w) \vdash^*_M (p, w)\}.$$

eps(*q*) is the closure of {*q*} under the relation {(*p*, *r*) : there is a transition (*p*, ε, *r*) ∈ Δ}. The following algorithm computes *eps:*

eps(q: state) =

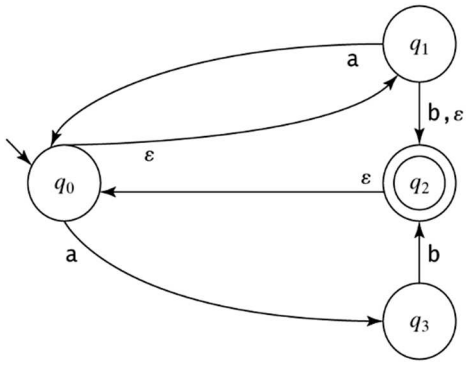
1. *result* = {*q*}.
2. While there exists some *p* ∈ *result* and
   some *r* ∉ *result* and
   some transition (*p*, ε, *r*) ∈ Δ do:
   Insert *r* into *result.*
3. Return *result*

Example:

Consider the following NDFSM M

The result of running *eps* on each of the states of *M* is:

$eps(q_0) = \{ q_0, q_1, q_2 \}$.
$eps(q_1) = \{ q_0, q_1, q_2 \}$.
$eps(q_2) = \{ q_0, q_1, q_2 \}$.
$eps(q_3) = \{ q_3 \}$.



A Simulation Algorithm

ndfsm-simulate (*M*: NDFSM, *w*: string) =

1) *current-state* = *eps*(*s*).
2) While any input symbols in *w* remain to be read do:
   a) *c* = get-next-symbol(*w*).
   b) *next-state* = ∅.
   c) For each state *q* in *current-state* do:
      For each state *p* such that (*q*, *c*, *p*) ∈ Δ do:
      next-state = *next-state* ∪ *eps*(*p*).
   d) *current-state* = *next-state*.
3) If *current-state* contains any states in *A*, accept. Else reject.

Theorem: For every DFSM there is an equivalent NDFSM.

Proof: Let M be a DFSM that accepts some language L. M is also an NDFSM that happens to contain no ε-transitions and whose transition relation happens to be a function. So the NDFSM that we claim must exist is simply M.

Theorem: For each NDFSM, there is an equivalent DFSM

Proof: By construction.

The Algorithm *ndfsm-to-dfsm*

ndfsmtodfsm(*M*: NDFSM) =

1. For each state *q* in K_M do:
   Compute *eps*(*q*).
2. *s′* = *eps*(*s*)
3. Compute δ′:
   3.1 *active-states* = {*s′*}.
   3.2 δ′ = ∅.
   3.3 While there exists some element *Q* of *active-states* for which δ′ has not yet been computed do:
      For each character *c* in Σ_M do:
         new-state = ∅.
         For each state *q* in *Q* do:
            For each state *p* such that (*q*, *c*, *p*) ∈ Δ do:
               new-state = *new-state* ∪ *eps*(*p*).
         Add the transition (*Q*, *c*, *new-state*) to δ′.
         If *new-state* ∉ *active-states* then insert it.
4. *K′* = *active-states*.
5. *A′* = {*Q* ∈ *K′* : *Q* ∩ *A* ≠ ∅ }.

Note: Refer class notes for problems on NDFSM to DFSM conversion

From FSMs to Operational Systems

An FSM is an abstraction. We can describe an FSM that solves a problem without worrying about many kinds of implementation details. FSMs for real problems can be turned into operational systems in any of a number of ways:

- An FSM can be translated into a circuit design and implemented directly in hardware.
- An FSM can be simulated by a general-purpose interpreter. Sometimes all that is required is a simulation. In other cases, a simulation can he used to check a design before it is translated into hardware.
- An FSM can be used as a specification for some critical aspect of the behavior of a complex system. The specification can then be implemented in software just as any specification might be. And the correctness of the implementation can be shown by verifying that the implementation satisfies the specification.

5.5. Simulators for FSMs

We begin by considering only deterministic FSMs. One approach is to think of an FSM as the specification for a simple, table-driven program and then proceed to write the code.

A Deterministic FSM Interpreter

dfsm-simulate(M: DFSM, w: string) =
1. st = s.
2. Repeat
   c = get-next-symbol(w).
   If c ≠ end-of-file then
   st = δ(st, c).
   until c = end-of-file.
3. If st ∈ A then accept else reject.

The algorithm ***dfsm-simulate*** runs in time approximately O(|w|), if we assume that the lookup table can be implemented in constant time.

Simulating Nondeterministic FSMs

Real computers are deterministic, so we have three choices if we want to execute a NDFSM:

1. Convert the NDFSM to a deterministic one:
   - Conversion can take time and space $2^{|K|}$.
   - Time to analyze string w: O(|w|)
2. Simulate the behavior of the nondeterministic one by constructing sets of states "on the fly" during execution
   - No conversion cost

- Time to analyze string w: $O(|w| \times |K|^2)$

3. Do a depth-first search of all paths through the nondeterministic machine.

In the 1st choice, converting an NDFSM to a DFSM can be very inefficient in terms of both time and space. space. If M has k states, it could lake time and space equal to $O(2^k)$ just to do the conversion, although the simulation after the conversion would take time equal to O(|w|). So we would like to follow 2nd choice, that directly simulates an NDFSM M without converting it to a DFSM first.

The idea is to simulate being in sets of states at once. But instead of generating all of the reachable sets of states right away, as *ndfsm-to-dfsm* does, it generates them on the fly as they are needed, being careful not to get stuck chasing ε-loops.

A NDFSM Interpreter/Simulator

ndfsm-simulate(M = (K, Σ, Δ, s, A): NDFSM, w: string) =

1. Declare the set st.
2. Declare the set st1.
3. st = eps(s).
4. Repeat
   4.1 c = get-next-symbol(w).
   4.2 If c ≠ end-of-file then do
   st1 = ∅.
   For all q ∈ st do
   For all r ∈ Δ(q, c) do
   st1 = st1 ∪ eps(r).
   st = st1.
   If st = ∅ then exit.
   until c = end-of-file.
5. If st ∩ A ≠ ∅ then accept else reject.

Now there is no conversion cost. To analyze a string w requires |w| passes through the main loop in step 4. ln the worst case, M is in all states all the time and each of them has a transition to every other one. So one pass could take as many as $O(|K|^2)$ steps, for a total cost of $O(w\,|K|^2)$

5.4. Minimizing FSMs

If we are going to solve a real problem with an FSM. we may want to find the smallest one that docs the job. We will say that a DFSM M is **minimal** iff there is no other DFSM M' such that L(M) = L(M') and M' has fewer states than M does.

We might want to be able to ask:

- Given a language L, is there a minimal DFSM that accepts L?
- If there is a minimal machine, is it unique?
- Given a DFSM M that accepts some language L, can we tell whether M is minimal?
- Given a DFSM M, can we construct a minimal equivalent DFSM A'?

The answer to all four questions is **yes**.

Building a Minimal DFSM for a Language

An effective way to think about the design of a DFSM M to accept some language L over an alphabet I is to cluster the strings in Σ*. in such a way that strings that share a future will drive M to the same state.

We will say that x and y are **indistinguishable** with respect to L, which we will write as $x \approx_L y$ iff: $\forall z \in \Sigma^*$ (either both xz and yz ∈ L or neither is).

EXAMPLE 5.22 How $\approx_L$ Depends on *L*

If $L = \{a\}^*$, then $a \approx_L aa \approx_L aaa$. But if $L = \{w \in \{a, b\}^* : |w| \text{ is even}\}$, then $a \approx_L aaa$, but it is not the case that $a \approx_L aa$ because, if $z = a$, we have $aa \in L$ but $aaa \notin L$.

We will say that x and y are ***distinguishable*** with respect to L, iff they are not indistinguishable. So, if x and y are distinguishable, then there exists at least one string z such that one but not both of xz and yz is in L.

Note that $\approx_L$ is an equivalence relation because it is:

- Reflexive: $\forall x \in \Sigma^* (x \approx_L x)$, because $\forall x, z \in \Sigma^* (xz \in L \leftrightarrow xz \in L)$.
- Symmetric: $\forall x, y \in \Sigma^* (x \approx_L y \rightarrow y \approx_L x)$, because $\forall x, y, z \in \Sigma^* ((xz \in L \leftrightarrow yz \in L) \leftrightarrow (yz \in L \leftrightarrow xz \in L))$.
- Transitive: $\forall x, y, z \in \Sigma^* (((x \approx_L y) \wedge (y \approx_L w)) \rightarrow (x \approx_L w))$, because: $\forall x, y, z \in \Sigma^* (((xz \in L \leftrightarrow yz \in L) \wedge (yz \in L \leftrightarrow wz \in L)) \rightarrow (xz \in L \leftrightarrow wz \in L))$.

Since $\approx_L$ is an equivalence relation, its equivalence classes constitute a partition of the set Σ^*. So:

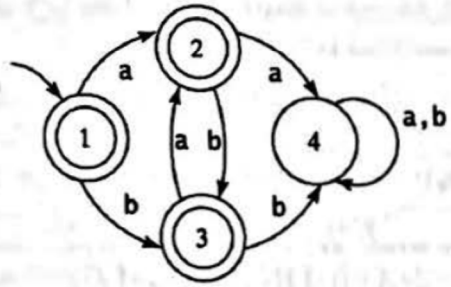
- No equivalence class of $\approx_L$ is empty, and
- Every string in Σ^* is in exactly one equivalence class of $\approx_L$.

Building a Minimal DFSM from ≈L: Let Σ = {a,b}. Let L = { w ∈ {a,b}* : no two adjacent characters are the same }. The equivalence classes of ≈L are:

[1]	[ε]	{ε}.
[2]	[a, aba, ababa, ...]	{all nonempty strings that end in a and have no identical adjacent characters}.
[3]	[b, ab, bab, abab, ...]	{all nonempty strings that end in b and have no identical adjacent characters}.
[4]	[aa, abaa, ababb ...]	{all strings that contain at least one pair of identical adjacent characters; these strings are not in L, no matter what comes next}.

We build a **minimal DFSM** M to accept L as follows:

- The equivalence classes of $\approx_L$ become the states of M.
- The start state is [ε] = [1].
- The accepting states are all equivalence classes that contain strings in L, namely [1], [2], and [3].
- $\delta([x], a) = [xa]$. So, for example, equivalence class [1] contains the string ε. If the character a follows ε, the resulting string, a, is in equivalence class [2]. So we create a transition from [1] to [2] labeled a. Equivalence class [2] contains the string a. If the character b follows a, the resulting string, ab, is in equivalence class [3]. So we create a transition from [2] to [3] labeled b. And so forth.



Minimizing an Existing DFSM

Now suppose that we already have a DFSM M that accepts L. In fact, possibly M's the only definition we have of L. In this case, it makes sense to construct a minimal DFSM to accept L by starting with M rather than with ≈L. There are **two approaches** that we could take to constructing a minimization algorithm:

1. Begin with M and collapse redundant slates, getting rid of one at a time until the resulting machine is minimal.
2. Begin by over-clustering the states of L into just two groups. accepting and nonaccepting, then iteratively split those groups apart until all the distinctions that L requires have been made.

We follow **2nd approach** in our example.

Our goal is to end up with a minimal machine in which all equivalent states of M have been collapsed. In order to do that, we need a precise definition of what it means for two states to be equivalent (and thus collapsible). We will use the following;

We will say that two states q and p in M are equivalent. which we will write q ≡ p iff for all strings w ∈ ∑*, either w drives M to an accepting state from both q and p or it drives M to a rejecting state from both q and p. In other words, no matter what continuation string comes next, M behaves identically from both states. Note that ≡ is an equivalence relation over states, so it will partition the states of M into a set of equivalence classes. The algorithm can be stated as follows.

minDFSM(*M*: DFSM) =

1. *classes* = {*A*, *K*–*A*}. /* Initially, just two classes of states, accepting and rejecting.

2. Repeat until a pass at which no change to *classes* has been made:

   2.1. *newclasses* = ∅. /* At each pass, we build a new set of classes, splitting the old ones as necessary. Then this new set becomes the old set, and the process is repeated.

   2.2. For each equivalence class *e* in *classes*, if *e* contains more than one state, see if it needs to be split:

      For each state *q* in *e* do: /* Look at each state and build a table of what it does. Then the tables for all states in the class can be compared to see if there are any differences that force splitting.

         For each character *c* in Σ do:

            Determine which element of *classes* *q* goes to if *c* is read.

         If there are any two states *p* and *q* such that there is any character *c* such that, when *c* is read, *p* goes to one element of *classes* and *q* goes to another, then *p* and *q* must be split. Create as many new equivalence classes as are necessary so that no state remains in the same class with a state whose behavior differs from its. Insert those classes into *newclasses*.

      If there are no states whose behavior differs, no splitting is necessary. Insert *e* into *newclasses*.

   2.3. *classes* = *newclasses*.

/* The states of the minimal machine will correspond exactly to the elements of *classes* at this point. We use the notation [*q*] for the element of *classes* that contains the original state *q*.

3. Return $M' = (classes, \Sigma, \delta, [s_M], \{[q: \text{the elements of } q \text{ are in } A_M]\})$, where $\delta_{M'}$ is constructed as follows:

   if $\delta_M(q, c) = p$, then $\delta_{M'}([q], c) = [p]$.

Clearly, no class that contains a single state can be split. So, if $|K|$ is k, then the maximum number of times that *minDFSM* can split classes is $k - 1$. Since *minDFSM* halts when no more splitting can occur, the maximum number of times it can go through the loop is $k - 1$. Thus *minDFSM* must halt in a finite number of steps. M' is the minimal DFSM that is equivalent to M since:

- M' is minimal: It splits classes and thus creates new states only when necessary to simulate M, and
- $L(M') = L(M)$: The proof of this is straightforward by induction on the length of the input string.

Note: Refer class notes for problems on Minimizing DFSM

Canonical form of Regular languages

A canonical form for some set of objects C assigns exactly one representation to each class of "equivalent" objects in C. Further each such representation is distinct, so two objects in C share the same representation iff they are "equivalent" in the sense for which we define the form.

Suppose that we had a canonical form for FSMs with the property that two FSMs share a canonical form iff they accept the same language. Further suppose that we had an algorithm that on input M, constructed M's canonical form. Then some questions about FSMs would become easy to answer. For example, we could test whether two FSMs are equivalent (i.e., they accept the same language), it would suffice to construct the canonical form for each of them and test whether the two forms are identical.

The algorithm minDFSM constructs, from any DFSM M, a minimal machine that accepts L(M). All minimal machines for L(M) are identical except possibly for state names. So if we could define a standard way to name states we could define a canonical machine to accept L(M) (and thus any regular language). The following algorithm does this by using the slate-naming convention.

buildFSMcanonicalform(*M*: FSM) =

1. $M' = ndfsmtodfsm(M)$.
2. $M\# = minDFSM(M')$.
3. Create a unique assignment of names to the states of $M\#$ as follows:
   - 3.1. Call the start state q_0.
   - 3.2. Define an order on the elements of Σ.
   - 3.3 Until all states have been named do:

     Select the lowest numbered named state that has not yet been selected. Call it q.

     Create an ordered list of the transitions out of q by the order imposed on their labels.

     Create an ordered list of the as yet unnamed states that those transitions enter by doing the following: If the first transition is (q, c_1, p_1), then put p_1 first. If the second transition is (q, c_2, p_2) and p_2 is not already on the list, put it next. If it is already on the list, skip it. Continue until all tran-

sitions have been considered. Remove from the list any states that have already been named.

Name the states on the list that was just created: Assign to the first one the name q_k, where k is the smallest index that hasn't yet been used. Assign the next name to the next state and so forth until all have been named.

4. Return *M#*.

Given two FSMs M1 and M2 *buildFSMcanonicalform(M1)* = *buildFSMcanonicalform(M2)* iff L(M1) = L(M2).