Chapter 3

INTRODUCTION TO AUTOMATA THEORY

In this chapter we study the most basic abstract model of computation. This model deals with machines that have a finite memory capacity. Section 3.1 deals with machines that operate deterministically for any given input while Section 3.2 deals with machines that are more flexible in the way they compute (i.e., nondeterministic choices are allowed).

3.1 Deterministic Finite-State Machines

A computer's ability to recognize specified patterns is desirable for many applications such as text editors, compilers, and databases. We will see that many of these patterns can be efficiently recognized with the simple finite-state machines discussed in this Chapter.

3.1.1 The cheap engineer's elevators

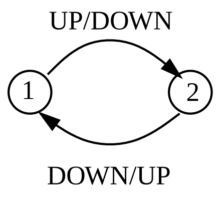
What kind of machine has only a finite amount of memory? Of course, you may first think that a desk top computer's memory (e.g., one with 64M RAM) is finite. But this is not exactly true since we have external memory (hard drives, tape drives, and floppy disks) that could extend the memory capacity to an arbitrary large limit.

We have a much simpler model in mind where a machine is viewed as a closed computation box with only a read-only tape head (or toggle switches) for external input. That is, the internal part of the machine can be in one of a finite number of *states*. A certain subset of these states, called *accepting states*, will indicate that the computation has been successful. If the machine halts in an accepting state, we accept (or recognize) the input as valid.

To illustrate this further we consider an over-simplified construction of an elevator control mechanism. Of course, in this example, we are not planning to recognize valid input, just to show how a real-world finite-state device operates.

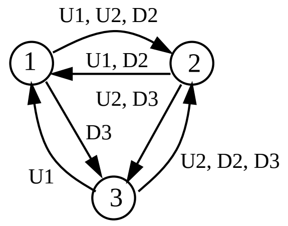
First consider an elevator that moves between two levels. We will build a device with two states $\{1, 2\}$, where the state number corresponds with what floor the elevator is currently located. To save cost we have two types of inputs UP and DOWN; a button on each floor indicating that a person on one of the floors wants to go to the other. The state changes, called *transitions*, of this elevator can be depicted in the following table format or graphical diagram format. The entries in the table denote a new state of machine after an input is received (the directional arcs on the digraph diagram denotes the same thing). There are four cases to consider. E.g., if the elevator is on floor 2 and the DOWN button is pressed then the elevator should move to floor 1.

	(input buttons)	
States	DOWN	UP
1	2	2
2	1	1



We can extend this low budget elevator example to handle more levels. With one additional floor we will have more combinations of buttons and states to deal with. At floor 2 we need both an up button, U2, and a down button, D2. For floor 1 we just need an up button, U1, and likewise for floor 3, we just need a down button, D3. This particular elevator with three states is represented as follows:

	(input buttons)			
States	U1	U2	D2	D3
1	2	2	2	3
2	1	3	1	3
3	1	2	2	2



One may see that the above elevator probably lacks in functionality since to travel two levels one has to press two buttons. Nevertheless, these two small examples should indicate what we mean by a finite-state machine.

3.1.2 Finite-state machines that accept/reject strings

We now consider finite-state machines where the input is from some finite character alphabet Σ. Our examples will mainly use simple character sets such as $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, c, d\}$ but in practice they may be as big as the set of 7-bit ASCII characters commonly used by computers. To do real computations we need a notion of an initial or starting state; we also need some means to determine whether the result of our computation is successful or not. To achieve this we need to designate an unique starting state and classify each state as an accepting or rejecting state.

A formal definition of our (first) finite-state computation model is given next.

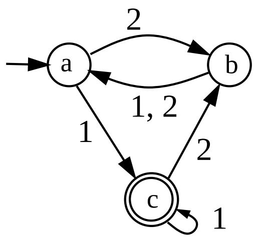
Definition 40. A *deterministic finite automaton (DFA)* is a five-tuple $M = (Q, \Sigma, \delta, s, F)$ where

1. Q is the finite set of machine states.
2. Σ is the finite input alphabet.
3. δ is a transition function from $Q \times \Sigma$ to Q.
4. $s \in Q$ is the start state.
5. $F \subseteq Q$ is the accepting (membership) states.

Notice that the set of rejecting states is determined by the set difference $Q \setminus F$. Other authors sometimes define the next state function δ as a partial function (i.e., not all states accept all inputs).

Example 41. A very simple DFA example is M1 $= (Q = \{a, b, c\}, \Sigma = \{1, 2\}, \delta, s = a, F = \{c\})$, where δ is represented in two different ways below.

δ	(input Σ)	
States	1	2
a	c	b
b	a	a
c	c	b

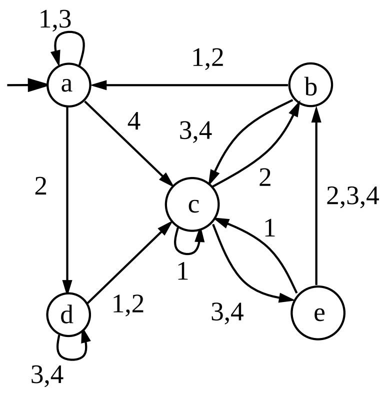


In the graphical representation we use double circles to denote the accepting states F of Q. Also the initial state $s \in Q$ has an isolated arrow pointing at it.

Example 42. A more complicated DFA example is M2 below with $Q = \{a, b, c, d, e\}$, $\Sigma = \{1, 2, 3, 4\}$, $s = a$, $F = \{c, e\}$ and δ is represented by the following transition table.

δ	(input Σ)			
States	1	2	3	4
a	a	d	a	c
b	a	a	c	c
c	c	b	e	e
d	c	c	d	d
e	c	b	b	b

It is easy to generate a directed graph (self-loops allowed) representation from above. We just view δ as an arc relationship on the states Q. Notice how we combine arcs (into one) with different labels between the same two states for ease of presentation, as done for this view of DFA M2.



3.1.3 Recognizing patterns with DFA

There are two main questions we have at this point concerning DFA and the process of pattern recognition of strings of characters.

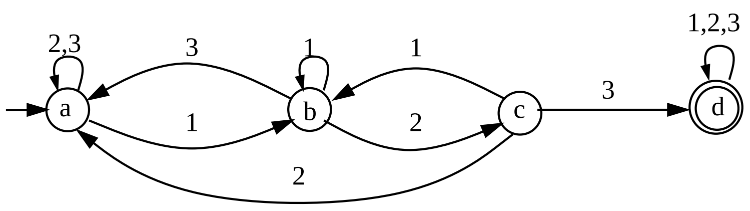
- For a given a DFA, what strings does it accept?
- For a given set of strings, can we build a DFA that recognizes just them?

Before proceeding we need a name for the set of inputs accepted by some automaton.

Definition 43. For a DFA M, the set of strings (*words*) accepted by M is called the *language* $L(M)$ decided (recognized) by M. The set $L(M)$ is simply as subset of Σ^*, all character sequences of the input alphabet Σ.

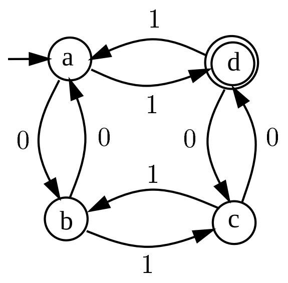
We will see later in Section 3.4 that the languages recognizable by finite automata are exactly those expressible as regular expressions.

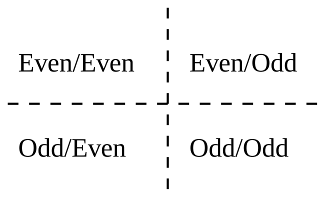
Example 44. For the DFA M listed below, $L(M)$ is the set of strings (over $\Sigma = \{1, 2, 3\}$) that contain the substring '123'.



To compute $L(M)$ we need to classify all possible strings that yield state transitions to an accept state from the initial state. When looking at a graphical representation of M, the strings are taken from the character sequence of traversed arcs. Note that more than one string may correspond to a given path because some arcs have more than one label.

Example 45. We construct a DFA that accepts all words with an even number of 0's and an odd number of 1's. A 0/1 parity guide for each of the four states of the DFA is given on the right.





We end this section by mentioning that there are some languages such as $L = \{0^n 1^n \mid n > 0\}$ that are not accepted by any finite-state automaton. Why is L not recognized by any DFA? Well, if we had a DFA M of m states then it would have problems accepting just the set L since the automaton has to keep a different state for each count of the 0's it reads before it reads the 1's. If two different counts i and j, $i < j$, of 0's share the same state then $0^j 1^i$ would be accepted by M, which is not in L.

3.2 Nondeterministic Finite-State Machines

Nondeterminism allows a machine to select one of several state transitions randomly. This includes a choice for initial state. This flexibility makes it easier (for a human designer) to build an automaton that recognizes strings in a particular language. Below we formally define this relaxed model of computation. We will see in the next section how to (algorithmically) produce an equivalent deterministic machine from a nondeterministic one.

Definition 46. A *nondeterministic finite automaton (NFA)* is a five-tuple $(Q, \Sigma, \delta, S, F)$ where

1. Q is the finite set of machine states.

2. Σ is the finite input alphabet.

3. δ is a function from $Q \times \Sigma$ to 2^Q, the set of subsets of Q.

4. $S \subseteq Q$ is a set of start (initial) states.

5. $F \subseteq Q$ is the accepting (membership) states.

Notice that the state transition function δ is more general for NFA's than DFA's. Besides having transitions to multiple states for a given input symbol, we can have $\delta(q, c)$ undefined for some $q \in Q$ and $c \in \Sigma$. This means that that we can design automata such that no state moves are possible for when in some state q and the next character read is c (i.e., the human designer does not have to worry about all cases).

An NFA accepts a string w if there exists a nondeterministic path following the legal moves of the transition function δ on input w to an accept state.

Other authors sometimes allow the next state function δ for NFA to include epsilon ϵ transitions. That is, a NFA's state may change to another state without needing to read the next character of the input string. These state jumps do not make NFA's any more powerful in recognizing languages because we can always add more transitions to bypass the epsilon moves (or add further start states if an epsilon leaves from a start state). For this introduction, we do not consider epsilon transitions further.

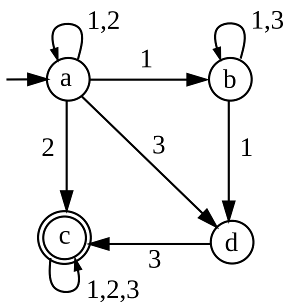
3.2.1 Using nondeterministic automata

We now present two examples of nondeterministic finite-state automata (NFA's).

Example 47. An NFA N with four states $Q = \{a, b, c, d\}$, input alphabet $\Sigma = \{1, 2, 3\}$, start states $S = \{a\}$, accepting states $F = \{c\}$ and transition function δ is given below:

δ	(input Σ)		
States	1	2	3
a	{a,b}	{a,c}	$\{d\}$
b	{b,d}	$\emptyset$	$\{b\}$
c	$\{c\}$	$\{c\}$	$\{c\}$
d	$\emptyset$	$\emptyset$	$\{c\}$

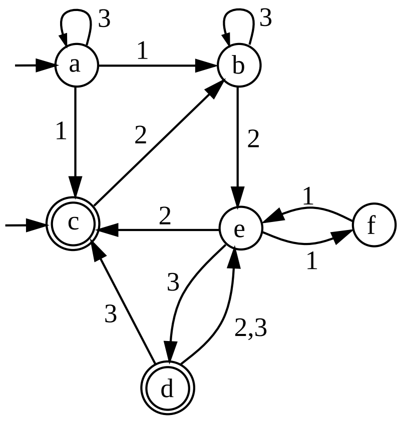
Note that there are no legal transitions from state b on input 2 (or from state d on inputs 1 or 2) in the above NFA. The corresponding graphical view is given below.



We can see that the language $L(N)$ accepted by this NFA N is the set of strings that start with any number of 1's and 2's, followed by a 2 or 33 or (1 and (1's and/or 3's) and 13). We will see how to describe languages such as $L(N)$ more easily when we cover regular expressions in Section 3.4 of these notes.

Example 48. An example NFA with multiple start states is N2 with six states $Q = \{a, b, c, d, e, f\}$, input alphabet $\Sigma = \{1, 2, 3\}$, start states $S = \{a, c\}$, accepting states $F = \{c, d\}$ and transition function δ is given below:

δ	(input Σ)		
States	1	2	3
a	{b,c}	$\emptyset$	$\{a\}$
b	$\emptyset$	$\{e\}$	$\{b\}$
c	$\emptyset$	$\{b\}$	$\emptyset$
d	$\emptyset$	$\{e\}$	$\{c, e\}$
e	$\{f\}$	$\{c\}$	$\{d\}$
f	$\{e\}$	$\emptyset$	$\emptyset$



The above example is somewhat complicated. What set of strings will this automata accept? Is there another NFA of smaller size (number of states) that recognizes the same language? We have to develop some tools to answer these questions more easily.

3.2.2 The reverse $R(L)$ of a language L

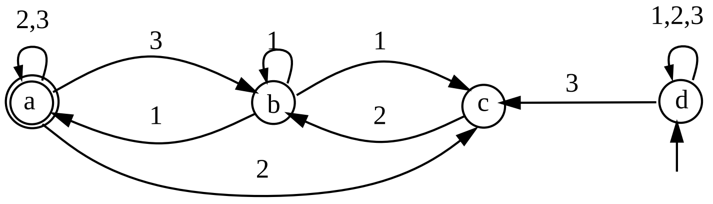
If we have an automaton (either DFA or NFA) M that recognizes a language L we can systematically construct an NFA M′ that recognizes the reverse language $R(L)$. The *reverse* of a string $w = c_1c_2c_3 \dots c_n$ is the string $w' = c_nc_{n-1} \dots c_2c_1$. If M accepts w then M′ accepts w'.

Definition 49. The *reverse or dual machine* M′ of an NFA M is constructed as follows:

1. The start states of M′ are the accept states of M.
2. The accept states of M′ are the initial states of M.
3. If $\delta(q_1, c) = q_2$ is in M then $\delta(q_2, c) = q_1$ is in M′. I.e., all transitions are reversed.

It is easy to see that the dual machine M′ of an automaton M recognizes the reverse strings that M accepts.

Example 50. The dual machine of Example 44 is given below.

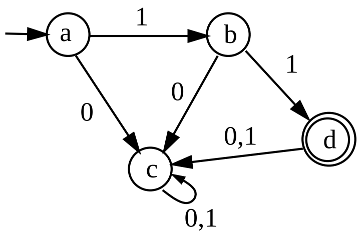


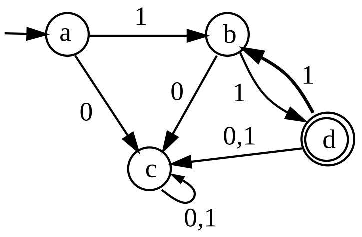
Notice that the dual machine may not be the simplest machine for recognizing the reverse language of a given automaton.

3.2.3 The closure $C(L)$ of a language L

We want to introduce another useful NFA associated with a given automaton. The *closure*, $C(L)$ of a language L is defined to be the set of strings that can be formed by concatenating together any number of strings of L. Given a DFA (or NFA) M that recognizes a language L we can build an NFA M′ that recognizes the closure of L by simply adding transitions from all accept state(s) to the neighbors of the initial state(s).

Example 51. The DFA displayed on the left below accepts only the word 11. The closure of this language, $C(L) = \{1^{2k} \mid k \geq 1\}$, is accepted by the NFA on the right.





In the above example only one transition arc $\delta(d, 1) = b$ was added since the transition $\delta(d, 0) = c$ already existed.

3.3 Recognition Capabilities of NFA's and DFA's

Although NFA's are easier than DFA's for the human to design, they are not as usable by a (deterministic) computer. This is because nondeterminism does not give precise steps for execution. This section shows how one can take advantage of both the convenience of NFA's and the practicality of DFA's. The following algorithm can be used to convert an NFA N to a DFA M that accepts the same set of strings.

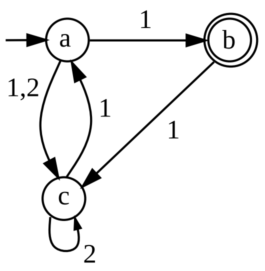
```
  algorithm NFAtoDFA(NFA N, DFA M)
1 The initial state $s_M$ of M is the set of all initial states of N, $S_N$.
  $Q_M = \{s_M\}$
```

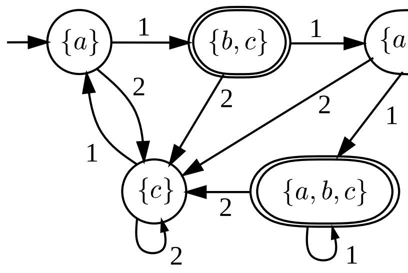
```
2   while $Q_M$ has not increased in size
        for each new state $q_M = \{a_1, a_2, \ldots, a_k\} \in Q_M$ do
            for each input $x \in \Sigma$ do
            $\delta_M(q_M, x)$ is the set $q'_M$ of all states of N reachable from $a_i$ on input $x$.
            (I.e., $q'_M = \{a_j \mid \delta_N(a_i, x) = a_j, 1 \leq i \leq k\}$)
            $Q_M = Q_M \cup \{q'_M\}$
            endfor
        endfor
    endwhile
3   The accepting states $F_M$ is the set of states that have an accepting state of $N$.
    (I.e., $F_M = \{q_M \mid a_i \in q_M$ and $a_i \in F_N\}$)
    end
```

The idea behind the above algorithm is to create potentially a state in M for every subset of states of N. Many of these states are not reachable so the algorithm often terminates with a smaller deterministic automaton than the worst case of $2^{|N|}$ states. The running time of this algorithm is $O(|Q_M| \cdot |\Sigma|)$ if the correct data structures are used.

The algorithm NFAtoDFA shows us that the recognition capabilities of NFA's and DFA's are equivalent. We already knew that any DFA can be viewed as a special case of an NFA; the above algorithm provides us with a method for mapping NFA's to DFA's.

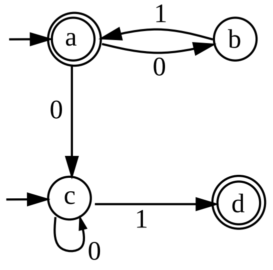
Example 52. For the simple NFA N given on the left below we construct the equivalent DFA M on the right, where $L(N) = L(M)$.

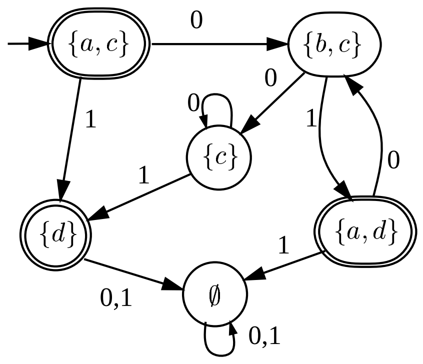




Notice how the resulting DFA from the previous example has only 5 states compared with the potential worst case of 8 states. This often happens as is evident in the next example too.

Example 53. For the NFA N2 given on the left below we construct the equivalent DFA M2 on the right, where $L(\text{N2}) = L(\text{M2})$.





In the above example notice that the empty subset $\emptyset$ is a state. This is sometimes called the *dead state* since no transitions are allowed out of it (and it is a non-accept state).

3.4 Regular Expressions

In this section we present a method for representing sets of strings over a fixed alphabet Σ. We begin with some formal definitions.

Definition 54. A *word* w over a given alphabet Σ is an element of $\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$. The *empty word* ϵ contains no symbols of Σ. A *language* L is a subset of words. The concatenation of two words w_1 and w_2, denoted $w_1 w_2$, is formed by juxtaposing the symbols. A *product of languages* L_1 and L_2 is $L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$. The *(Kleene) closure* of a language L is defined by $L^* = \bigcup_{i=0}^{\infty} L^i$.

The following property holds for the empty word ϵ and any word $w \in \Sigma^*$, $\epsilon w = w = w\epsilon$. For any language L, $L^0 = \{\epsilon\}$, $L^1 = L$ and $L^2 = LL$.

Example 55. If $\Sigma = \{0, 1\}$ and $L = \{0, 10\}$ then L^* is the set of words, including ϵ, that have at least one 0 following each 1.

Definition 56. The standard *regular expressions* over alphabet Σ (and the sets they designate) are as follows:

1. Any $c \in \Sigma$ is a regular expression (set $\{c\}$).

2. If E_1 (for some set S_1) and E_2 (for some set S_2) are regular expressions then so are:

   - $E_1 | E_2$ (union $S_1 \cup S_2$).
     Often denoted $E_1 + E_2$.

   - $E_1 E_2$ (language concatenation $S_1 S_2$).

   - E_1^* (Kleene closure S_1^*).

The following table illustrates several regular expressions over the alphabet $\Sigma = \{a, b, c\}$ and the sets of strings, which we will shortly call regular languages, that they represent.

regular expression	**regular language**
a	$\{a\}$
ab	$\{ab\}$
$a\|bb$	$\{a, bb\}$
$(a\|b)c$	$\{ac, bc\}$
c^*	$\{\epsilon, c, cc, ccc, \ldots\}$
$(a\|b\|c)cba$	$\{acba, bcba, ccba\}$
$a^*\|b^*\|c^*$	$\{\epsilon, a, b, c, aa, bb, cc, aaa, bbb, ccc, \ldots\}$
$(a\|b^*)c(c^*)$	$\{ac, acc, accc, \ldots, c, cc, ccc, \ldots, bc, bcc, bbccc, \ldots\}$

Definition 57. A *regular language (set)* over an alphabet Σ is either the empty set, the set $\{\epsilon\}$, or the set of words designated by some regular expression.

3.4.1 The UNIX extensions to regular expressions

For the users' convenience, the UNIX operating system extends (for programs such as vi, bash, grep, lex and perl) the regular expressions mentioned above for pattern matching. However, these new operators do not extend the sets of languages that are recognizable. Some of the most common new features are listed below for the alphabet Σ being the set of ASCII characters.

Character Classes and Wild Card Symbol. A range of characters can be enclosed in square brackets. For example `[a-z]` would denote the set of lower case letters. A period . is a wild card symbol used to denote any character except a newline.

Line Beginning and Ending. To match a string that begins the line use the hat ˆ as the first character of the pattern string. To match a string that ends the line use the dollar sign $ as the last character of the pattern string. For example `^[0-9]*$` will match both empty lines or lines containing only digits.

Additional Operators. Let E be a regular expression. The regular expression $E?$ denotes exactly 0 or 1 matches of E. This is shorthand for the regular expression $(\epsilon\ |E)$. The regular expression E^+ denotes EE^*, that is, 1 or more occurrences of E.

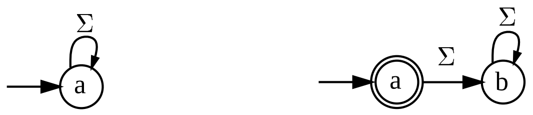
Note to match one of the special symbols above like * or . (instead of invoking its special feature) we have to escape it with a preceding backslash \ character. For example, `big.*\.` will match "bigest." and "biggy." where the period is matched. The line beginning and ending characters were added since, by default, most UNIX programs do substring matching.

3.5 Regular Sets and Finite-State Automata

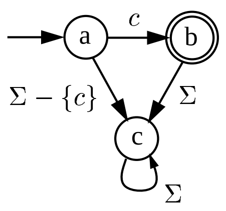
We now want to present a characterization of the computational power of finite state automata. We have already seen that DFA's and NFA's have the same computational power. The set of languages accepted/decided by automata are precisely the set of regular languages (sets). We show how to build an NFA that recognizes the set of words depicted by any regular expression.

Theorem 58 (Kleene's Theorem). For any regular language L there is a DFA M such that $L(M) = L$.

Proof. It suffices to find a NFA N that accepts L since we have already seen how to convert NFA's to DFA's. (See Section 3.3.)
An automaton for $L = \emptyset$ and an automaton for $L = \{\epsilon\}$ are given below.



Now suppose E is a regular expression for L. We construct N based on the length of E. If $E = \{c\}$ for some $c \in \Sigma$ we can use the following automaton.



By induction we only need to show how to construct N for E being one of $E_1 + E_2$, E_1E_2 or E_1^*, for smaller regular expressions E_1 and E_2. Let use assume we have correct automata M1 and M2 for E_1 and E_2.

Case 1: $E = E_1 + E_2$
We construct a (nondeterministic) automaton N representing E simply by taking the union of the two machines M1 and M2.

Case 2: $E = E_1E_2$
We construct an automaton N representing E as follows. We do this by altering slightly the union of the two machines M1 and M2. The initial states of N will be the initial states of M1. The the initial states of M2 will only initial states of N if at least one of M1's initial states is an accepting state. The final states of N will be the final states of M2. (I.e., the final states of M1 become ordinary states.) For each transistion (q_1, q_2) to a final state q_2 of M1 we add transitions to the initial states of M2. That is, for $c \in \Sigma$, if $q_{1_j} \in \delta_1(q_{1_i}, c)$ for some final state $q_{1_j} \in F_1$ then $q_{2_k} \in \delta_N(q_1, c)$ for each start state $q_{2_k} \in S_2$.

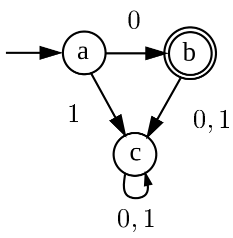
Case 3: $E = E_1^*$

The closure of an automaton was seen in Section 3.2.3. An automaton representing E is the union of the closure $C(M1)$ and the automaton representing $\{\epsilon\}$ given above. $\square$

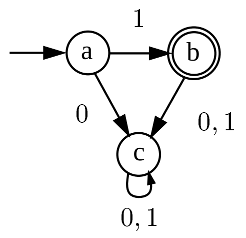
Kleene's Theorem is actually stronger than what we mentioned above. He also proved that for any finite automaton M there exists a regular expression that represents the language decided by M. The construction is simple, but detailed, and is less useful so we omit it.

Example 59. For the regular expression $(01)^* + 1$ we construct an automaton that accepts the strings matched. First we build automata M1 and M2 that accept the simple languages $\{0\}$ and $\{1\}$.

M1:

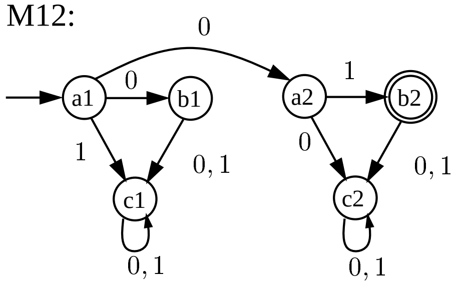


M2:

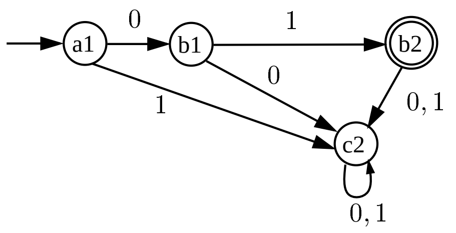


We next construct an automaton M12 that accepts the language $\{01\}$. We can easily reduce the number of states of M12 to create an equivalent automaton M3.

M12:

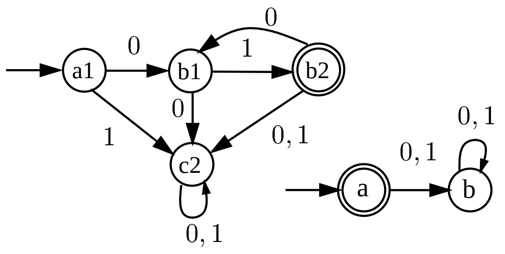


M3:

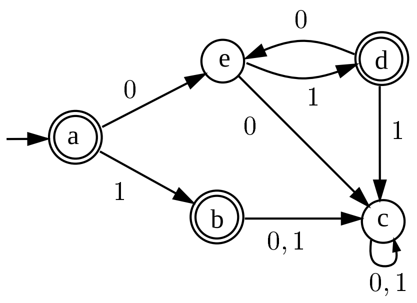


We next construct an automaton M4 that accepts the language represented by the regular expression $(01)^*$.

M4:

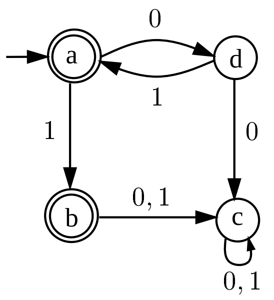


The union of the automata M2 and M4 is an automaton N that accepts the regular language depicted by the expression $(01)^* + 1$. In the next section we show how to minimize automata to produce the following final deterministic automaton (from the output of algorithm NFA2DFA on N) that accepts this language.



Usually more complicated (longer length) regular expressions require automata with more states. However, this is not true in general.

Example 60. The DFA for the regular expression $(01)^*(\epsilon + 1)$, displayed below, has one fewer state than the previous example.



3.6 Minimizing Deterministic Finite-State Machines

There are standard techniques for minimizing deterministic automata. We present an efficient algorithm based on finding (and eliminating) equivalent states.

Definition 61. For a DFA $M = (Q, \Sigma, \delta, s, F)$ and any $q \in Q$ define the DFA $M_q = (Q, \Sigma, \delta, q, F)$, that is, s is replaced with q in M_q. We say two states p and q of M are *distinguishable (k-distinguishable)* if there exists a string $w \in \Sigma^*$ (of length k) such that exactly one of M_p or M_q accepts w. If there is no such string w then we say p and q are *equivalent*.

Note that the empty string ϵ may also be used to distinguish two states of an automaton.

Lemma 62. If a DFA M has two equivalent states p and q then there exists a smaller DFA M′ such that $L(M) = L(M')$.

Proof. Assume $M = (Q, \Sigma, \delta, s, F)$ and $p \neq s$. We create an equivalent DFA M′ $= (Q \setminus \{p\}, \Sigma, \delta', s, F \setminus \{p\})$ where δ' is δ with all instances of $\delta(q_i, c) = p$ replaced with $\delta'(q_i, c) = q$ and all instances of $\delta(p, c) = q_i$ deleted. The resulting automaton M′ is deterministic and accepts language $L(M)$. □

Lemma 63. Two states p and q are (not) k-distinguishable if and only if for each $c \in \Sigma$, the states $\delta(p, c)$ and $\delta(q, c)$ are (not) $(k-1)$-distinguishable.

Proof. Consider all strings $w = cw'$ of length k. If $\delta(p, c)$ and $\delta(q, c)$ are $(k-1)$-distinguishable by some string w' then p and q must be k-distinguishable by w. Likewise, p and q being k-distinguishable by w implies there exists two states $\delta(p, c)$ and $\delta(q, c)$ that are $(k-1)$-distinguishable by the shorter string w'. $\square$

Algorithm MinimizeDFA:

Our algorithm to find equivalent states of a DFA $M = (Q, \Sigma, \delta, s, F)$ begins by defining a series equivalent relations $\equiv_0, \equiv_1, \ldots$ on the states of Q.

$p \equiv_0 q$	if both p and q are in F or both not in F.
$p \equiv_{k+1} q$	if $p \equiv_k q$ and, for each $c \in \Sigma$, $\delta(p, c) \equiv_k \delta(q, c)$.

We stop generating these equivalence classes when $\equiv_n$ and $\equiv_{n+1}$ are identical. Lemma 63 guarantees no more non-equivalent states. Since there can be at most $|Q|$ non-equivalent states this bounds the number of equivalence relations $\equiv_k$ generated. We can eliminated one state from M (using Lemma 62) whenever there exists two states p and q such that $p \equiv_n q$. In practice, we often eliminate more than one (i.e., all but one) state per equivalence class.

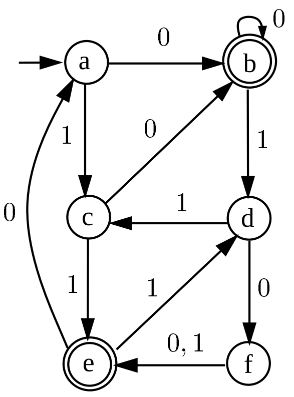
Theorem 64. There exists a polynomial-time algorithm to minimize any DFA M.

Proof. To compute $\equiv_{k+1}$ from $\equiv_k$ we have to determine the equivalence (or non-equivalence) for at most $\binom{|Q|}{2} = O(|Q|^2)$ possible pairs of states p and q. Each equivalence check requires $2|\Sigma|$ transitions look-ups. Since we have to compute this for at most $n \leq |Q|$ different equivalence classes $\equiv_k$, the preceding algorithm MinimizeDFA runs in time $O(|\Sigma| \cdot |Q|^3)$. $\square$

Currently there are no direct, efficient minimization algorithms for the nondeterministic counterparts of DFA. Note that the minimized equivalent DFA for an NFA may be larger then the original (nonminimized) NFA.

We end our introduction to automata theory by showing how to use this minimization algorithm. The first example shows how to verify that an automaton is minimal and the second shows how to find equivalent states for elimination.

Example 65. We use the algorithm MinimizeDFA to show the following automaton M has the smallest number of states for the regular language it represents.



The initial equivalent relation $\equiv_0$ is $\{a, c, d, f\}\{b, e\}$ based solely on the final states of M. We now calculate $\equiv_1$ using the recursive definition:

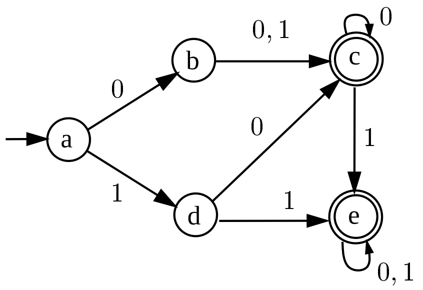
$$\begin{gathered}
\delta(a,1) = c \not\equiv_0 \delta(c,1) = e \;\Rightarrow\; a \not\equiv_1 c. \\
\delta(a,0) = e \not\equiv_0 \delta(d,0) = f \;\Rightarrow\; a \not\equiv_1 d. \\
\delta(a,1) = c \not\equiv_0 \delta(f,1) = e \;\Rightarrow\; a \not\equiv_1 f. \\
\delta(c,0) = b \not\equiv_0 \delta(d,0) = f \;\Rightarrow\; c \not\equiv_1 d. \\
(\delta(c,0) = b \equiv_0 \delta(f,0) = e \text{ and } \delta(c,1) = e \equiv_0 \delta(f,1) = e) \;\Rightarrow\; c \equiv_1 f. \\
\delta(d,0) = f \not\equiv_0 \delta(f,0) = e \;\Rightarrow\; d \not\equiv_1 f. \\
\delta(b,0) = b \not\equiv_0 \delta(e,0) = a \;\Rightarrow\; b \not\equiv_1 e.
\end{gathered}$$

So $\equiv_1$ is $\{a\}\{b\}\{c, f\}\{d\}\{e\}$. We now calculate $\equiv_2$ to check the two possible remaining equivalent states:

$$\delta(c,0) = b \not\equiv_1 \delta(f,0) = e \;\Rightarrow\; c \not\equiv_2 f.$$

This shows that all states of M are non-equivalent (i.e., our automaton is minimum).

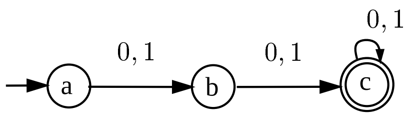
Example 66. We use the algorithm MinimizeDFA to show that the following automaton M2 can be reduced.



The initial equivalent relation $\equiv_0$ is $\{a, b, d\}\{c, e\}$. We now calculate $\equiv_1$:

$$\begin{gathered}
\delta(a,0) = b \not\equiv_0 \delta(b,0) = c \;\Rightarrow\; a \not\equiv_1 b. \\
\delta(a,0) = b \not\equiv_0 \delta(d,0) = c \;\Rightarrow\; a \not\equiv_1 d. \\
(\delta(b,0) = c \equiv_0 \delta(d,0) = c \text{ and } \delta(b,1) = c \equiv_0 \delta(d,1) = e) \;\Rightarrow\; b \equiv_1 d. \\
(\delta(c,0) = c \equiv_0 \delta(e,0) = e \text{ and } \delta(c,1) = e \equiv_0 \delta(e,1) = e) \;\Rightarrow\; c \equiv_1 e.
\end{gathered}$$

So $\equiv_1$ is $\{a\}\{b,d\}\{c,e\}$. We calculate $\equiv_2$ in the same fashion and see that it is the same as $\equiv_1$. This shows that we can eliminate, say, states d and e to yield the following minimum DFA that recognizes the same language as M2 does.



To test ones understanding, we invite the reader reproduce the final automaton of Example 59 by using the algorithms NFA2DFA and MinimizeDFA.