Algorithm 3.2. (*Subset construction.*) Constructing a DFA from an NFA.

Input. An NFA N.

Output. A DFA D accepting the same language.

Method. Our algorithm constructs a transition table *Dtran* for D. Each DFA state is a set of NFA states and we construct *Dtran* so that D will simulate "in parallel" all possible moves N can make on a given input string.

We use the operations in Fig. 3.24 to keep track of sets of NFA states (s represents an NFA state and T a set of NFA states).

OPERATION	DESCRIPTION
ϵ-*closure*(s)	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
ϵ-*closure*(T)	Set of NFA states reachable from some NFA state s in T on ϵ-transitions alone.
move(T, a)	Set of NFA states to which there is a transition on input symbol a from some NFA state s in T.

Fig. 3.24. Operations on NFA states.

Before it sees the first input symbol, N can be in any of the states in the set ϵ-*closure*(s_0), where s_0 is the start state of N. Suppose that exactly the states in set T are reachable from s_0 on a given sequence of input symbols, and let a be the next input symbol. On seeing a, N can move to any of the states in the set *move*(T, a). When we allow for ϵ-transitions, N can be in any of the states in ϵ-*closure*(*move*(T, a)), after seeing the a.

```
initially, ε-closure(s0) is the only state in Dstates and it is unmarked;
while there is an unmarked state T in Dstates do begin
      mark T;
      for each input symbol a do begin
            U := ε-closure(move(T, a));
            if U is not in Dstates then
                  add U as an unmarked state to Dstates;
            Dtran[T, a] := U
      end
end
```

Fig. 3.25. The subset construction.

We construct *Dstates*, the set of states of D, and *Dtran*, the transition table for D, in the following manner. Each state of D corresponds to a set of NFA states that N could be in after reading some sequence of input symbols including all possible ϵ-transitions before or after symbols are read. The start state of D is ϵ-*closure*(s_0). States and transitions are added to D using the algorithm of Fig. 3.25. A state of D is an accepting state if it is a set of NFA states containing at least one accepting state of N.

```
push all states in T onto stack;
initialize ε-closure(T) to T;
while stack is not empty do begin
        pop t, the top element, off of stack;
        for each state u with an edge from t to u labeled ε do
             if u is not in ε-closure(T) do begin
                  add u to ε-closure(T);
                  push u onto stack
             end
end
```

Fig. 3.26. Computation of ϵ-*closure*.

The computation of ϵ-*closure*(T) is a typical process of searching a graph for nodes reachable from a given set of nodes. In this case the states of T are the given set of nodes, and the graph consists of just the ϵ-labeled edges of the NFA. A simple algorithm to compute ϵ-*closure*(T) uses a stack to hold states whose edges have not been checked for ϵ-labeled transitions. Such a procedure is shown in Fig. 3.26. □

Example 3.15. Figure 3.27 shows another NFA N accepting the language $(a|b)^*abb$. (It happens to be the one in the next section, which will be mechanically constructed from the regular expression.) Let us apply Algorithm 3.2 to N. The start state of the equivalent DFA is ϵ-*closure*(0), which is $A = \{0, 1, 2, 4, 7\}$, since these are exactly the states reachable from state 0 via a path in which every edge is labeled ϵ. Note that a path can have no edges, so 0 is reached from itself by such a path.

The input symbol alphabet here is $\{a, b\}$. The algorithm of Fig. 3.25 tells us to mark A and then to compute

$$\epsilon\text{-}closure(move(A,\ a)).$$

We first compute *move*(A, a), the set of states of N having transitions on a from members of A. Among the states 0, 1, 2, 4 and 7, only 2 and 7 have such transitions, to 3 and 8, so

$$\epsilon\text{-}closure(move(\{0, 1, 2, 4, 7\},\ a)) = \epsilon\text{-}closure(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$$

Let us call this set B. Thus, $Dtran[A,\ a] = B$.

Among the states in A, only 4 has a transition on b to 5, so the DFA has a transition on b from A to

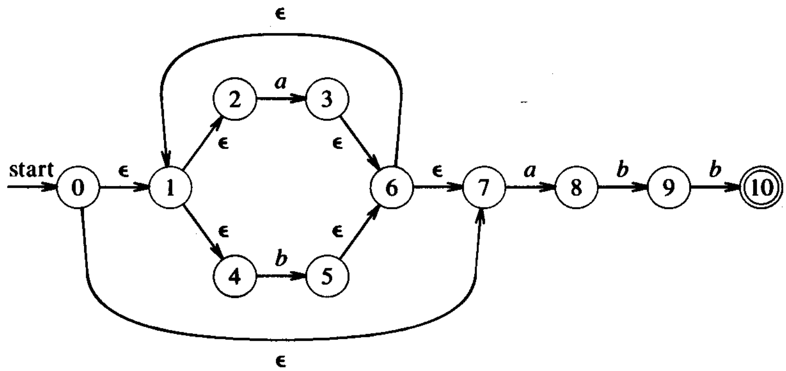

Fig. 3.27. NFA N for $(a|b)^*abb$.

$$C = \epsilon\text{-}closure(\{5\}) = \{1, 2, 4, 5, 6, 7\}$$

Thus, $Dtran[A, b] = C$.

If we continue this process with the now unmarked sets B and C, we eventually reach the point where all sets that are states of the DFA are marked. This is certain since there are "only" 2^{11} different subsets of a set of eleven states, and a set, once marked, is marked forever. The five different sets of states we actually construct are:

$$\begin{array}{ll} A = \{0, 1, 2, 4, 7\} & D = \{1, 2, 4, 5, 6, 7, 9\} \\ B = \{1, 2, 3, 4, 6, 7, 8\} & E = \{1, 2, 4, 5, 6, 7, 10\} \\ C = \{1, 2, 4, 5, 6, 7\} & \end{array}$$

State A is the start state, and state E is the only accepting state. The complete transition table *Dtran* is shown in Fig. 3.28.

STATE	INPUT SYMBOL	
	a	*b*
A	*B*	*C*
B	*B*	*D*
C	*B*	*C*
D	*B*	*E*
E	*B*	*C*

Fig. 3.28. Transition table *Dtran* for DFA.

Also, a transition graph for the resulting DFA is shown in Fig. 3.29. It should be noted that the DFA of Fig. 3.23 also accepts $(a|b)^*abb$ and has one

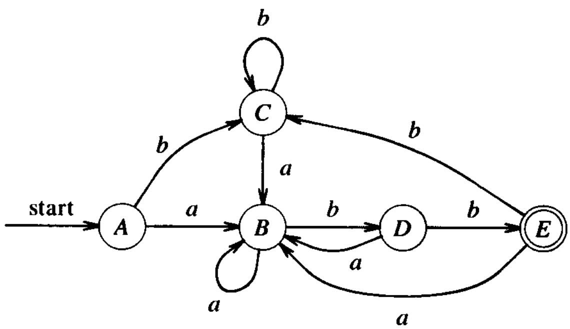

Fig. 3.29. Result of applying the subset construction to Fig. 3.27.

fewer state. We discuss the question of minimization of the number of states of a DFA in Section 3.9. □

3.7 FROM A REGULAR EXPRESSION TO AN NFA

There are many strategies for building a recognizer from a regular expression, each with its own strengths and weaknesses. One strategy that has been used in a number of text-editing programs is to construct an NFA from a regular expression and then to simulate the behavior of the NFA on an input string using Algorithms 3.3 and 3.4 of this section. If run-time speed is essential, we can convert the NFA into a DFA using the subset construction of the previous section. In Section 3.9, we see an alternative implementation of a DFA from a regular expression in which an intervening NFA is not explicitly constructed. This section concludes with a discussion of time-space tradeoffs in the implementation of recognizers based on NFA and DFA.

Construction of an NFA from a Regular Expression

We now give an algorithm to construct an NFA from a regular expression. There are many variants of this algorithm, but here we present a simple version that is easy to implement. The algorithm is syntax-directed in that it uses the syntactic structure of the regular expression to guide the construction process. The cases in the algorithm follow the cases in the definition of a regular expression. We first show how to construct automata to recognize ϵ and any symbol in the alphabet. Then, we show how to construct automata for expressions containing an alternation, concatenation, or Kleene closure operator. For example, for the expression $r|s$, we construct an NFA inductively from the NFA's for r and s.

As the construction proceeds, each step introduces at most two new states, so the resulting NFA constructed for a regular expression has at most twice as many states as there are symbols and operators in the regular expression.