Java II
Finite Automata I

Bernd Kiefer

`Bernd.Kiefer@dfki.de`

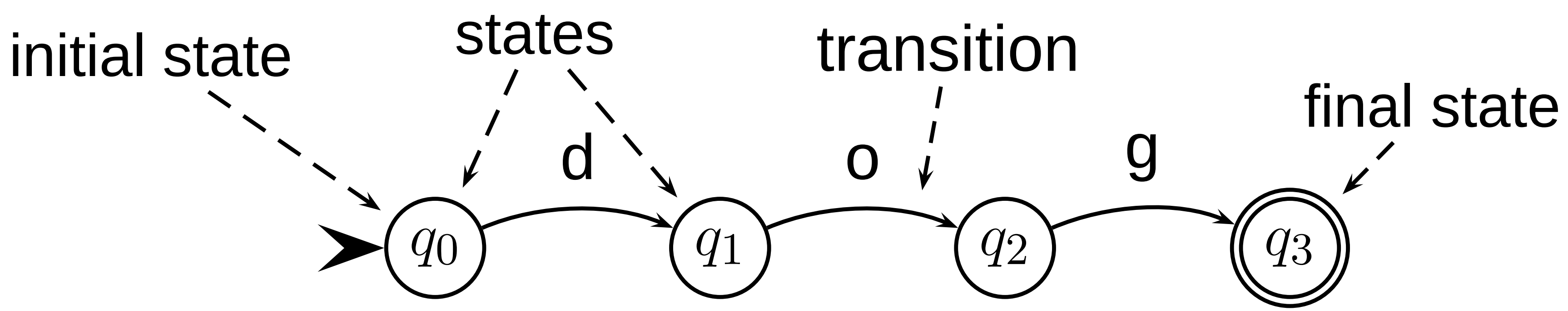
Deutsches Forschungszentrum für künstliche Intelligenz

Processing Regular Expressions

- We already learned about Java's regular expression functionality
- Now we get to know the machinery behind
  - `Pattern` and
  - `Matcher` classes
- Compiling a regular expression into a `Pattern` object produces a *Finite Automaton*
- This automaton is then used to perform the matching tasks
- We will see how to construct a finite automaton that *recognizes* an input string, i.e., tries to find a full match

Definition: Finite Automaton

- A finite automaton (FA) is a tuple $A = < Q, \Sigma, \delta, q_0, F >$
  - Q a finite non-empty set of states
  - Σ a finite alphabet of input letters
  - δ a (total) transition function $Q \times \Sigma \longrightarrow Q$
  - $q_0 \in Q$ the initial state
  - $F \subseteq Q$ the set of final (accepting) states
- Transition graphs (diagrams):



Finite Automata: Matching

- A finite automaton *accepts* a given input string s if there is a sequence of states $p_1, p_2, \ldots, p_{|s|} \in Q$ such that
  1. $p_1 = q_0$, the start state
  2. $\delta(p_i, s_i) = p_{i+1}$, where s_i is the i-th character in s
  3. $p_{|s|} \in F$, i.e., a final state
- A string is successfully *matched* if we have found the appropriate sequence of states
- Imagine the string on an input tape with a pointer that is advanced when using a δ transition
- The set of strings accepted by an automaton is the accepted *language*, analogous to regular expressions

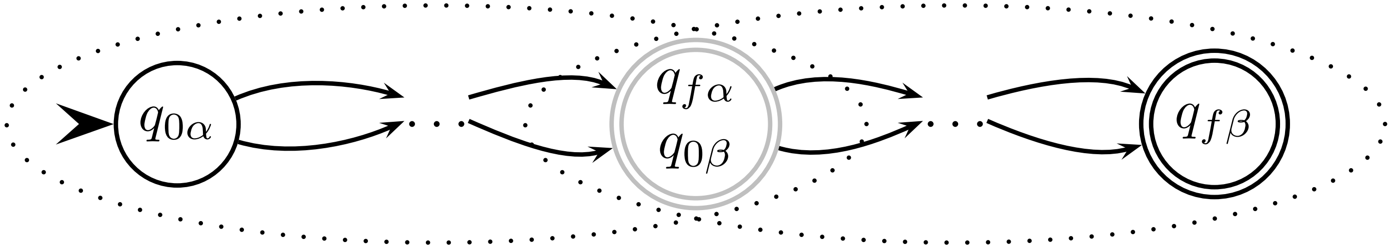
(Non)deterministic Automata

- in the definition of automata, δ was a total function $\Rightarrow$

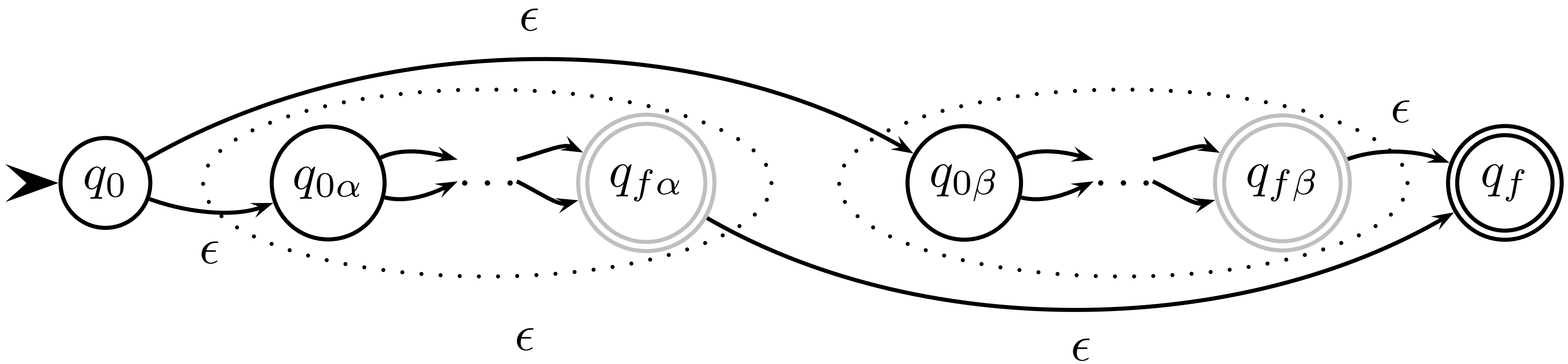
  given an input string, the path through the automaton is uniquely determined
- those automata are therefore called *deterministic*
- for *nondeterministic FA*, δ is a *transition relation*
- $\delta : Q \times \Sigma \cup \{\epsilon\} \longrightarrow \mathcal{P}(Q)$, where $\mathcal{P}(Q)$ is the powerset of Q
- allows transitions from one state into several states with the same input symbol
- need not be total
- can have transitions labeled ϵ (not in Σ), which represents the empty string

Construct nondeterminstic automata from regular expressions

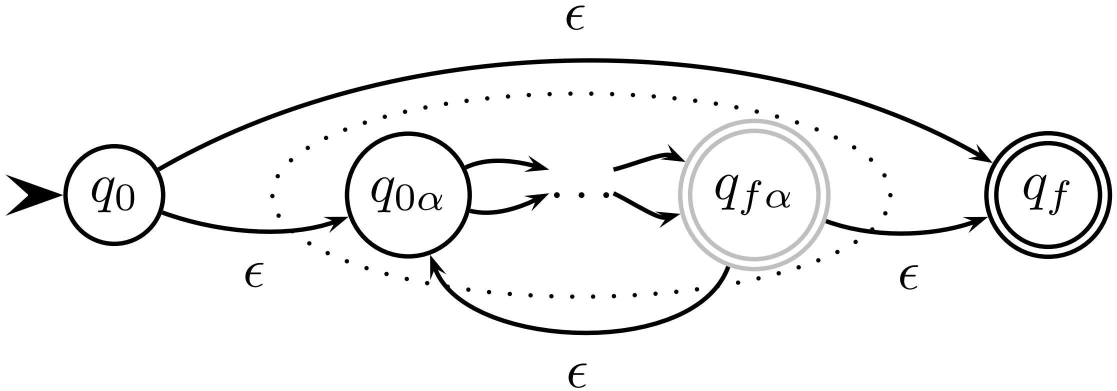
$(\alpha\beta)$



$(\alpha \mid \beta)$



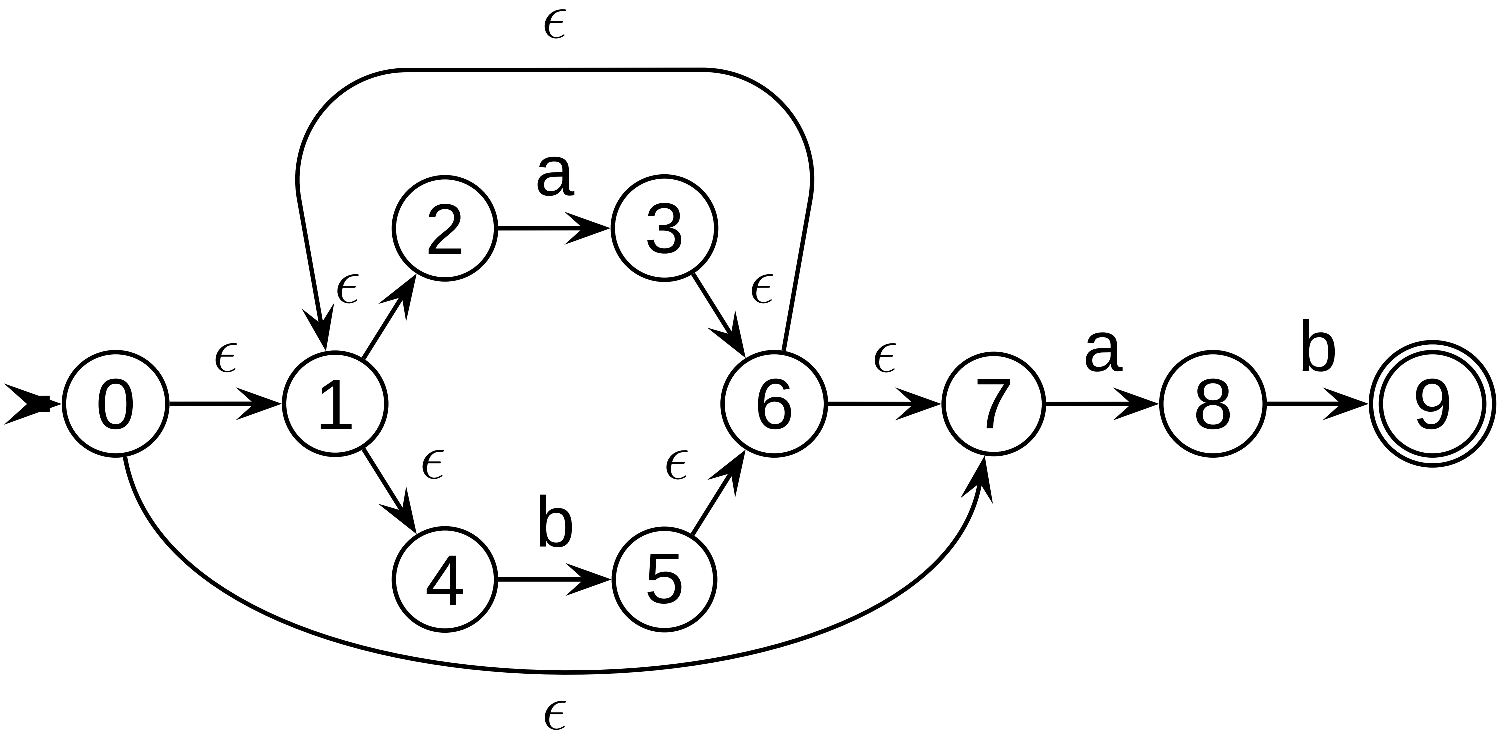
$(\alpha)*$



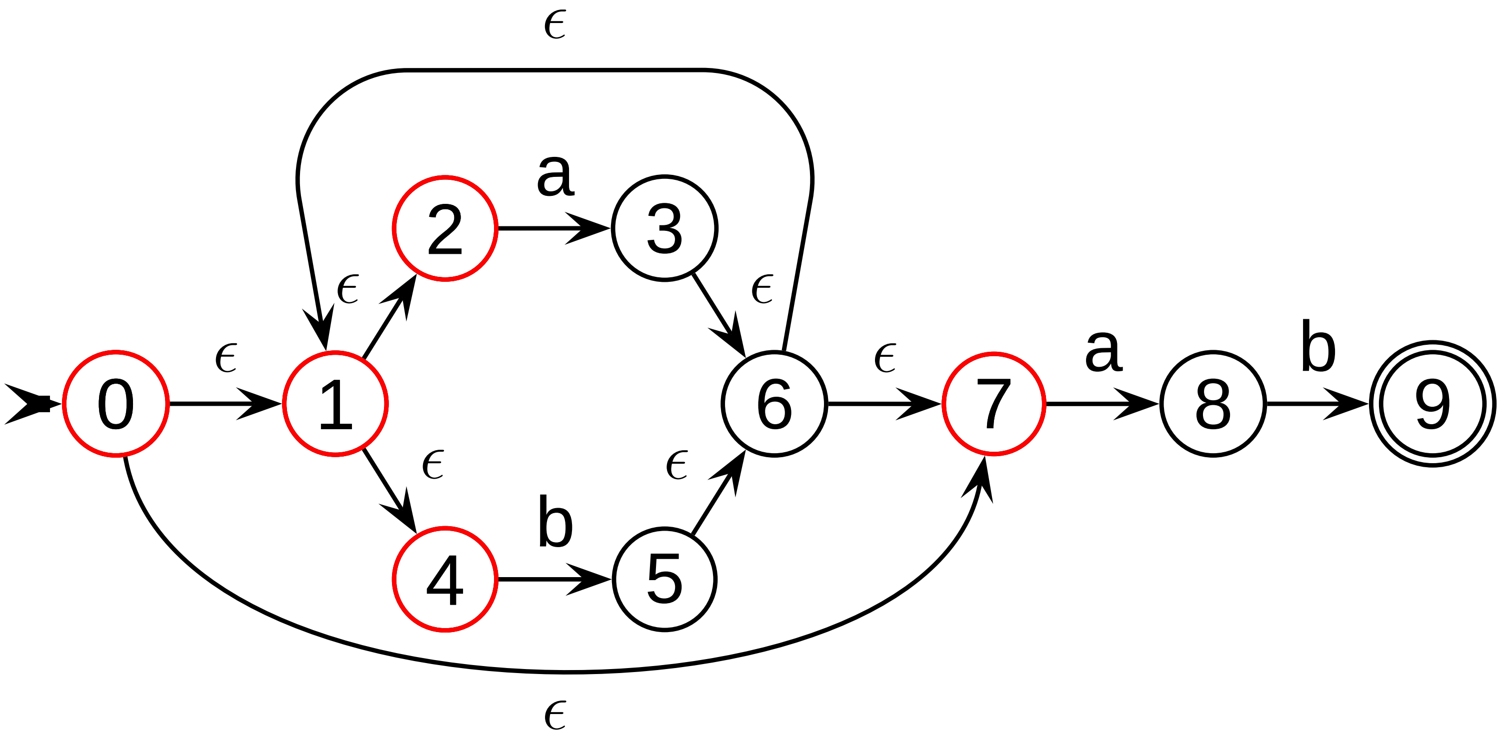
NFA vs. DFA

- Traversing a DFA is easy given the input string: the path is uniquely determined
- In contrast, traversing an NFA requires keeping track of a set of (current) states, starting with the set $\{q_o\}$
- Processing the next input symbol means taking all possible outgoing transitions from this set and collecting the new set
- From every NFA, an equivalent DFA (one which does accept the same language), can be computed
- Basic Idea: track the subsets that can be reached for every possible input

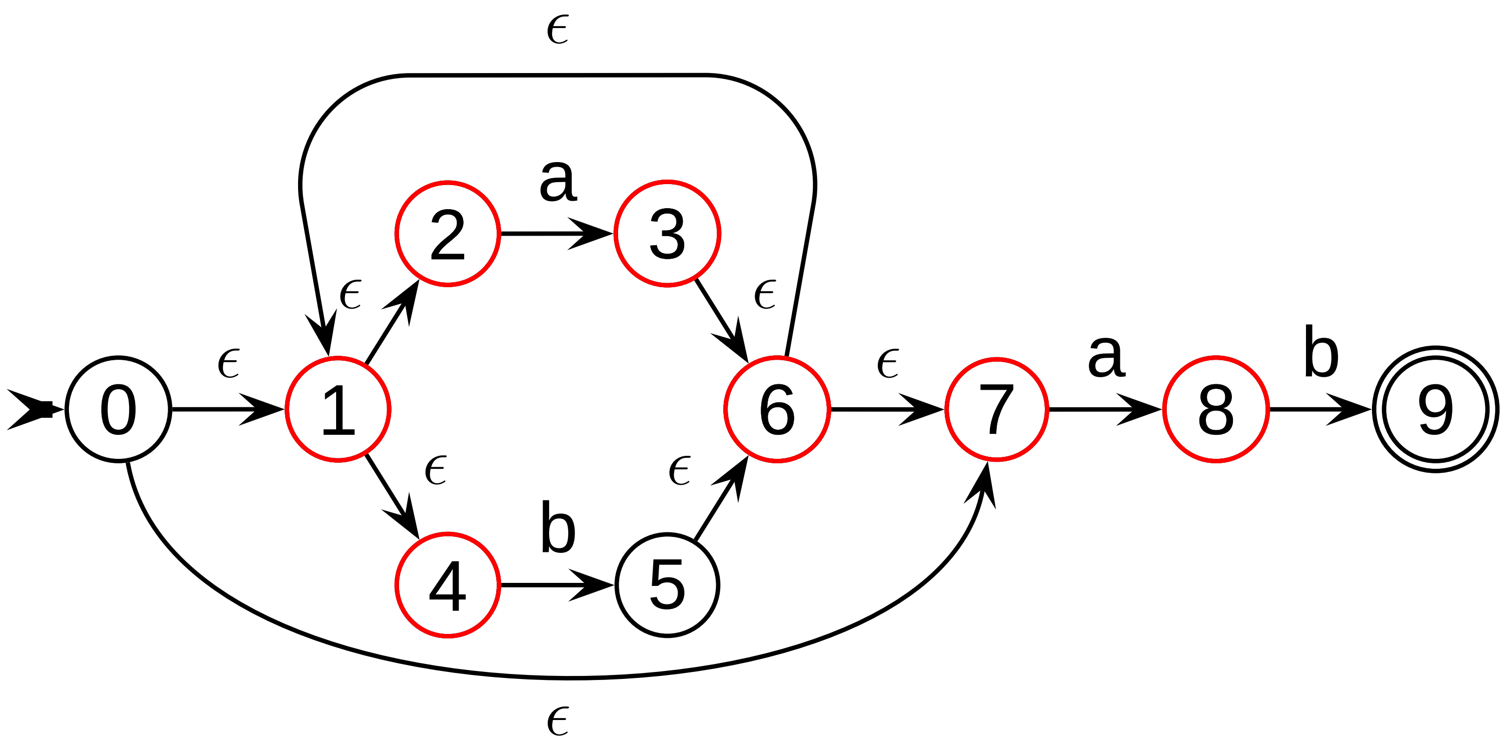
Traversing an NFA



abab

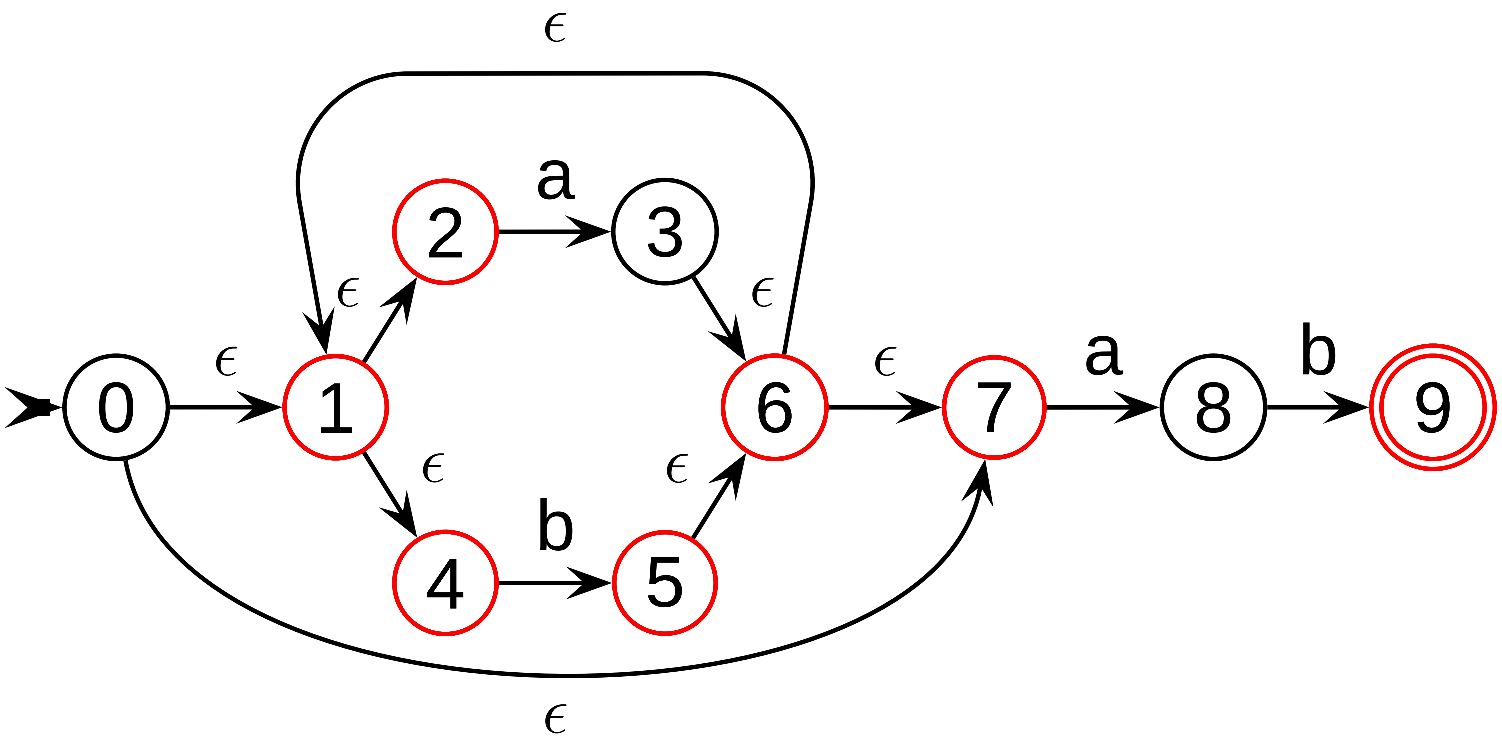


abab

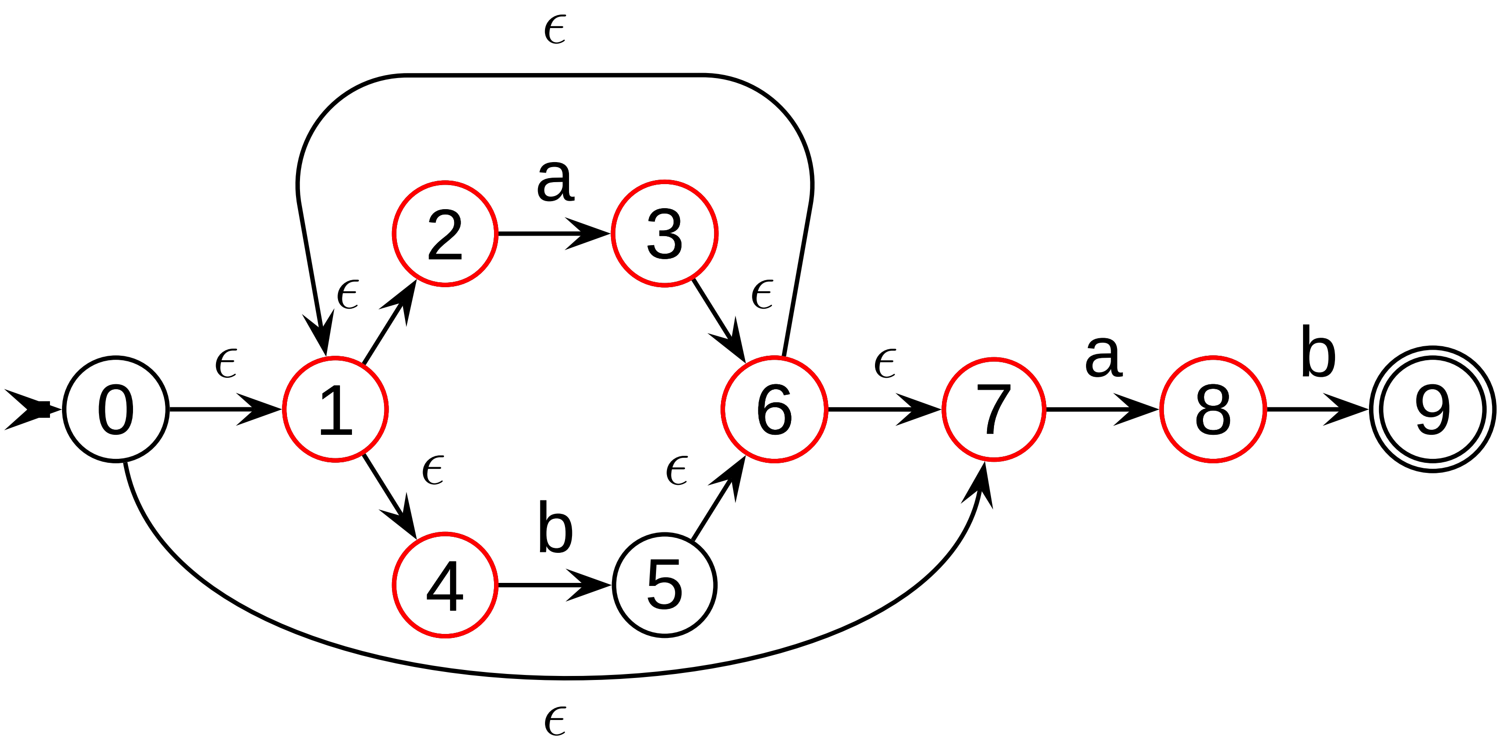


abab

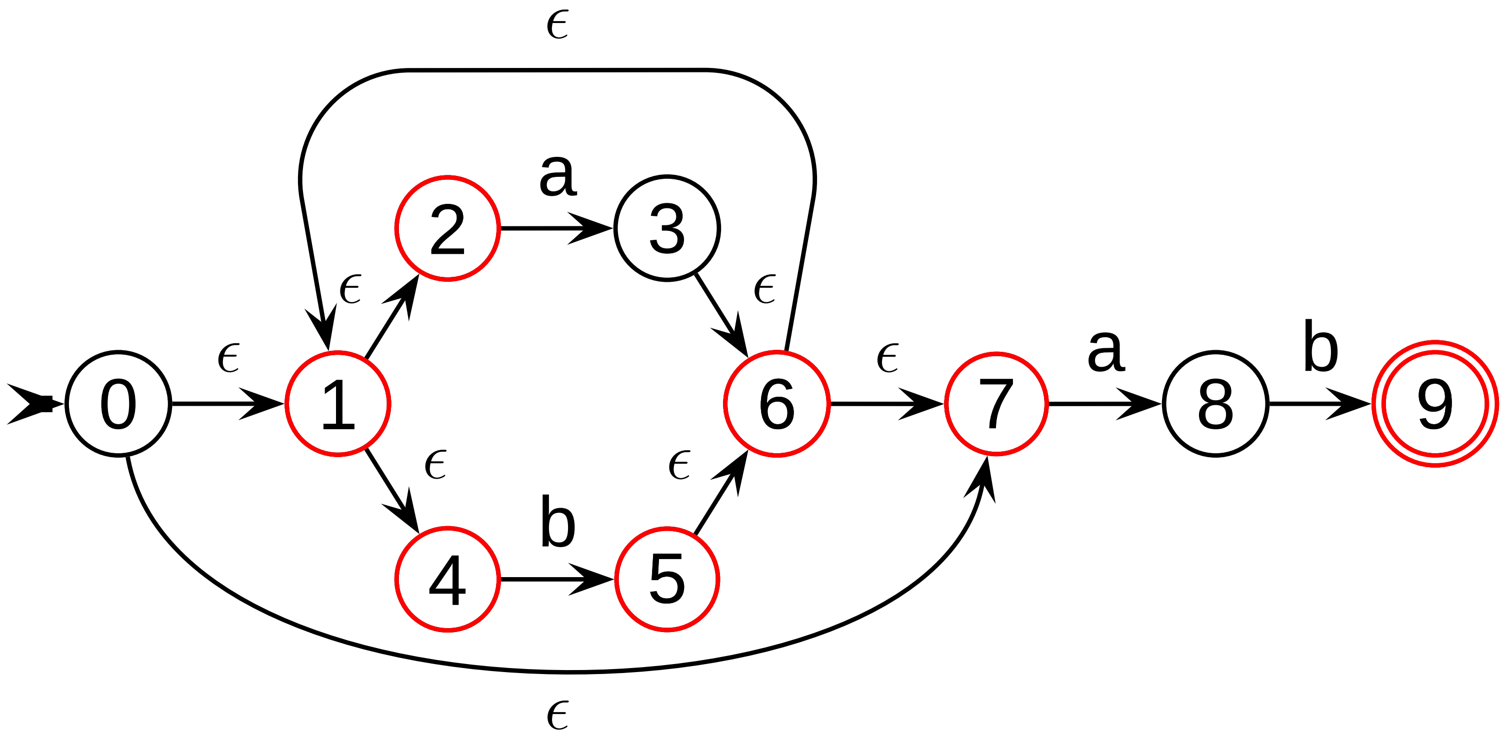
Traversing an NFA



ababab



ababb



abab

NFA $\longrightarrow$ DFA: Subset Construction

- Simulate “in parallel” all possible moves the automaton can make
- The states of the resulting DFA will represent sets of states of the NFA, i.e., elements of $\mathcal{P}(Q)$
- We use two operations on states/state-sets of the NFA

ϵ-*closure*(T)	Set of states reachable from any state s in T on ϵ-transitions
move(T, a)	Set of states to which there is a transition from one state in T on input symbol a

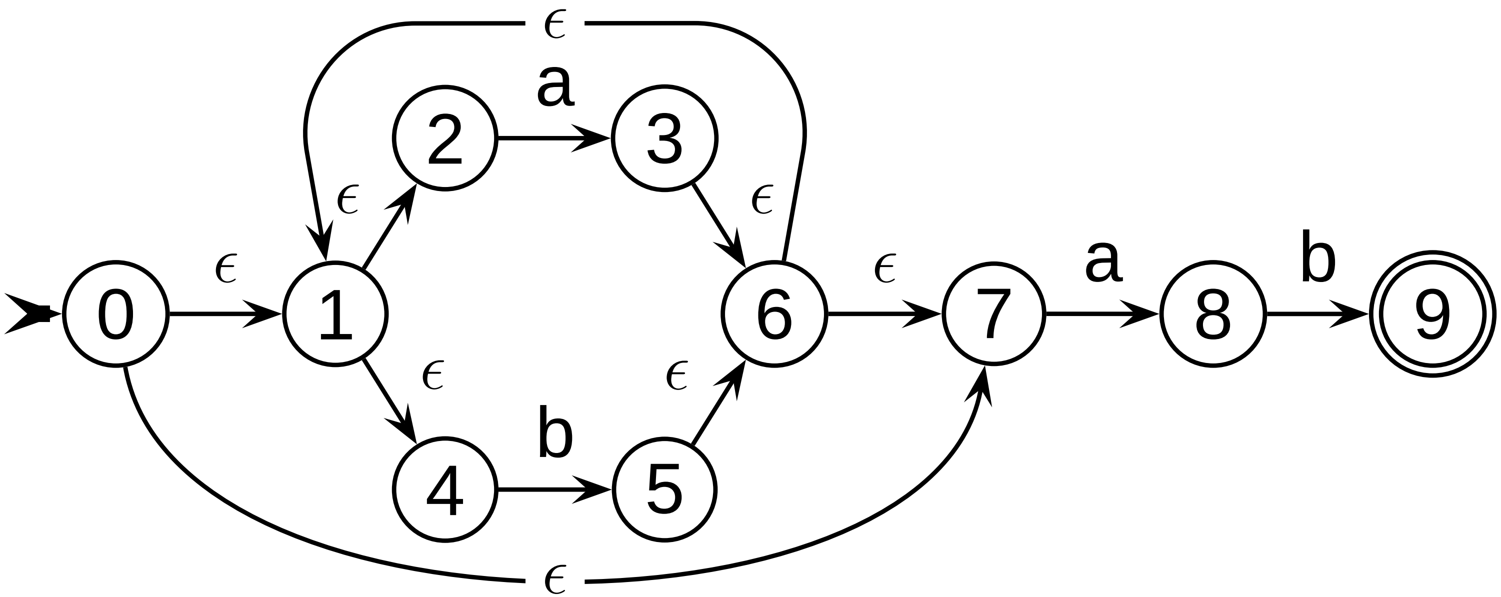
- The final states of the DFA are those where the corresponding NFA subset contains a final state

Algorithm: Subset Construction

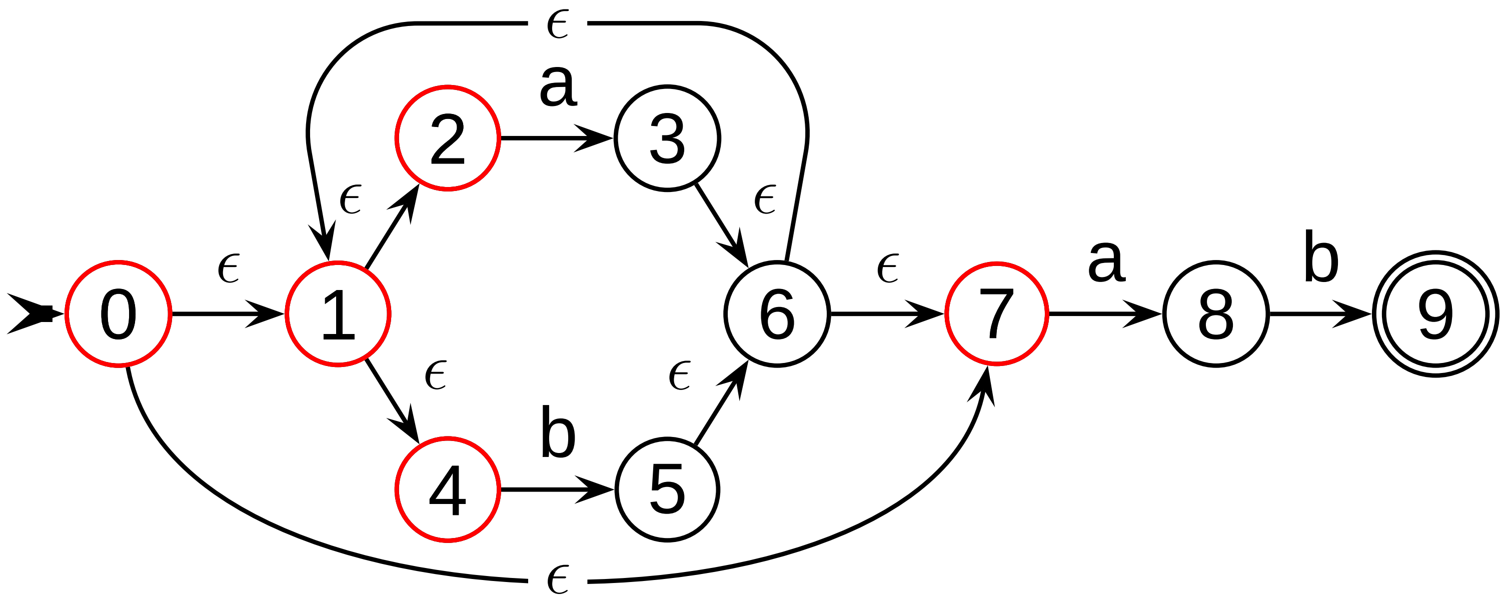
proc *SubsetConstruction*(s_0) $\equiv$
 DFAStates $=$ ϵ-*closure*$(\{s_0\})$
 while there is an unmarked state T in *DFAStates* **do**
  mark T
  for each input symbol a **do**
   U $:=$ ϵ-*closure*(*move*(T, a))
   DFADelta$[T, a]$ $:=$ U
   if $U \notin$ *DFAStates* **then** add U as unmarked state to *DFAStates*

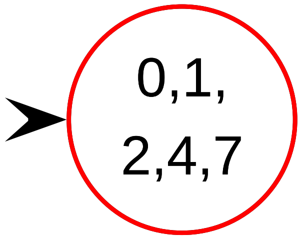
proc ϵ-*closure*(T) $\equiv$
 ϵ-*closure* $:=$ T; *to_check* $:=$ T
 while *to_check* not empty **do**
  get some state t from *to_check*
  for each state u with edge labeled ϵ from t to u
   if $u \notin$ ϵ-*closure* **then** add u to ϵ-*closure* and *to_check*

Example: Subset Construction

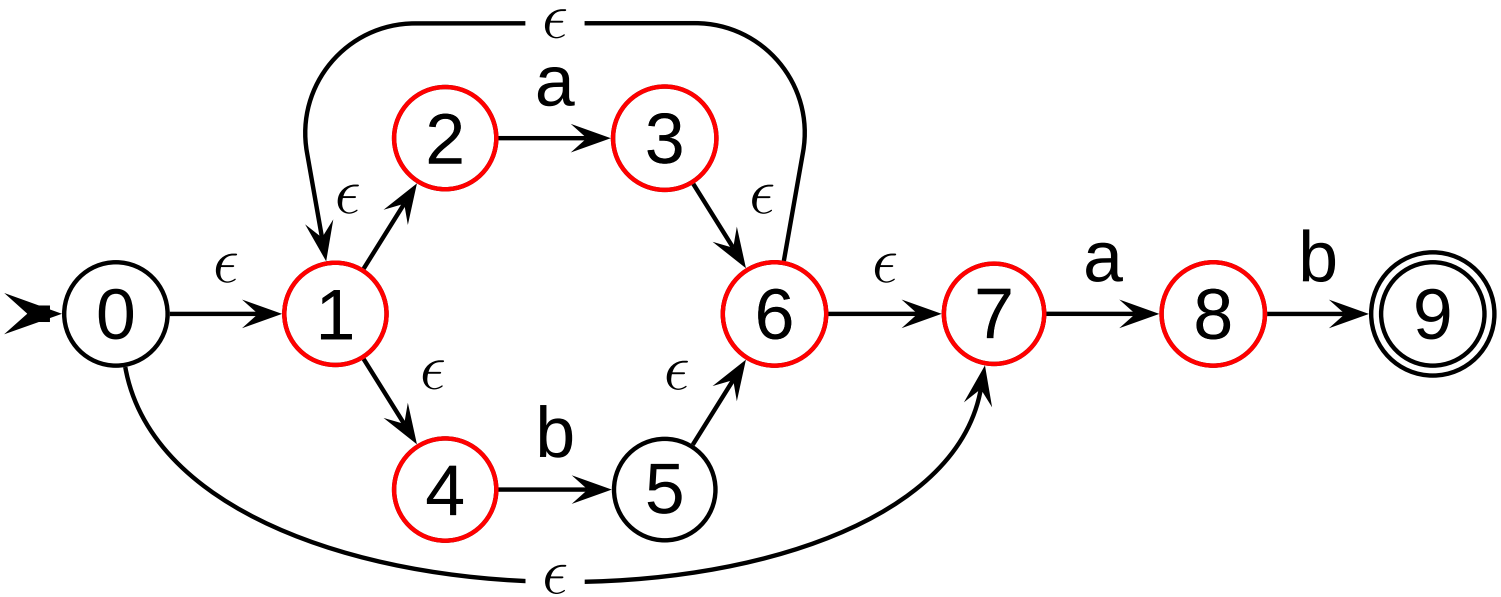


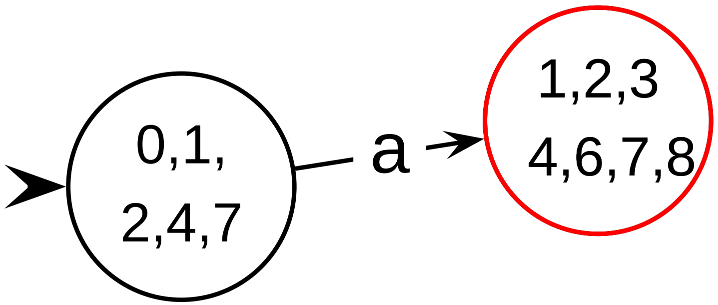
Example: Subset Construction



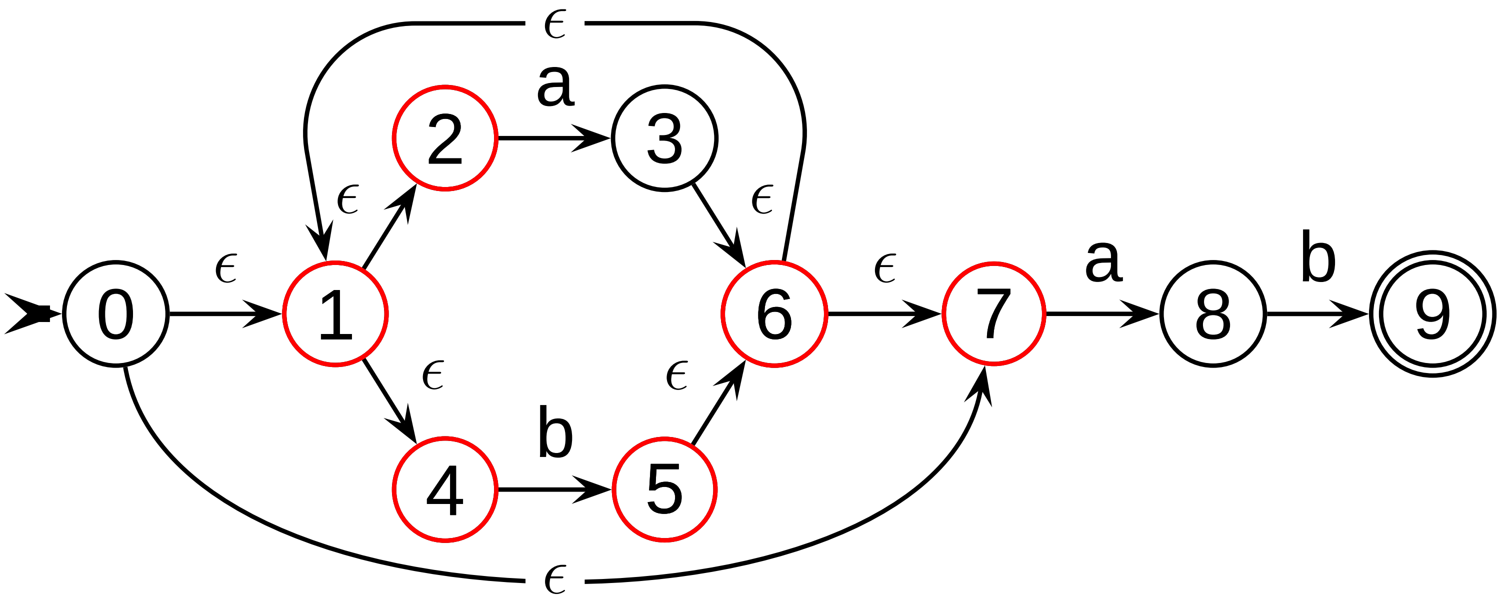


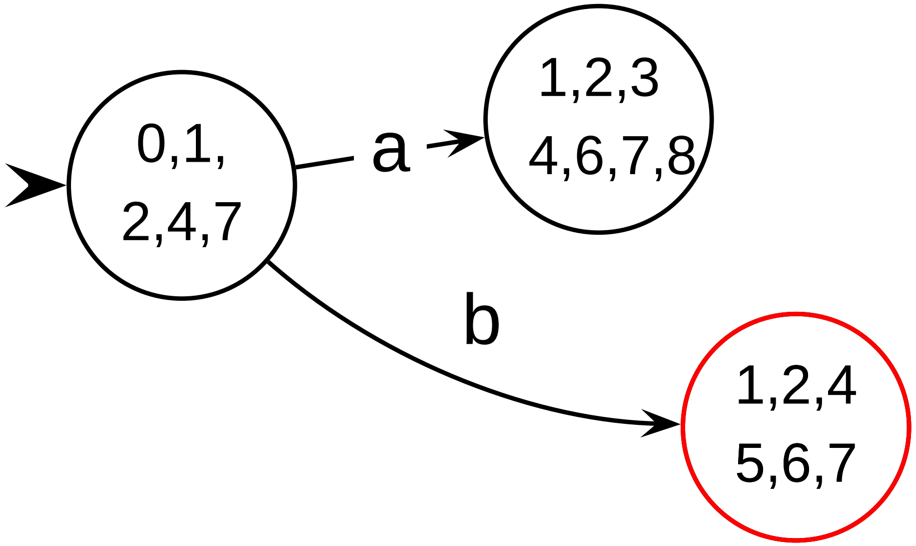
Example: Subset Construction



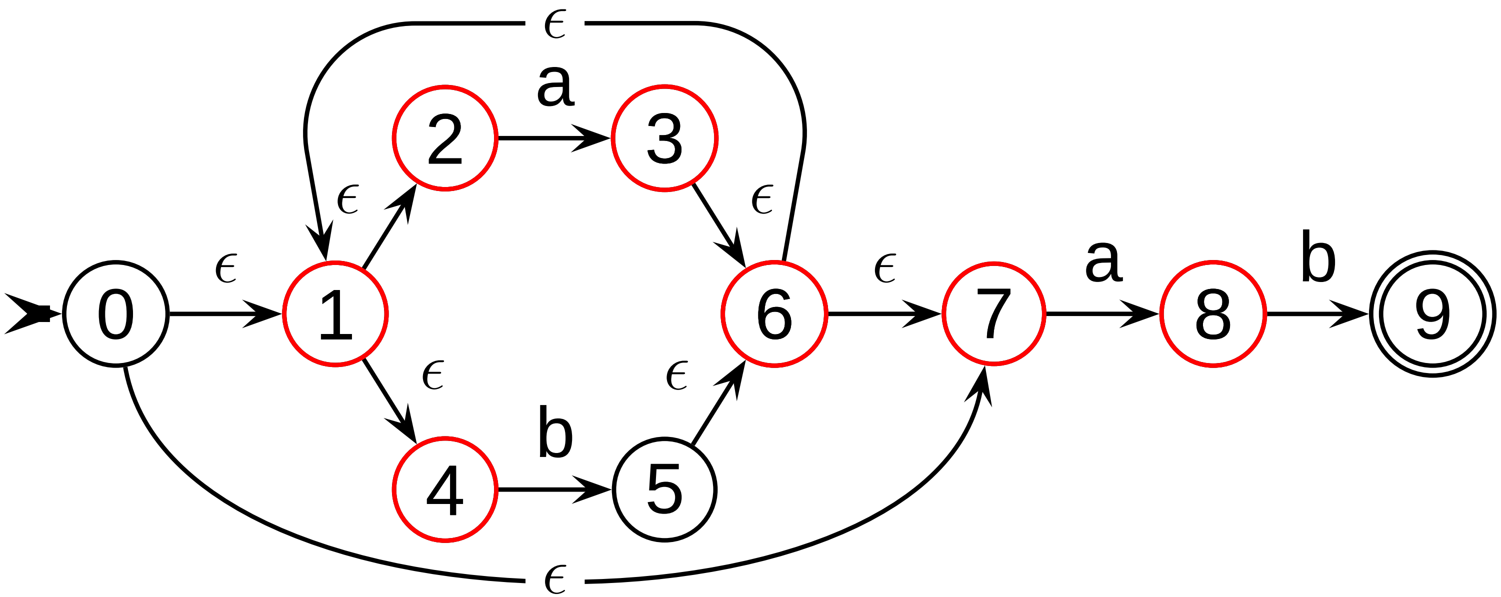


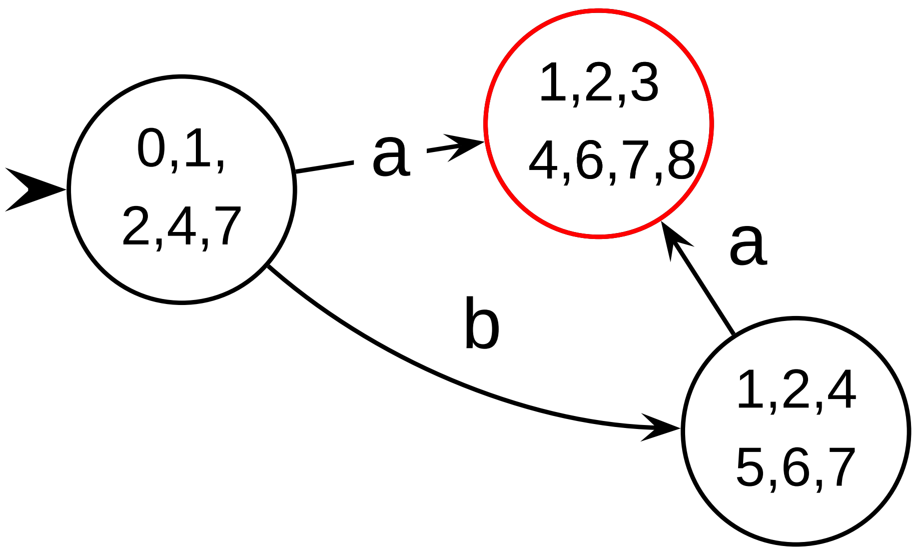
Example: Subset Construction



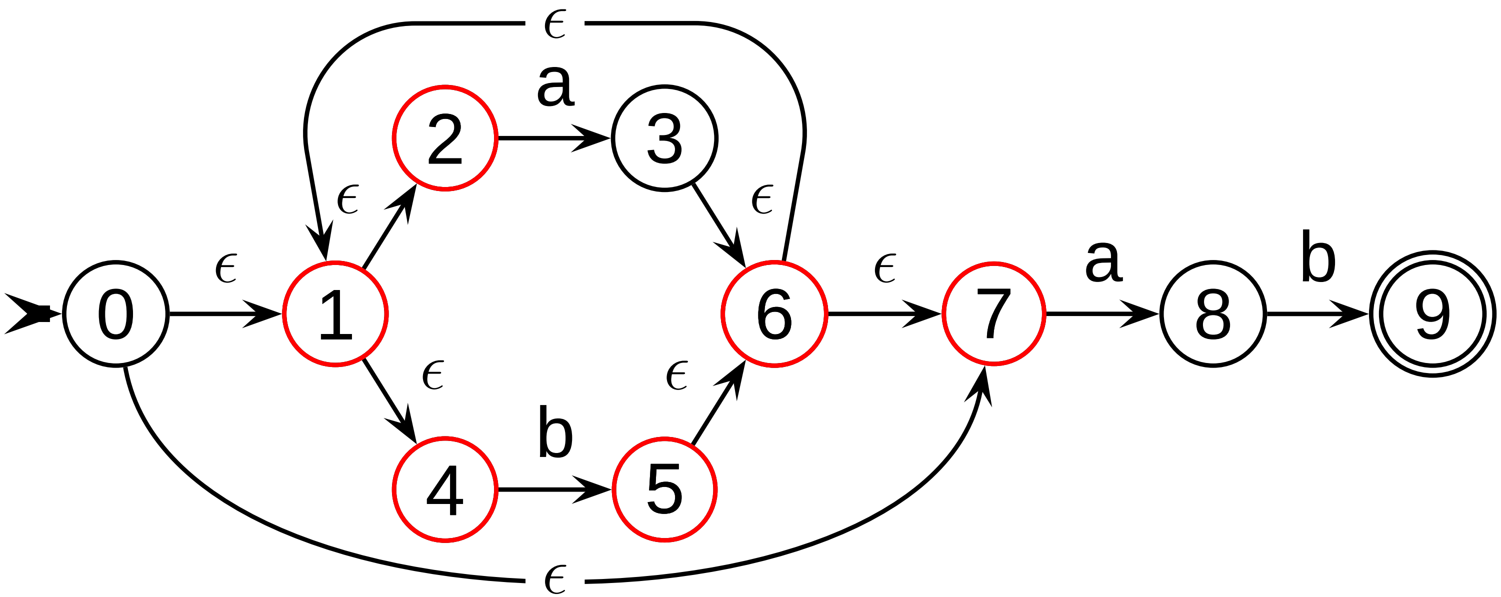


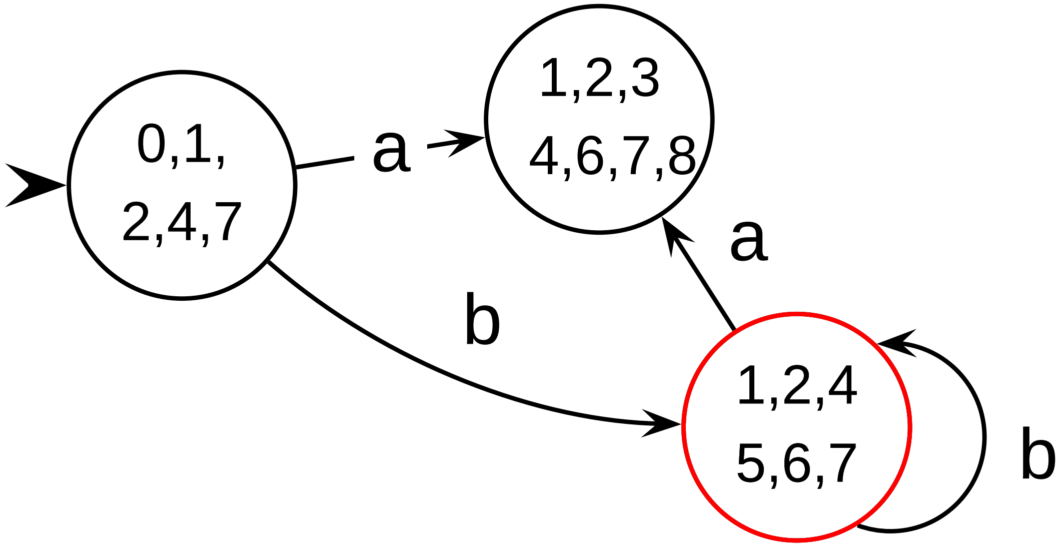
Example: Subset Construction

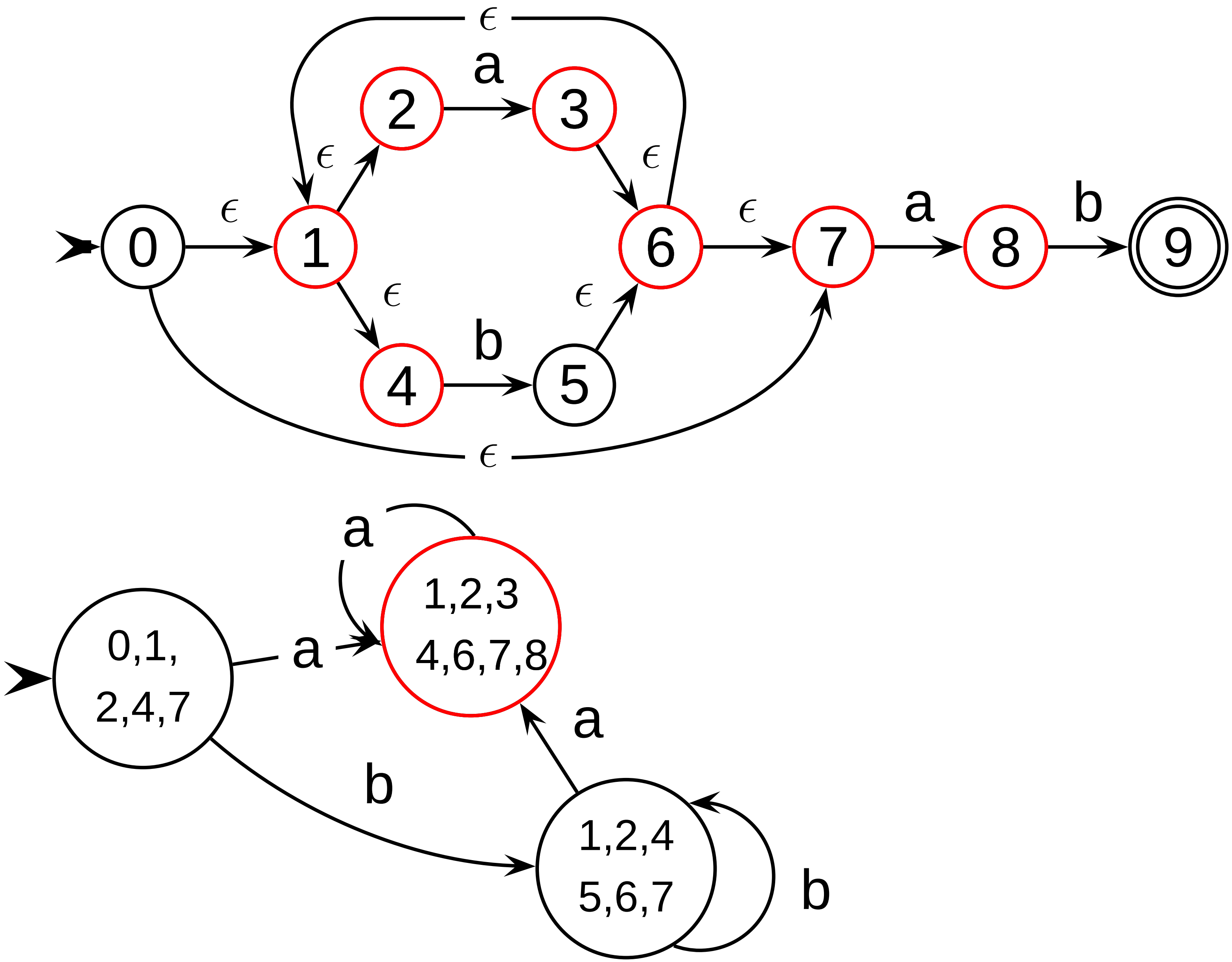




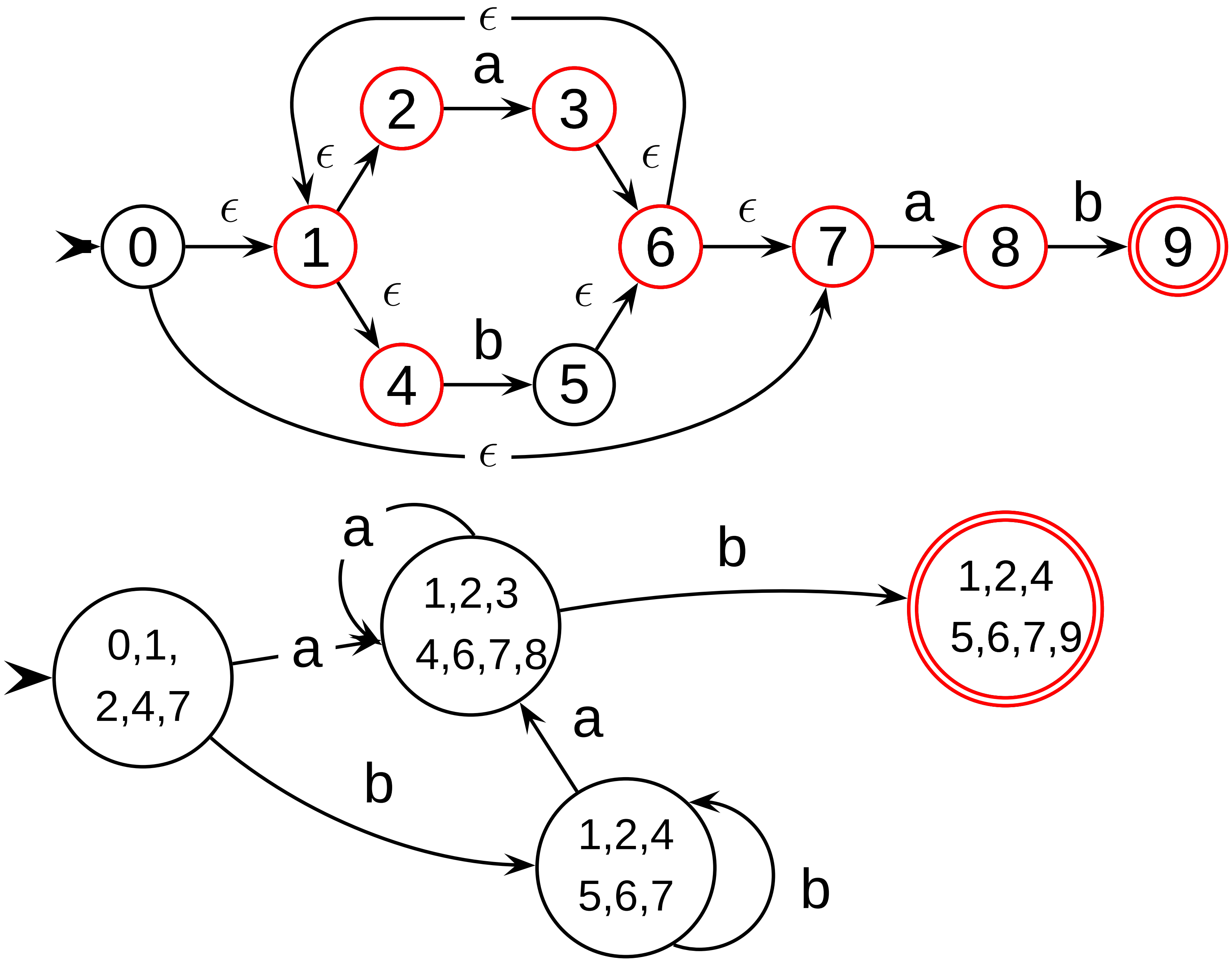
Example: Subset Construction



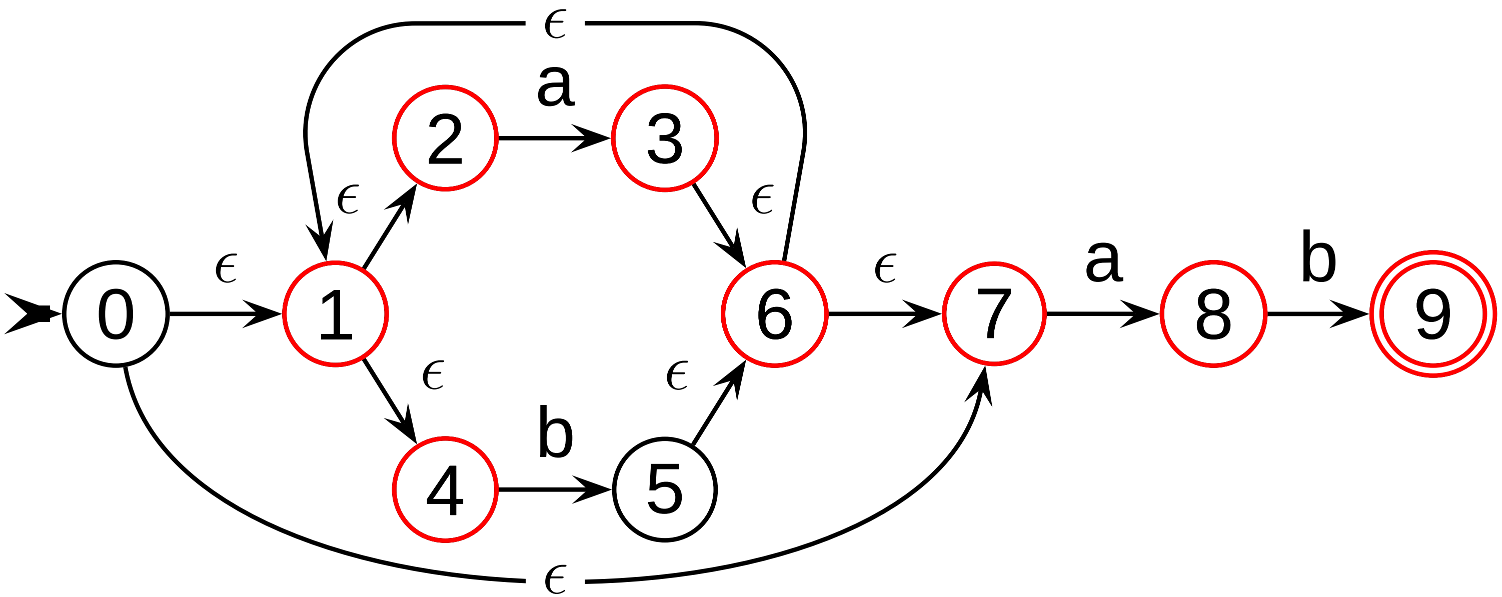


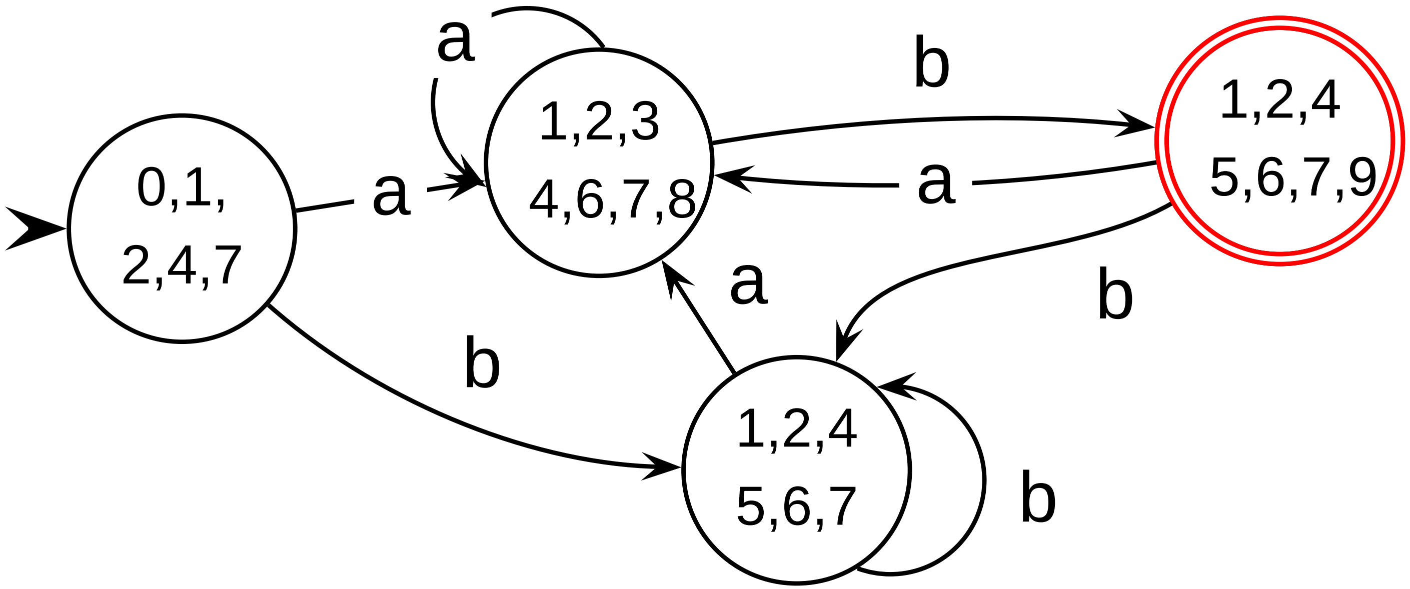

ϵ
a
2
3
ϵ
ϵ
ϵ
ϵ
0
1
6
7
a
8
b
9
ϵ
ϵ
b
4
5
ϵ
a
1,2,3
4,6,7,8
0,1,
2,4,7
a
a
b
1,2,4
5,6,7
b

Example: Subset Construction



Example: Subset Construction





Time/Space Considerations

- DFA traversal is linear to the length of input string x
- NFA needs $\mathcal{O}(n)$ space (states+transitions), where n is the length of the regular expression
- NFA traversal may need time $n \times |x|$, so why use NFAs?

Time/Space Considerations

- DFA traversal is linear to the length of input string x
- NFA needs $\mathcal{O}(n)$ space (states+transitions), where n is the length of the regular expression
- NFA traversal may need time $n \times |x|$, so why use NFAs?
- There are DFA that have at least 2^n states!

Time/Space Considerations

- DFA traversal is linear to the length of input string x
- NFA needs $\mathcal{O}(n)$ space (states+transitions), where n is the length of the regular expression
- NFA traversal may need time $n \times |x|$, so why use NFAs?
- There are DFA that have at least 2^n states!
- Solution 1: “Lazy” construction of the DFA: construct DFA states on the fly up to a certain amount and cache them

Time/Space Considerations

- DFA traversal is linear to the length of input string x
- NFA needs $\mathcal{O}(n)$ space (states+transitions), where n is the length of the regular expression
- NFA traversal may need time $n \times |x|$, so why use NFAs?
- There are DFA that have at least 2^n states!
- Solution 1: “Lazy” construction of the DFA: construct DFA states on the fly up to a certain amount and cache them
- Solution 2: Try to minimize the DFA:
  There is a unique (modulo state names) minimal automaton for a regular language!

Minimization Algorithm by Hopcroft

proc *Minimize*$() \equiv$

$B_1 = F;\ B_2 = Q\ F$

$E = \{B_1, B_2\}$

$k = 3$

for $a \in \Sigma$ **do**

$a(i) = \{s \in Q | s \in B_i \wedge \exists t : \delta(t, a) = s\}$

$L =$ the smaller of the $a(i)$

while $L \neq \emptyset$ **do**

take some $i \in L$ and delete it

for $j < k$ s.th. $\exists t \in B_j$