Chapter 6. Finite Automata

Algorithmics and Programming III

FIB

Albert Oliveras Enric Rodríguez

Q1 2019–2020

Version December 9, 2019

Motivation

- We want to search for a sequence of characters (a pattern) p in a text t

Motivation

- We want to search for a sequence of characters (a pattern) p in a text t
- A possible solution:

```
// Returns whether p occurs in t starting at position i
bool occurs_at(const string& p, int i, const string& t) {
  for (int j = 0; j < p.size(); ++j)
    if (p[j] != t[i+j]) return false;
  return true;
}

// Returns whether p occurs in t
bool occurs(const string& p, const string& t) {
  for (int i = 0; i + p.size() <= t.size(); ++i)
    if (occurs_at(p, i, t)) return true;
  return false;
}
```

- In the worst case it makes $\Theta(|p| \cdot |t|)$ comparisons of characters

Motivation

- We want to search for a sequence of characters (a pattern) p in a text t
- A possible solution:

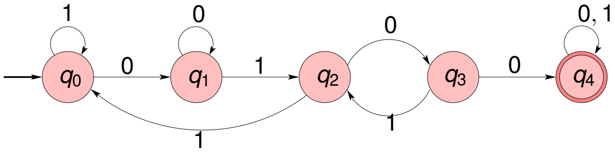
```
// Returns whether p occurs in t starting at position i
bool occurs_at(const string& p, int i, const string& t) {
  for (int j = 0; j < p.size(); ++j)
    if (p[j] != t[i+j]) return false;
  return true;
}

// Returns whether p occurs in t
bool occurs(const string& p, const string& t) {
  for (int i = 0; i + p.size() <= t.size(); ++i)
    if (occurs_at(p, i, t)) return true;
  return false;
}
```

- In the worst case it makes $\Theta(|p| \cdot |t|)$ comparisons of characters
- But it is rather naive: it does not use info of previous attempts

Motivation

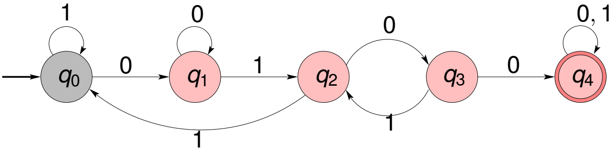
- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- This automaton accepts exactly the texts that contain p

Motivation

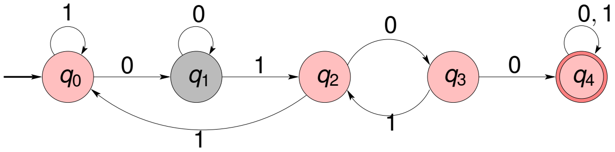
- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- It has 5 states

Motivation

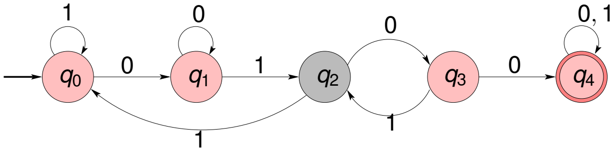
- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- It has 5 states

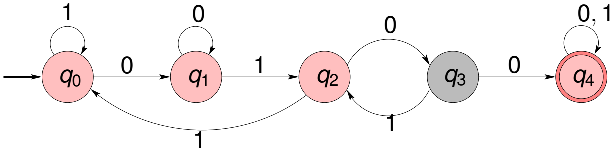
Motivation

- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- It has 5 states

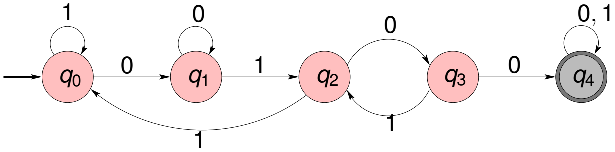
- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- It has 5 states

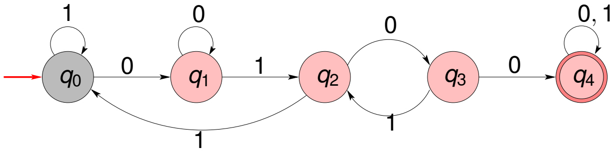
Motivation

- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



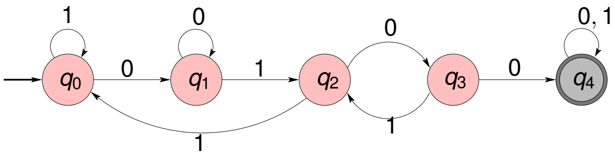
- It has 5 states

- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- It has one initial state (indicated by the arrow without source node)

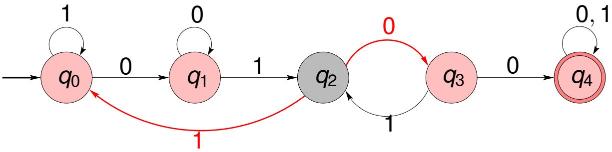
- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- It has one accepting state (indicated by the double circle)

Motivation

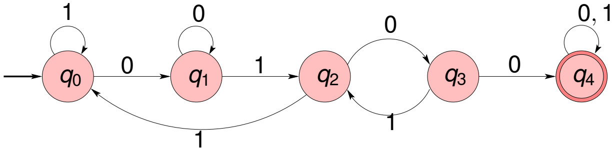
- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- Each state has two transitions, one for each symbol

Motivation

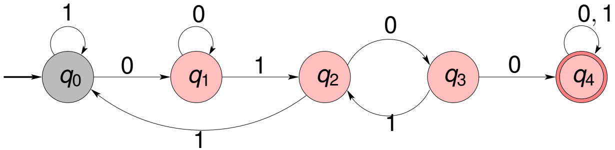
- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- The automaton starts at q_0 and reads text from left to right changing state according to the symbol that has been read

Motivation

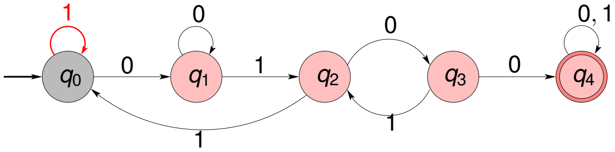
- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- The automaton starts at q_0 and reads text from left to right changing state according to the symbol that has been read
- For instance let us run the automaton on the text 101010010

Motivation

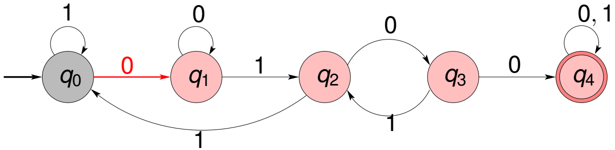
- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- The automaton starts at q_0 and reads text from left to right changing state according to the symbol that has been read
- For instance let us run the automaton on the text 101010010

Motivation

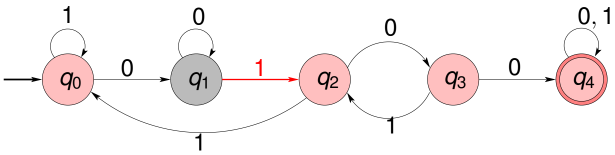
- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- The automaton starts at q_0 and reads text from left to right changing state according to the symbol that has been read
- For instance let us run the automaton on the text 101010010

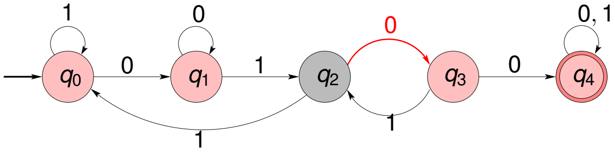
Motivation

- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- The automaton starts at q_0 and reads text from left to right changing state according to the symbol that has been read
- For instance let us run the automaton on the text 101010010

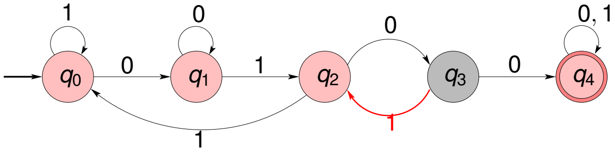
- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- The automaton starts at q_0 and reads text from left to right changing state according to the symbol that has been read
- For instance let us run the automaton on the text 101010010

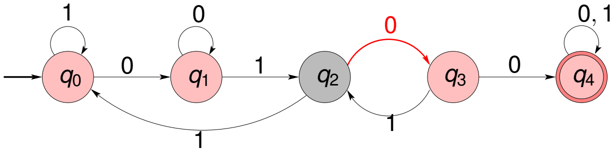
Motivation

- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- The automaton starts at q_0 and reads text from left to right changing state according to the symbol that has been read
- For instance let us run the automaton on the text 101010010

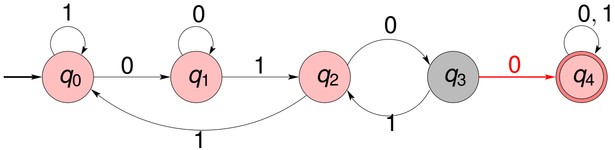
- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- The automaton starts at q_0 and reads text from left to right changing state according to the symbol that has been read
- For instance let us run the automaton on the text 101010010

Motivation

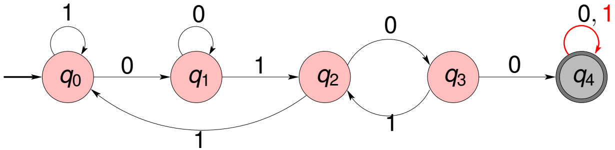
- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- The automaton starts at q_0 and reads text from left to right changing state according to the symbol that has been read
- For instance let us run the automaton on the text 101010010

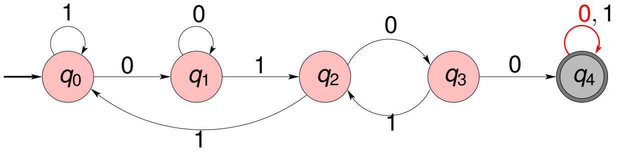
Motivation

- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- The automaton starts at q_0 and reads text from left to right changing state according to the symbol that has been read
- For instance let us run the automaton on the text 101010010

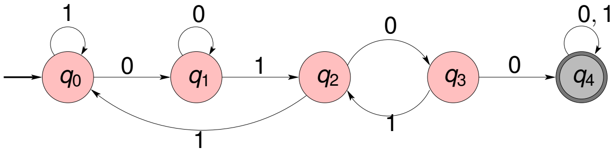
- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- The automaton starts at q_0 and reads text from left to right changing state according to the symbol that has been read
- For instance let us run the automaton on the text 101010010

Motivation

- Let us assume the text only contains binary digits
- For searching e.g. $p = 0100$ we can use the following finite automaton:



- The automaton starts at q_0 and reads text from left to right changing state according to the symbol that has been read
- For instance let us run the automaton on the text 101010010
- Since in the end we are at an accepting state, the text is accepted

Motivation

- It can be proved that for any pattern p
  one can build a finite automaton recognizing p in time $\Theta(|p|)$
- For processing a text t this automaton takes exactly $|t|$ steps
- Algorithm for pattern search based on finite automata costs $\Theta(|p| + |t|)$
- Compare with the worst-case cost $\Theta(|p| \cdot |t|)$ of the naive algorithm!

Alphabets

- An alphabet is a finite non-empty set.

  The elements of an alphabet are called symbols.

- An alphabet is a finite non-empty set.

  The elements of an alphabet are called symbols.

- For example:
  - The Latin alphabet $\{\mathtt{A}, \mathtt{B}, \mathtt{C}, \dots, \mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$
  - The decimal alphabet $\{\mathtt{0}, \mathtt{1}, \mathtt{2}, \dots, \mathtt{7}, \mathtt{8}, \mathtt{9}\}$
  - The binary alphabet $\{\mathtt{0}, \mathtt{1}\}$
  - The ASCII alphabet
  - The DNA alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$

Alphabets

- An alphabet is a finite non-empty set.

  The elements of an alphabet are called symbols.

- For example:
  - The Latin alphabet $\{\mathtt{A}, \mathtt{B}, \mathtt{C}, \ldots, \mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$
  - The decimal alphabet $\{\mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots, \mathtt{7}, \mathtt{8}, \mathtt{9}\}$
  - The binary alphabet $\{\mathtt{0}, \mathtt{1}\}$
  - The ASCII alphabet
  - The DNA alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$

- An alphabet will be usually represented with the letter Σ

Words

- Given alphabet Σ, a word or string is a finite sequence of symbols of Σ
- For example:
  - $\texttt{FIB}$ is a word over the Latin alphabet $\{\texttt{A}, \texttt{B}, \texttt{C}, \ldots, \texttt{X}, \texttt{Y}, \texttt{Z}\}$
  - $\texttt{2019}$ is a word over the decimal alphabet $\{\texttt{0}, \texttt{1}, \texttt{2}, \ldots, \texttt{7}, \texttt{8}, \texttt{9}\}$
  - $\texttt{010100}$ is a word over the binary alphabet $\{\texttt{0}, \texttt{1}\}$

Words

- Given alphabet Σ, a word or string is a finite sequence of symbols of Σ
- For example:
  - $\texttt{FIB}$ is a word over the Latin alphabet $\{\texttt{A}, \texttt{B}, \texttt{C}, \ldots, \texttt{X}, \texttt{Y}, \texttt{Z}\}$
  - $\texttt{2019}$ is a word over the decimal alphabet $\{\texttt{0}, \texttt{1}, \texttt{2}, \ldots, \texttt{7}, \texttt{8}, \texttt{9}\}$
  - $\texttt{010100}$ is a word over the binary alphabet $\{\texttt{0}, \texttt{1}\}$
- The empty word, denoted λ, corresponds to an empty sequence

Words

- Given alphabet Σ, a word or string is a finite sequence of symbols of Σ
- For example:
  - $\mathtt{FIB}$ is a word over the Latin alphabet $\{\mathtt{A}, \mathtt{B}, \mathtt{C}, \ldots, \mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$
  - $\mathtt{2019}$ is a word over the decimal alphabet $\{\mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots, \mathtt{7}, \mathtt{8}, \mathtt{9}\}$
  - $\mathtt{010100}$ is a word over the binary alphabet $\{\mathtt{0}, \mathtt{1}\}$
- The empty word, denoted λ, corresponds to an empty sequence
- The set of all words over an alphabet Σ is denoted by Σ^*

Languages

- A language $\mathcal{L}$ is any set of words over an alphabet Σ, i.e., $\mathcal{L} \subseteq \Sigma^*$

Languages

- A language $\mathcal{L}$ is any set of words over an alphabet Σ, i.e., $\mathcal{L} \subseteq \Sigma^*$
- For example, over the alphabet $\Sigma = \{0, 1\}$, we can consider the language $\mathcal{L}$ of all words that contain an occurrence of 0100:

$$\mathcal{L} = \{0100, 00100, 10100, 01000, 01001, \ldots\}$$

Languages

- A language $\mathcal{L}$ is any set of words over an alphabet Σ, i.e., $\mathcal{L} \subseteq \Sigma^*$
- For example, over the alphabet $\Sigma = \{0, 1\}$, we can consider the language $\mathcal{L}$ of all words that contain an occurrence of 0100:

$$\mathcal{L} = \{0100, 00100, 10100, 01000, 01001, \ldots\}$$

- $\mathcal{L} = \emptyset$ and $\mathcal{L} = \{\lambda\}$ are other examples of languages

Deterministic Finite Automata

- A (deterministic) finite automaton (DFA) consists of:

- A (deterministic) finite automaton (DFA) consists of:
  - Q, a finite set of states

Deterministic Finite Automata

- A (deterministic) finite automaton (DFA) consists of:
  - Q, a finite set of states
  - Σ, an alphabet (called input alphabet)

Deterministic Finite Automata

- A (deterministic) finite automaton (DFA) consists of:
  - Q, a finite set of states
  - Σ, an alphabet (called input alphabet)
  - δ, a transition function from $Q \times \Sigma$ to Q

Deterministic Finite Automata

- A (deterministic) finite automaton (DFA) consists of:
  - Q, a finite set of states
  - Σ, an alphabet (called input alphabet)
  - δ, a transition function from $Q \times \Sigma$ to Q
  - $q_0 \in Q$, called the initial or start state

Deterministic Finite Automata

- A (deterministic) finite automaton (DFA) consists of:
  - Q, a finite set of states
  - Σ, an alphabet (called input alphabet)
  - δ, a transition function from $Q \times \Sigma$ to Q
  - $q_0 \in Q$, called the initial or start state
  - $F \subseteq Q$, called the set of final or accepting states

Deterministic Finite Automata

- $(Q, \Sigma, \delta, q_0, F)$ is an example of DFA, where:
  - $Q = \{q_0, q_1, q_2, q_3, q_4\}$
  - $\Sigma = \{0, 1\}$
  - δ is described by the following transition table:

	q_0	q_1	q_2	q_3	q_4
0	q_1	q_1	q_3	q_4	q_4
1	q_0	q_2	q_0	q_2	q_4

  - the initial state is q_0
  - $F = \{q_4\}$

Deterministic Finite Automata

- $(Q, \Sigma, \delta, q_0, F)$ is an example of DFA, where:
  - $Q = \{q_0, q_1, q_2, q_3, q_4\}$
  - $\Sigma = \{0, 1\}$
  - δ is described by the following transition table:

	q_0	q_1	q_2	q_3	q_4
0	q_1	q_1	q_3	q_4	q_4
1	q_0	q_2	q_0	q_2	q_4

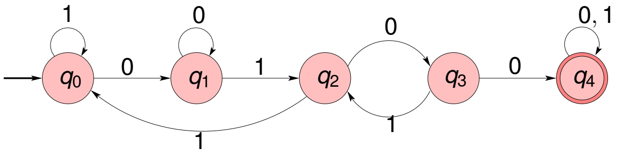
  - the initial state is q_0
  - $F = \{q_4\}$
- Often the initial and accepting states are marked in the transition table

Deterministic Finite Automata

- $(Q, \Sigma, \delta, q_0, F)$ is an example of DFA, where:
  - $Q = \{q_0, q_1, q_2, q_3, q_4\}$
  - $\Sigma = \{0, 1\}$
  - δ is described by the following transition table:

	q_0	q_1	q_2	q_3	q_4
0	q_1	q_1	q_3	q_4	q_4
1	q_0	q_2	q_0	q_2	q_4

  - the initial state is q_0
  - $F = \{q_4\}$
- Often the initial and accepting states are marked in the transition table
- An alternative representation of the automaton with a transition diagram:



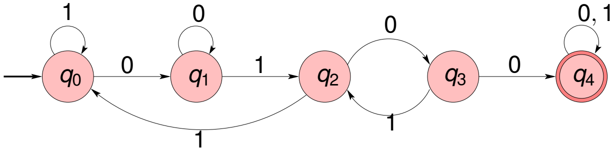
Deterministic Finite Automata

- When needed we will extend transition functions from symbols to words
- If q is a state and a is a symbol,
  then $\delta(q, a)$ is the state we reach from q after reading symbol a
- Similarly, if q is a state and ω is a word,
  then $\delta(q, \omega)$ will be the state we reach from q after reading word ω

Deterministic Finite Automata

- When needed we will extend transition functions from symbols to words
- If q is a state and a is a symbol, then $\delta(q, a)$ is the state we reach from q after reading symbol a
- Similarly, if q is a state and ω is a word, then $\delta(q, \omega)$ will be the state we reach from q after reading word ω
- Given transition function $\delta : Q \times \Sigma \to Q$, we extend it to $Q \times \Sigma^* \to Q$ recursively:
  - for any $q \in Q$, $\delta(q, \lambda) = q$
  - for any $q \in Q$ and word of the form $a\,\omega$, $\delta(q, a\,\omega) = \delta(\delta(q, a), \omega)$

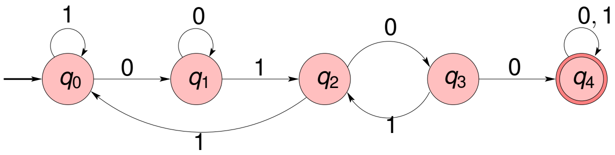
- When needed we will extend transition functions from symbols to words
- If q is a state and a is a symbol,
  then $\delta(q, a)$ is the state we reach from q after reading symbol a
- Similarly, if q is a state and ω is a word,
  then $\delta(q, \omega)$ will be the state we reach from q after reading word ω
- Given transition function $\delta : Q \times \Sigma \to Q$,
  we extend it to $Q \times \Sigma^* \to Q$ recursively:
  - for any $q \in Q$, $\delta(q, \lambda) = q$
  - for any $q \in Q$ and word of the form $a\,\omega$, $\delta(q, a\,\omega) = \delta(\delta(q, a), \omega)$
- For example, with the DFA



we have $\delta(q_0, 0100) =$

Deterministic Finite Automata

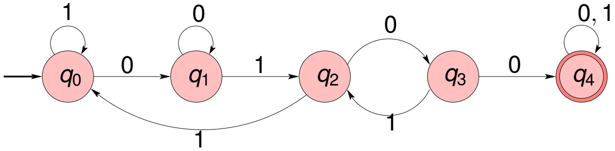
- When needed we will extend transition functions from symbols to words
- If q is a state and a is a symbol,
  then $\delta(q, a)$ is the state we reach from q after reading symbol a
- Similarly, if q is a state and ω is a word,
  then $\delta(q, \omega)$ will be the state we reach from q after reading word ω
- Given transition function $\delta : Q \times \Sigma \to Q$,
  we extend it to $Q \times \Sigma^* \to Q$ recursively:
  - for any $q \in Q$, $\delta(q, \lambda) = q$
  - for any $q \in Q$ and word of the form $a\,\omega$, $\delta(q, a\,\omega) = \delta(\delta(q, a), \omega)$
- For example, with the DFA



we have $\delta(q_0, 0100) = \delta(q_1, 100) =$

Deterministic Finite Automata

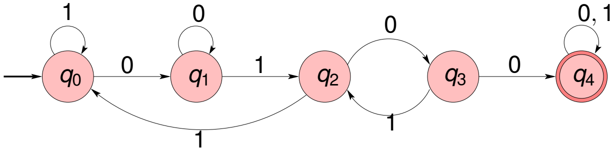
- When needed we will extend transition functions from symbols to words
- If q is a state and a is a symbol,
  then $\delta(q, a)$ is the state we reach from q after reading symbol a
- Similarly, if q is a state and ω is a word,
  then $\delta(q, \omega)$ will be the state we reach from q after reading word ω
- Given transition function $\delta : Q \times \Sigma \to Q$,
  we extend it to $Q \times \Sigma^* \to Q$ recursively:
  - for any $q \in Q$, $\delta(q, \lambda) = q$
  - for any $q \in Q$ and word of the form $a\,\omega$, $\delta(q, a\,\omega) = \delta(\delta(q, a), \omega)$
- For example, with the DFA



we have $\delta(q_0, 0100) = \delta(q_1, 100) = \delta(q_2, 00) =$

Deterministic Finite Automata

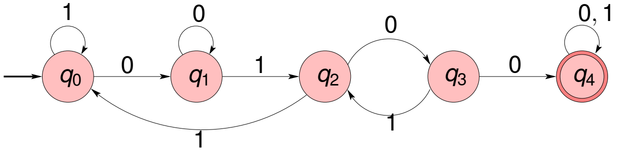
- When needed we will extend transition functions from symbols to words
- If q is a state and a is a symbol,
  then $\delta(q, a)$ is the state we reach from q after reading symbol a
- Similarly, if q is a state and ω is a word,
  then $\delta(q, \omega)$ will be the state we reach from q after reading word ω
- Given transition function $\delta : Q \times \Sigma \to Q$,
  we extend it to $Q \times \Sigma^* \to Q$ recursively:
  - for any $q \in Q$, $\delta(q, \lambda) = q$
  - for any $q \in Q$ and word of the form $a\,\omega$, $\delta(q, a\,\omega) = \delta(\delta(q, a), \omega)$
- For example, with the DFA



we have $\delta(q_0, 0100) = \delta(q_1, 100) = \delta(q_2, 00) = \delta(q_3, 0) =$

Deterministic Finite Automata

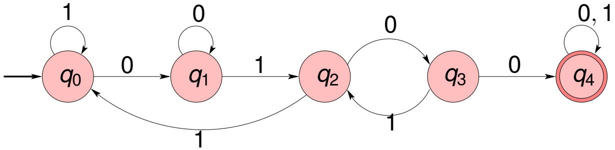
- When needed we will extend transition functions from symbols to words
- If q is a state and a is a symbol,
  then $\delta(q, a)$ is the state we reach from q after reading symbol a
- Similarly, if q is a state and ω is a word,
  then $\delta(q, \omega)$ will be the state we reach from q after reading word ω
- Given transition function $\delta : Q \times \Sigma \to Q$,
  we extend it to $Q \times \Sigma^* \to Q$ recursively:
  - for any $q \in Q$, $\delta(q, \lambda) = q$
  - for any $q \in Q$ and word of the form $a\,\omega$, $\delta(q, a\,\omega) = \delta(\delta(q, a), \omega)$
- For example, with the DFA



we have $\delta(q_0, 0100) = \delta(q_1, 100) = \delta(q_2, 00) = \delta(q_3, 0) = \delta(q_4, \lambda) =$

Deterministic Finite Automata

- When needed we will extend transition functions from symbols to words
- If q is a state and a is a symbol,
  then $\delta(q, a)$ is the state we reach from q after reading symbol a
- Similarly, if q is a state and ω is a word,
  then $\delta(q, \omega)$ will be the state we reach from q after reading word ω
- Given transition function $\delta : Q \times \Sigma \to Q$,
  we extend it to $Q \times \Sigma^* \to Q$ recursively:
  - for any $q \in Q$, $\delta(q, \lambda) = q$
  - for any $q \in Q$ and word of the form $a\,\omega$, $\delta(q, a\,\omega) = \delta(\delta(q, a), \omega)$
- For example, with the DFA



we have $\delta(q_0, 0100) = \delta(q_1, 100) = \delta(q_2, 00) = \delta(q_3, 0) = \delta(q_4, \lambda) = q_4$

- Given a DFA $A = (Q, \Sigma, \delta, q_0, F)$,
  a word $\omega \in \Sigma^*$ is accepted by A if
  following the transition function from the initial state
  we reach an accepting state:

$$\delta(q_0, \omega) \in F$$

Regular Languages

- Given a DFA $A = (Q, \Sigma, \delta, q_0, F)$,
  a word $\omega \in \Sigma^*$ is accepted by A if
  following the transition function from the initial state
  we reach an accepting state:

$$\delta(q_0, \omega) \in F$$

- The language of A, denoted $L(A)$, is the set of all words accepted by A

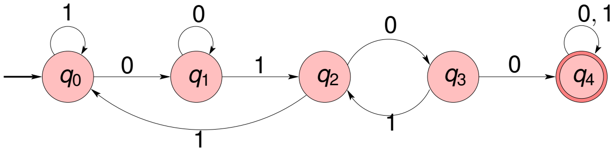
Regular Languages

- Given a DFA $A = (Q, \Sigma, \delta, q_0, F)$,
  a word $\omega \in \Sigma^*$ is accepted by A if
  following the transition function from the initial state
  we reach an accepting state:

$$\delta(q_0, \omega) \in F$$

- The language of A, denoted $L(A)$, is the set of all words accepted by A

- For example, the language of



  consists of all words that contain an occurrence of 0100

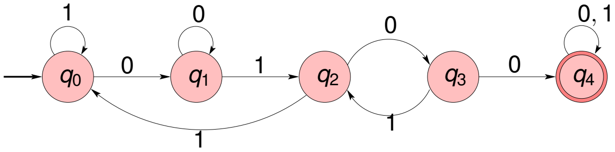
Regular Languages

- Given a DFA $A = (Q, \Sigma, \delta, q_0, F)$,
  a word $\omega \in \Sigma^*$ is accepted by A if
  following the transition function from the initial state
  we reach an accepting state:

$$\delta(q_0, \omega) \in F$$

- The language of A, denoted $L(A)$, is the set of all words accepted by A

- For example, the language of



  consists of all words that contain an occurrence of 0100

- A language $\mathcal{L}$ is called regular if it is the language of some DFA

- So the language of all words containing an occurrence of 0100 is regular

Nondeterministic Finite Automata

- DFA's are deterministic: automata can only be in one state at any time
  - there is a single initial state
  - at each state, there is exactly one transition that can be taken

Nondeterministic Finite Automata

- DFA's are deterministic: automata can only be in one state at any time
  - there is a single initial state
  - at each state, there is exactly one transition that can be taken
- In nondeterministic finite automata (NFA) this is no longer true

Nondeterministic Finite Automata

- DFA's are deterministic: automata can only be in one state at any time
  - there is a single initial state
  - at each state, there is exactly one transition that can be taken
- In nondeterministic finite automata (NFA) this is no longer true
- NFA's have the same expressive power as DFA's:
  a language is accepted by some DFA iff it is accepted by some NFA
- But NFA's are usually more compact and easier to design

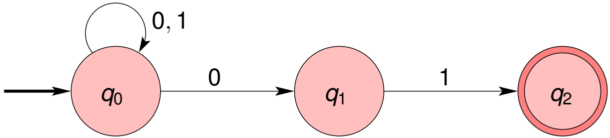
Nondeterministic Finite Automata

- The difference between DFA's and NFA's is in the transition function δ

  In NFA's:

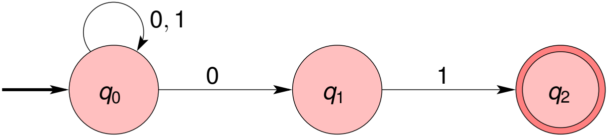
  - δ is a function that takes a state and an input symbol as arguments (as in DFA's)
  - δ returns a set of zero, one, or more states
    (rather than returning exactly one state, as DFA's do)

- For example, this NFA accepts exactly the words that end in 01:

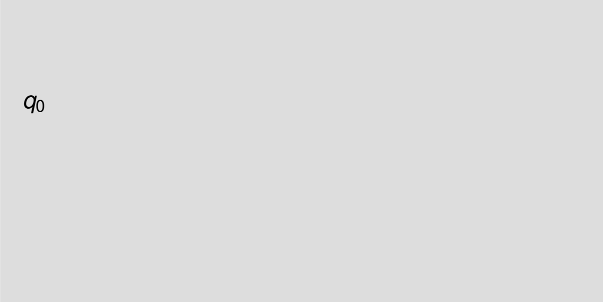


Nondeterministic Finite Automata

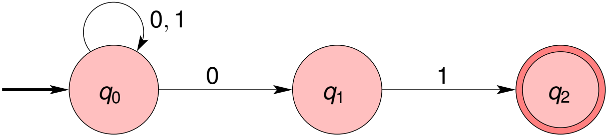
- For example, this NFA accepts exactly the words that end in 01:



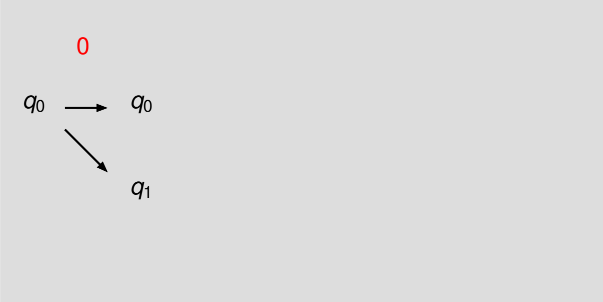
- Instead of having a single execution thread, an NFA has a tree of threads



- For example, this NFA accepts exactly the words that end in 01:

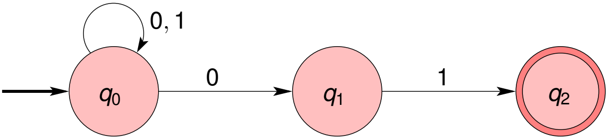


- Instead of having a single execution thread, an NFA has a tree of threads

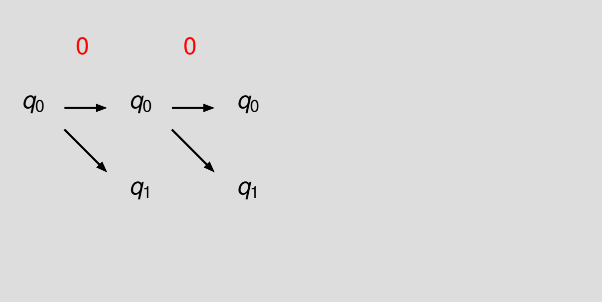


Nondeterministic Finite Automata

- For example, this NFA accepts exactly the words that end in 01:

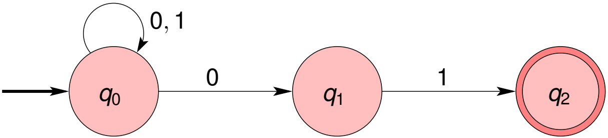


- Instead of having a single execution thread, an NFA has a tree of threads

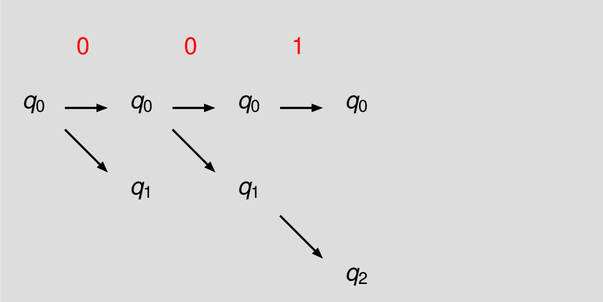


Nondeterministic Finite Automata

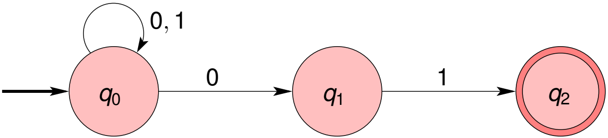
- For example, this NFA accepts exactly the words that end in 01:



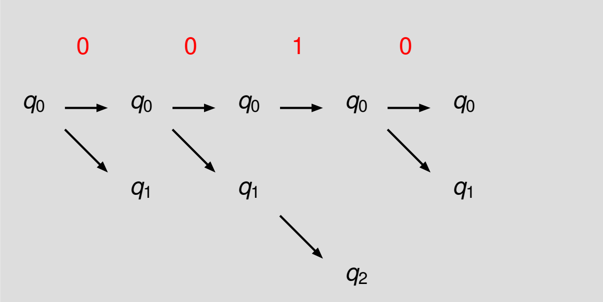
- Instead of having a single execution thread, an NFA has a tree of threads



- For example, this NFA accepts exactly the words that end in 01:

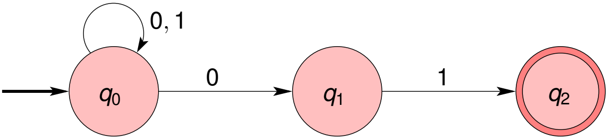


- Instead of having a single execution thread, an NFA has a tree of threads

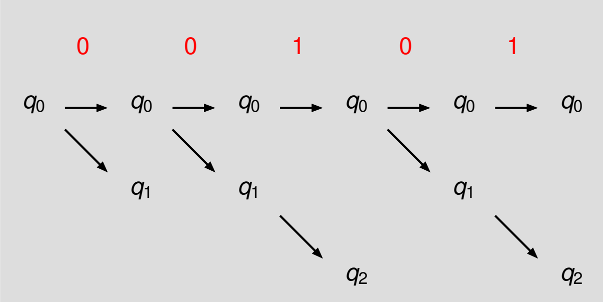


Nondeterministic Finite Automata

- For example, this NFA accepts exactly the words that end in 01:

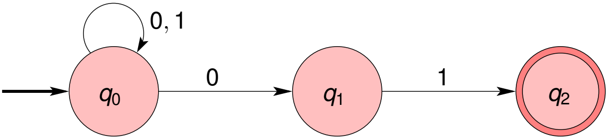


- Instead of having a single execution thread, an NFA has a tree of threads

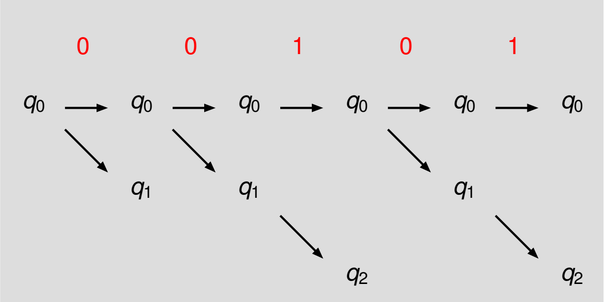


Nondeterministic Finite Automata

- For example, this NFA accepts exactly the words that end in 01:



- Instead of having a single execution thread, an NFA has a tree of threads



- Threads are run simultaneously, so the NFA can be in several states

Nondeterministic Finite Automata

- A nondeterministic finite automaton (NFA) consists of:

Nondeterministic Finite Automata

- A nondeterministic finite automaton (NFA) consists of:
  - Q, a finite set of states

Nondeterministic Finite Automata

- A nondeterministic finite automaton (NFA) consists of:
  - Q, a finite set of states
  - Σ, an alphabet (called input alphabet)

Nondeterministic Finite Automata

- A nondeterministic finite automaton (NFA) consists of:
  - Q, a finite set of states
  - Σ, an alphabet (called input alphabet)
  - δ, a transition function from $Q \times \Sigma$ to $2^Q = \{\text{subsets of } Q\}$

Nondeterministic Finite Automata

- A nondeterministic finite automaton (NFA) consists of:
  - Q, a finite set of states
  - Σ, an alphabet (called input alphabet)
  - δ, a transition function from $Q \times \Sigma$ to $2^Q = \{\text{subsets of } Q\}$
  - $q_0 \in Q$, called the initial state

Nondeterministic Finite Automata

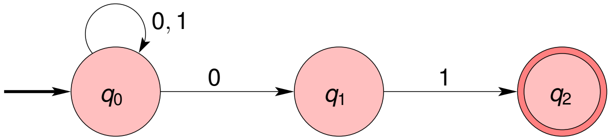
- A nondeterministic finite automaton (NFA) consists of:
  - Q, a finite set of states
  - Σ, an alphabet (called input alphabet)
  - δ, a transition function from $Q \times \Sigma$ to $2^Q = \{\text{subsets of } Q\}$
  - $q_0 \in Q$, called the initial state
  - $F \subseteq Q$, called the set of final or accepting states

Nondeterministic Finite Automata

- $(Q, \Sigma, \delta, q_0, F)$ is an example of NFA, where:
  - $Q = \{q_0, q_1, q_2\}$
  - $\Sigma = \{0, 1\}$
  - the initial state is q_0
  - $F = \{q_2\}$
  - δ is described by the following transition table:

	q_0	q_1	q_2
0	$\{q_0, q_1\}$	$\emptyset$	$\emptyset$
1	$\{q_0\}$	$\{q_2\}$	$\emptyset$

- The representation of the same NFA with a transition diagram:



Nondeterministic Finite Automata

- When there is no transition from a state q on a symbol a, then $\delta(q, a) = \emptyset$

Nondeterministic Finite Automata

- When there is no transition from a state q on a symbol a, then $\delta(q, a) = \emptyset$
- A DFA is also an NFA:

  for each state and symbol, δ returns a set consisting of a single state

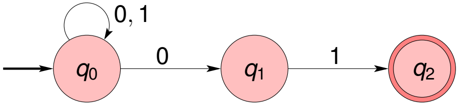
- Again we will extend transition functions from symbols to words
- If q is a state and a is a symbol,
  then $\delta(q, a)$ are the states we can reach from q after reading symbol a
- If q is a state and ω is a word,
  then $\delta(q, \omega)$ will be the states we can reach from q after reading word ω

- Again we will extend transition functions from symbols to words
- If q is a state and a is a symbol,
  then $\delta(q, a)$ are the states we can reach from q after reading symbol a
- If q is a state and ω is a word,
  then $\delta(q, \omega)$ will be the states we can reach from q after reading word ω
- Given transition function $\delta : Q \times \Sigma \rightarrow 2^Q$, we extend it to $Q \times \Sigma^* \rightarrow 2^Q$:
  - for any $q \in Q$, $\delta(q, \lambda) = \{q\}$
  - for any $q \in Q$ and word of the form $a\,\omega$,

$$\delta(q, a\,\omega) = \bigcup_{p \in \delta(q,a)} \delta(p, \omega)$$

Nondeterministic Finite Automata

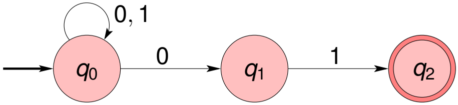
For example, with the NFA



$\delta(q_0, 00101)$

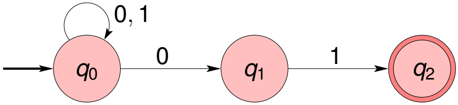
Nondeterministic Finite Automata

For example, with the NFA



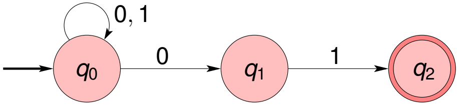
$$\begin{aligned} &\delta(q_0, 00101) \\ =\ &\delta(q_0, 0101) \cup \delta(q_1, 0101) \end{aligned}$$

For example, with the NFA



$$
\begin{aligned}
&\delta(q_0, 00101)\\
&= \delta(q_0, 0101) \cup \delta(q_1, 0101)\\
&= \delta(q_0, 101) \cup \delta(q_1, 101)
\end{aligned}
$$

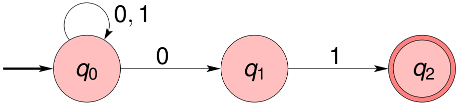
For example, with the NFA



$$
\begin{aligned}
& \delta(q_0, 00101) \\
= {} & \delta(q_0, 0101) \cup \delta(q_1, 0101) \\
= {} & \delta(q_0, 101) \cup \delta(q_1, 101) \\
= {} & \delta(q_0, 01) \cup \delta(q_2, 01)
\end{aligned}
$$

Nondeterministic Finite Automata

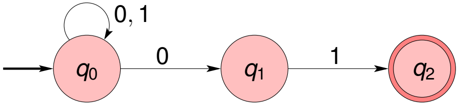
For example, with the NFA



$$
\begin{aligned}
& \delta(q_0, 00101) \\
= {} & \delta(q_0, 0101) \cup \delta(q_1, 0101) \\
= {} & \delta(q_0, 101) \cup \delta(q_1, 101) \\
= {} & \delta(q_0, 01) \cup \delta(q_2, 01) \\
= {} & \delta(q_0, 1) \cup \delta(q_1, 1)
\end{aligned}
$$

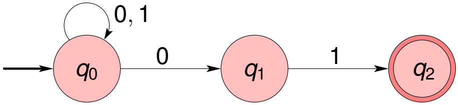
Nondeterministic Finite Automata

For example, with the NFA



$$
\begin{aligned}
&\delta(q_0, 00101) \\
&= \delta(q_0, 0101) \cup \delta(q_1, 0101) \\
&= \delta(q_0, 101) \cup \delta(q_1, 101) \\
&= \delta(q_0, 01) \cup \delta(q_2, 01) \\
&= \delta(q_0, 1) \cup \delta(q_1, 1) \\
&= \delta(q_0, \lambda) \cup \delta(q_2, \lambda)
\end{aligned}
$$

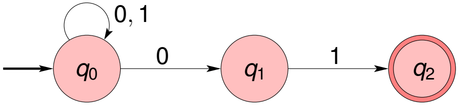
For example, with the NFA



$$
\begin{aligned}
& \delta(q_0, 00101) \\
= {} & \delta(q_0, 0101) \cup \delta(q_1, 0101) \\
= {} & \delta(q_0, 101) \cup \delta(q_1, 101) \\
= {} & \delta(q_0, 01) \cup \delta(q_2, 01) \\
= {} & \delta(q_0, 1) \cup \delta(q_1, 1) \\
= {} & \delta(q_0, \lambda) \cup \delta(q_2, \lambda) \\
= {} & \{q_0, q_2\}
\end{aligned}
$$

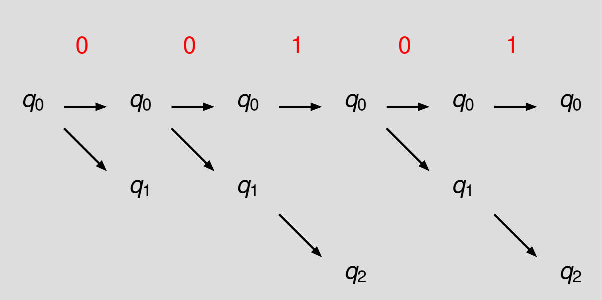
Nondeterministic Finite Automata

For example, with the NFA



$$\begin{aligned}
&\delta(q_0, 00101) \\
&= \delta(q_0, 0101) \cup \delta(q_1, 0101) \\
&= \delta(q_0, 101) \cup \delta(q_1, 101) \\
&= \delta(q_0, 01) \cup \delta(q_2, 01) \\
&= \delta(q_0, 1) \cup \delta(q_1, 1) \\
&= \delta(q_0, \lambda) \cup \delta(q_2, \lambda) \\
&= \{q_0, q_2\}
\end{aligned}$$

$\delta(q, \omega)$ is the column of states after reading ω if 1st column consists of q only



Nondeterministic Finite Automata

- Intuitively, an NFA accepts a word ω if we can choose the next states while reading ω so that we go from the start state to an accepting state
- If other choices lead to a nonaccepting state, or do not lead to any state (i.e., the sequence of states dies), ω is still accepted by the NFA

Nondeterministic Finite Automata

- Intuitively, an NFA accepts a word ω if we can choose the next states while reading ω so that we go from the start state to an accepting state
- If other choices lead to a nonaccepting state, or do not lead to any state (i.e., the sequence of states dies), ω is still accepted by the NFA
- Given an NFA $A = (Q, \Sigma, \delta, q_0, F)$,
  a word $\omega \in \Sigma^*$ is accepted by A if
  following the transition function from the initial state
  we can reach an accepting state:

$$\delta(q_0, \omega) \cap F \neq \emptyset$$

I.e., ω is accepted if $\delta(q_0, \omega)$ contains at least one accepting state

Nondeterministic Finite Automata

- Intuitively, an NFA accepts a word ω if we can choose the next states while reading ω so that we go from the start state to an accepting state
- If other choices lead to a nonaccepting state, or do not lead to any state (i.e., the sequence of states dies), ω is still accepted by the NFA
- Given an NFA $A = (Q, \Sigma, \delta, q_0, F)$,
  a word $\omega \in \Sigma^*$ is accepted by A if
  following the transition function from the initial state
  we can reach an accepting state:

  $$\delta(q_0, \omega) \cap F \neq \emptyset$$

  I.e., ω is accepted if $\delta(q_0, \omega)$ contains at least one accepting state
- The language of A, denoted $L(A)$, is the set of all words accepted by A

Subset Construction

- Given an NFA N, the subset construction allows constructing a DFA D that accepts the same language as N
- Let $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ be an NFA
- Let $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ be a DFA where:

Subset Construction

- Given an NFA N, the subset construction allows constructing a DFA D that accepts the same language as N
- Let $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ be an NFA
- Let $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ be a DFA where:
  - The input alphabets are the same

Subset Construction

- Given an NFA N, the subset construction allows constructing a DFA D that accepts the same language as N
- Let $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ be an NFA
- Let $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ be a DFA where:
  - The input alphabets are the same
  - $Q_D = 2^{Q_N}$, i.e., states of Q_D are sets of states (subsets) of Q_N

Subset Construction

- Given an NFA N, the subset construction allows constructing a DFA D that accepts the same language as N
- Let $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ be an NFA
- Let $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ be a DFA where:
  - The input alphabets are the same
  - $Q_D = 2^{Q_N}$, i.e., states of Q_D are sets of states (subsets) of Q_N
  - The initial state of D is a singleton consisting of the initial state of N

Subset Construction

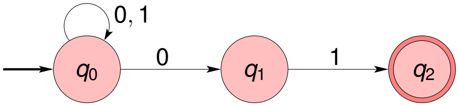
- Given an NFA N, the subset construction allows constructing a DFA D that accepts the same language as N
- Let $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ be an NFA
- Let $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ be a DFA where:
  - The input alphabets are the same
  - $Q_D = 2^{Q_N}$, i.e., states of Q_D are sets of states (subsets) of Q_N
  - The initial state of D is a singleton consisting of the initial state of N
  - F_D is the set of $S \subseteq Q_N$ such that $S \cap F_N \neq \emptyset$ (S is an accepting state of D if it contains an accepting state of N)

Subset Construction

- Given an NFA N, the subset construction allows constructing a DFA D that accepts the same language as N
- Let $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ be an NFA
- Let $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ be a DFA where:
  - The input alphabets are the same
  - $Q_D = 2^{Q_N}$, i.e., states of Q_D are sets of states (subsets) of Q_N
  - The initial state of D is a singleton consisting of the initial state of N
  - F_D is the set of $S \subseteq Q_N$ such that $S \cap F_N \neq \emptyset$ (S is an accepting state of D if it contains an accepting state of N)
  - For each $S \subseteq Q_N$ and for each $a \in \Sigma$,

$$\delta_D(S, a) = \bigcup_{q \in S} \delta_N(q, a)$$

For example, if N is

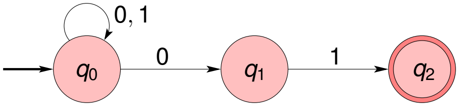


then the transition table of D is as follows:

	0	1
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$

Subset Construction

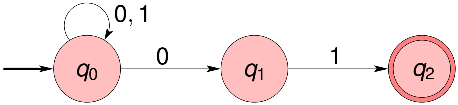
For example, if N is



then the transition table of D is as follows:

	0	1
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

For example, if N is

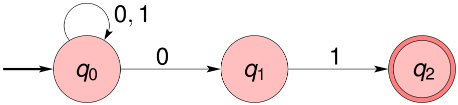


then the transition table of D is as follows:

	0	1
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

Subset Construction

For example, if N is



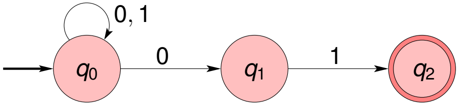
then the transition table of D is as follows:

	0	1
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

States of D that are unreachable from $\{q_0\}$ can be ignored, so the number of states of D is usually smaller than $2^{|Q_N|}$

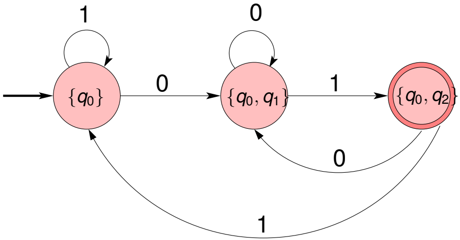
- Procedure for constructing the transition table
  1. Add transitions from $\{q_0\}$ to the table
  2. For each new state S, add transitions from S to the table

For example, if N is



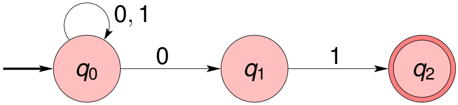
then the transition table of D is as follows:

	0	1
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$



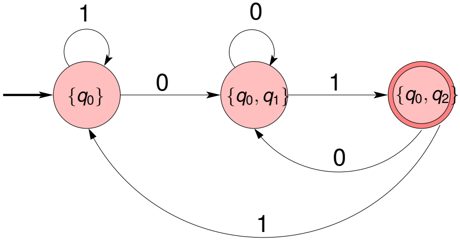
Subset Construction

For example, if N is



then the transition table of D is as follows:

	0	1
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

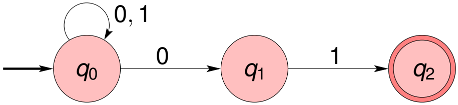


This is a proper DFA!

Though entries in table are sets, states of the automaton **are** sets

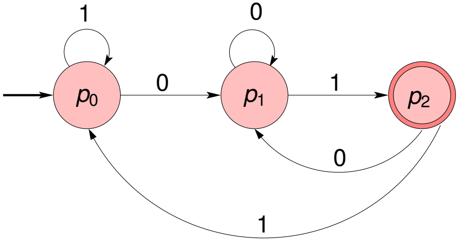
Subset Construction

For example, if N is



then the transition table of D is as follows:

	0	1
p_0	p_1	p_0
p_1	p_1	p_2
p_2	p_1	p_0



Finite Automata with λ-Transitions

- Next we will give yet another extension of finite automata
- Now an NFA will be allowed to make a transition spontaneously, without consuming any input symbol
- These transitions are called λ-transitions, as λ stands for the empty word
- They do not expand the class of languages accepted by finite automata, but they do give us some added “programming convenience”

Finite Automata with λ-Transitions

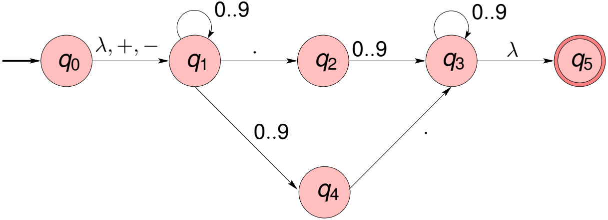
- View the automaton as accepting the sequences of labels along paths from the start state to an accepting state

Finite Automata with λ-Transitions

- View the automaton as accepting the sequences of labels along paths from the start state to an accepting state

  But each λ is invisible: it contributes nothing to the word along the path

- E.g., the following automaton



  accepts decimal numbers consisting of:

  1. An optional $+$ or $-$ sign,
  2. A string of digits,
  3. A decimal point, and
  4. Another string of digits.

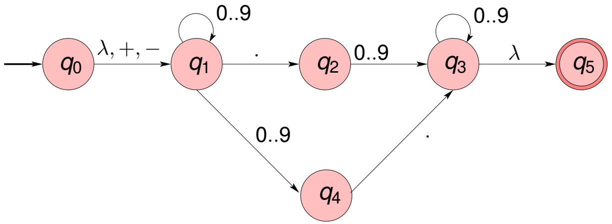
  Strings (2) or (4) can be empty, but at least one is not

Finite Automata with λ-Transitions

- A λ-nondeterministic finite automaton (λ-NFA) consists of:
  - Q, a finite set of states
  - Σ, an alphabet (called input alphabet)
  - δ, a transition function from $Q \times (\Sigma \cup \{\lambda\})$ to 2^Q
  - $q_0 \in Q$, called the initial state
  - $F \subseteq Q$, called the set of final or accepting states

Finite Automata with λ-Transitions

The transition table of



is

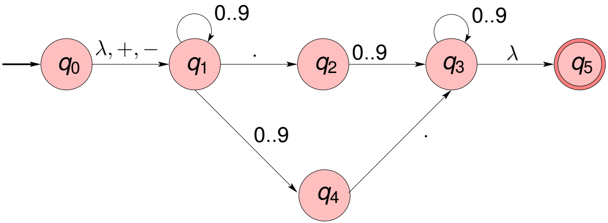
	λ	$+,-$	.	0..9
q_0	$\{q_1\}$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\emptyset$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_3	$\{q_5\}$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_4	$\emptyset$	$\emptyset$	$\{q_3\}$	$\emptyset$
q_5	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Finite Automata with λ-Transitions

- We define the λ-closure of a state q, denoted $\Lambda(q)$, as the set of states reachable from q along paths made only of λ-trans.

Finite Automata with λ-Transitions

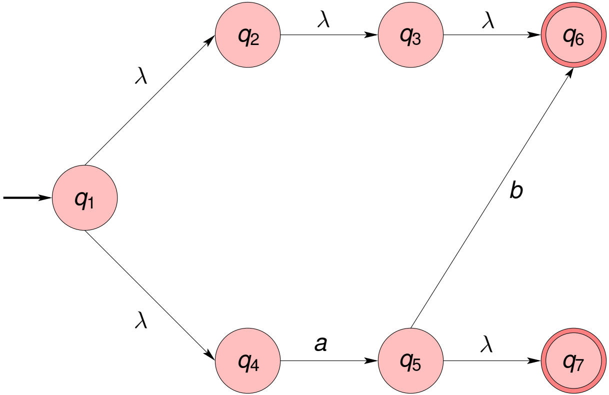
- We define the λ-closure of a state q, denoted $\Lambda(q)$, as the set of states reachable from q along paths made only of λ-trans.
- E.g., for



each state is its own λ-closure, except for q_0 and q_3:

- $\Lambda(q_0) = \{q_0, q_1\}$
- $\Lambda(q_3) = \{q_3, q_5\}$

Finite Automata with λ-Transitions



- $\Lambda(q_1) = \{q_1, q_2, q_3, q_4, q_6\}$
- $\Lambda(q_2) = \{q_2, q_3, q_6\}$
- $\Lambda(q_4) = \{q_4\}$
- $\ldots$

Finite Automata with λ-Transitions

- As usual we will extend transition functions from symbols to words
- If q is a state and ω is a word,
  then $\hat{\delta}(q, \omega)$ will be the states we can reach from q after reading word ω
  but taking into account that λ-transitions do not consume symbols

Finite Automata with λ-Transitions

- As usual we will extend transition functions from symbols to words
- If q is a state and ω is a word, then $\hat{\delta}(q,\omega)$ will be the states we can reach from q after reading word ω but taking into account that λ-transitions do not consume symbols
- Given $\delta : Q \times (\Sigma \cup \{\lambda\}) \to 2^Q$, we extend it to $\hat{\delta} : Q \times \Sigma^* \to 2^Q$:
  - for any $q \in Q$, $\hat{\delta}(q,\lambda) = \Lambda(q)$
  - for any $q \in Q$ and word of the form $a\,\omega$,

$$\hat{\delta}(q, a\,\omega) = \bigcup_{p \in \Lambda(q)} \bigcup_{r \in \delta(p,a)} \hat{\delta}(r,\omega)$$

Finite Automata with λ-Transitions

- As usual we will extend transition functions from symbols to words

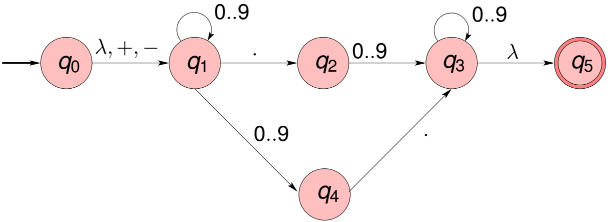
- If q is a state and ω is a word,
  then $\hat{\delta}(q, \omega)$ will be the states we can reach from q after reading word ω
  but taking into account that λ-transitions do not consume symbols

- Given $\delta : Q \times (\Sigma \cup \{\lambda\}) \to 2^Q$, we extend it to $\hat{\delta} : Q \times \Sigma^* \to 2^Q$:
  - for any $q \in Q$, $\hat{\delta}(q, \lambda) = \Lambda(q)$
  - for any $q \in Q$ and word of the form $a\,\omega$,

$$\hat{\delta}(q, a\,\omega) = \bigcup_{p \in \Lambda(q)} \bigcup_{r \in \delta(p,a)} \hat{\delta}(r, \omega)$$

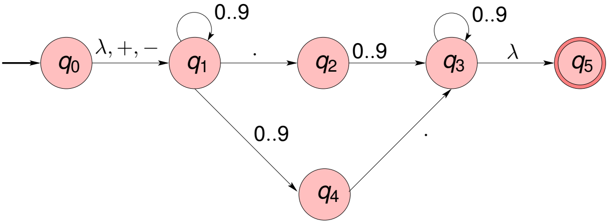
- Note the difference between
  $\delta(q, a)$ (the states we can move after reading symbol a in one transition)
  $\hat{\delta}(q, a)$ (allowing λ-transitions before/after reading symbol a)

Finite Automata with λ-Transitions



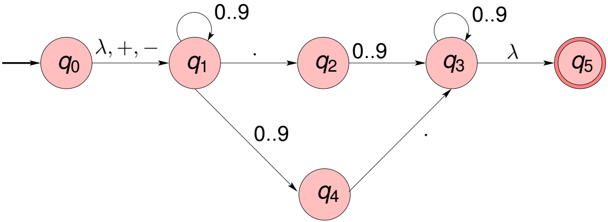
- $\hat{\delta}(q_0, 5.6)$

Finite Automata with λ-Transitions

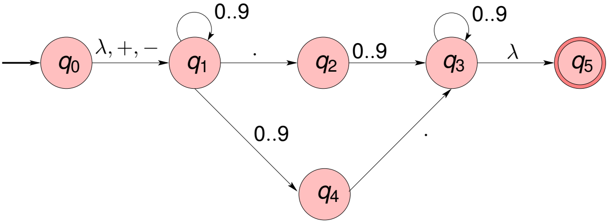


- $\hat{\delta}(q_0, 5.6)$
  - $\Lambda(q_0) = \{q_0, q_1\}$

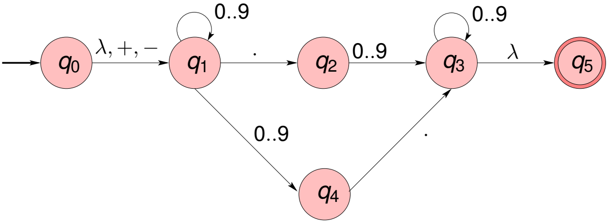
Finite Automata with λ-Transitions



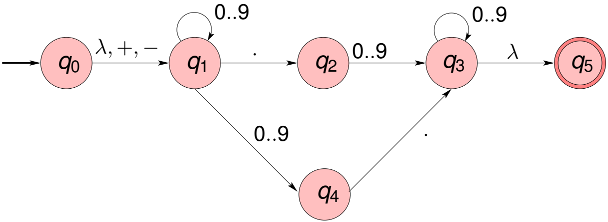
- $\hat{\delta}(q_0, 5.6)$
  - $\Lambda(q_0) = \{q_0, q_1\}$
  - $\delta(q_0, 5) = \emptyset$



- $\hat{\delta}(q_0, 5.6)$
  - $\Lambda(q_0) = \{q_0, q_1\}$
  - $\delta(q_0, 5) = \emptyset$
  - $\delta(q_1, 5) = \{q_1, q_4\}$

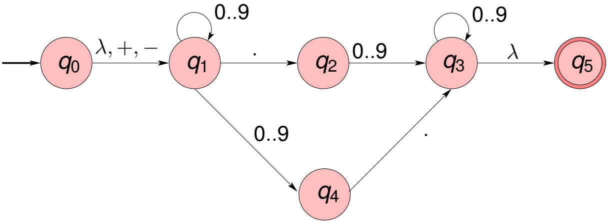


- $\hat{\delta}(q_0, 5.6) = \hat{\delta}(q_1, .6) \cup \hat{\delta}(q_4, .6)$
  - $\Lambda(q_0) = \{q_0, q_1\}$
  - $\delta(q_0, 5) = \emptyset$
  - $\delta(q_1, 5) = \{q_1, q_4\}$



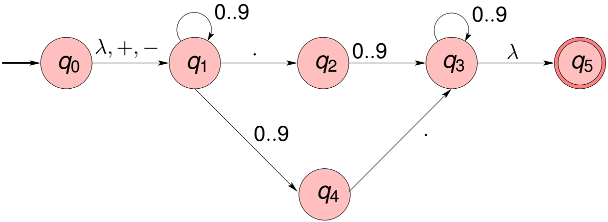
- $\hat{\delta}(q_0, 5.6) = \hat{\delta}(q_1, .6) \cup \hat{\delta}(q_4, .6)$

Finite Automata with λ-Transitions

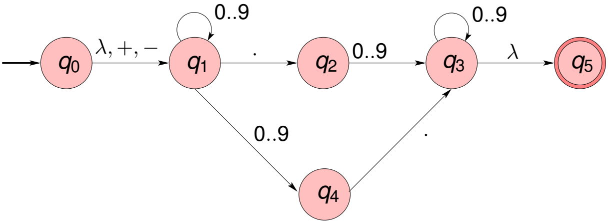


- $\hat{\delta}(q_0, 5.6) = \hat{\delta}(q_1, .6) \cup \hat{\delta}(q_4, .6)$
  - $\Lambda(q_1) = \{q_1\}$

Finite Automata with λ-Transitions

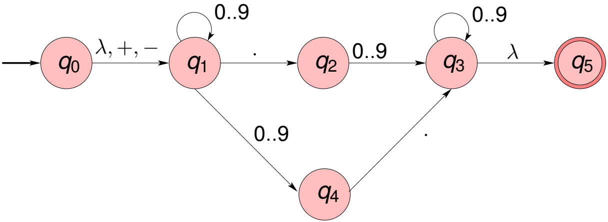


- $\hat{\delta}(q_0, 5.6) = \hat{\delta}(q_1, .6) \cup \hat{\delta}(q_4, .6)$
  - $\Lambda(q_1) = \{q_1\}$
  - $\delta(q_1, .) = \{q_2\}$



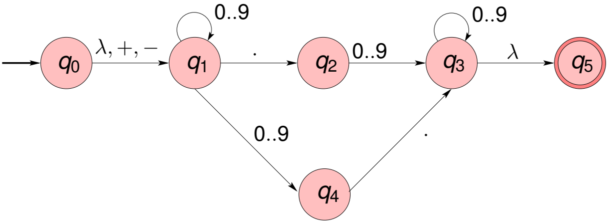
- $\hat{\delta}(q_0, 5.6) = \hat{\delta}(q_2, 6) \cup \hat{\delta}(q_4, .6)$
  - $\Lambda(q_1) = \{q_1\}$
  - $\delta(q_1, .) = \{q_2\}$

Finite Automata with λ-Transitions

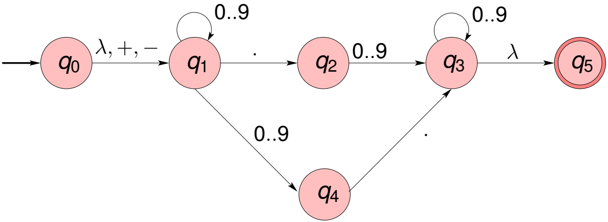


- $\hat{\delta}(q_0, 5.6) = \hat{\delta}(q_2, 6) \cup \hat{\delta}(q_4, .6)$

Finite Automata with λ-Transitions

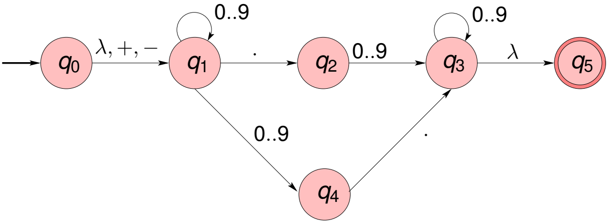


- $\hat{\delta}(q_0, 5.6) = \hat{\delta}(q_2, 6) \cup \hat{\delta}(q_4, .6)$
  - $\Lambda(q_4) = \{q_4\}$



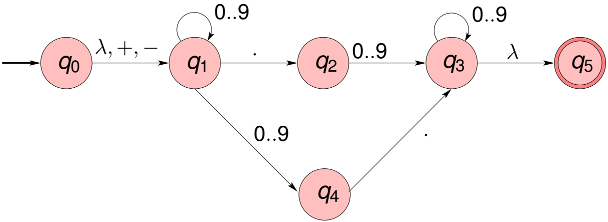
- $\hat{\delta}(q_0, 5.6) = \hat{\delta}(q_2, 6) \cup \hat{\delta}(q_4, .6)$
  - $\Lambda(q_4) = \{q_4\}$
  - $\delta(q_4, .) = \{q_3\}$

Finite Automata with λ-Transitions



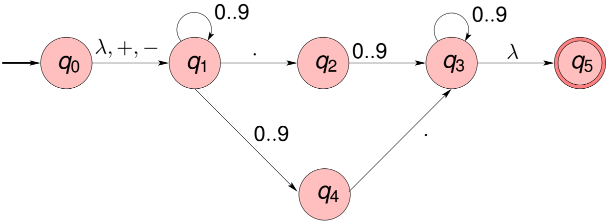
- $\hat{\delta}(q_0, 5.6) = \hat{\delta}(q_2, 6) \cup \hat{\delta}(q_3, 6)$
  - $\Lambda(q_4) = \{q_4\}$
  - $\delta(q_4, .) = \{q_3\}$

Finite Automata with λ-Transitions

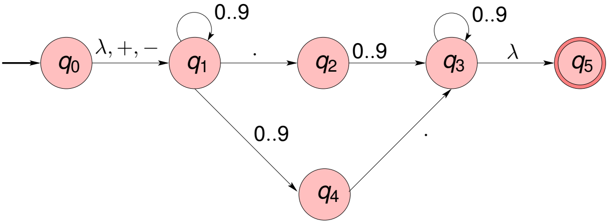


- $\hat{\delta}(q_0, 5.6) = \hat{\delta}(q_2, 6) \cup \hat{\delta}(q_3, 6)$

Finite Automata with λ-Transitions

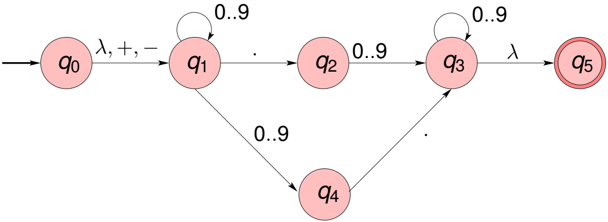


- $\hat{\delta}(q_0, 5.6) = \hat{\delta}(q_2, 6) \cup \hat{\delta}(q_3, 6)$
    - $\Lambda(q_2) = \{q_2\}$



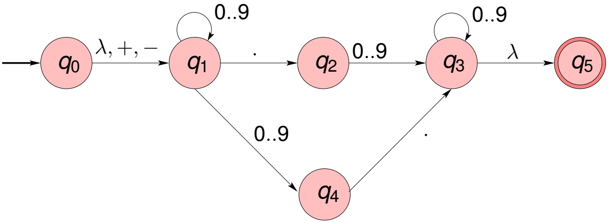
- $\hat{\delta}(q_0, 5.6) = \hat{\delta}(q_2, 6) \cup \hat{\delta}(q_3, 6)$
  - $\Lambda(q_2) = \{q_2\}$
  - $\delta(q_2, 6) = \{q_3\}$

Finite Automata with λ-Transitions



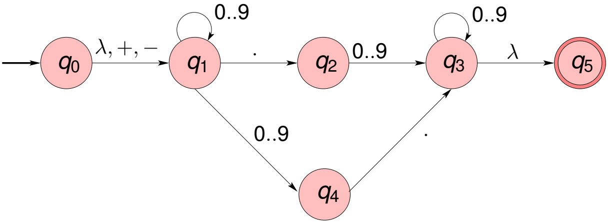
- $\hat{\delta}(q_0, 5.6) = \hat{\delta}(q_3, \lambda) \cup \hat{\delta}(q_3, 6)$
  - $\Lambda(q_2) = \{q_2\}$
  - $\delta(q_2, 6) = \{q_3\}$

Finite Automata with λ-Transitions



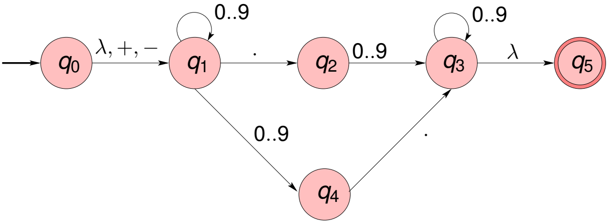
- $\hat{\delta}(q_0, 5.6) = \hat{\delta}(q_3, \lambda) \cup \hat{\delta}(q_3, 6)$

Finite Automata with λ-Transitions



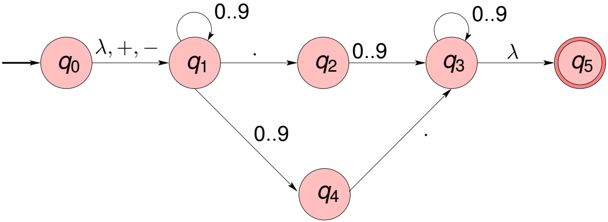
- $\hat{\delta}(q_0, 5.6) = \hat{\delta}(q_3, \lambda) \cup \hat{\delta}(q_3, 6)$
  - $\Lambda(q_3) = \{q_3, q_5\}$

Finite Automata with λ-Transitions



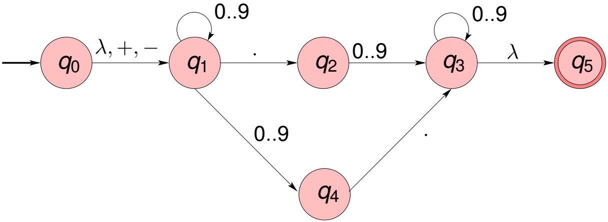
- $\hat{\delta}(q_0, 5.6) = \hat{\delta}(q_3, \lambda) \cup \hat{\delta}(q_3, 6)$
  - $\Lambda(q_3) = \{q_3, q_5\}$
  - $\delta(q_3, 6) = \{q_3\}$

Finite Automata with λ-Transitions

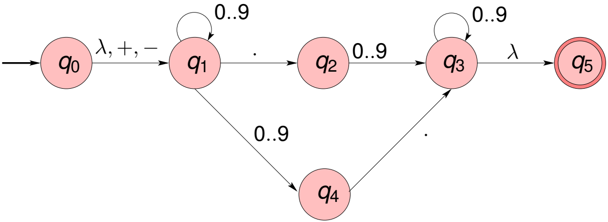


- $\hat{\delta}(q_0, 5.6) = \hat{\delta}(q_3, \lambda) \cup \hat{\delta}(q_3, 6)$
    - $\Lambda(q_3) = \{q_3, q_5\}$
    - $\delta(q_3, 6) = \{q_3\}$
    - $\delta(q_5, 6) = \emptyset$

Finite Automata with λ-Transitions



- $\hat{\delta}(q_0, 5.6) = \hat{\delta}(q_3, \lambda)$
  - $\Lambda(q_3) = \{q_3, q_5\}$
  - $\delta(q_3, 6) = \{q_3\}$
  - $\delta(q_5, 6) = \emptyset$



- $\hat{\delta}(q_0, 5.6) = \hat{\delta}(q_3, \lambda) = \{q_3, q_5\}$

Finite Automata with λ-Transitions

- Given a λ-NFA $A = (Q, \Sigma, \delta, q_0, F)$, a word $\omega \in \Sigma^*$ is accepted by A if

$$\hat{\delta}(q_0, \omega) \cap F \neq \emptyset$$

  I.e., ω is accepted if $\hat{\delta}(q_0, \omega)$ contains at least one accepting state

Finite Automata with λ-Transitions

- Given a λ-NFA $A = (Q, \Sigma, \delta, q_0, F)$, a word $\omega \in \Sigma^*$ is accepted by A if

$$\hat{\delta}(q_0, \omega) \cap F \neq \emptyset$$

  I.e., ω is accepted if $\hat{\delta}(q_0, \omega)$ contains at least one accepting state

- The language of A, denoted $L(A)$, is the set of all words accepted by A

Finite Automata with λ-Transitions

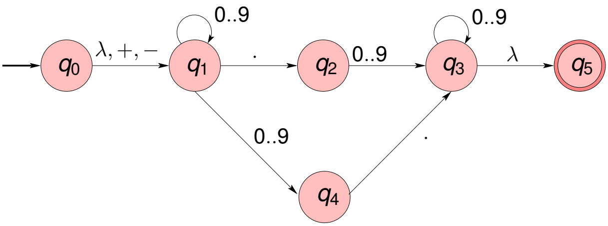
- Given a λ-NFA $A = (Q, \Sigma, \delta, q_0, F)$, a word $\omega \in \Sigma^*$ is accepted by A if

$$\hat{\delta}(q_0, \omega) \cap F \neq \emptyset$$

  I.e., ω is accepted if $\hat{\delta}(q_0, \omega)$ contains at least one accepting state

- The language of A, denoted $L(A)$, is the set of all words accepted by A

- For example, for the automaton A



  since $\hat{\delta}(q_0, 5.6) = \{q_3, q_5\}$ we conclude that $5.6 \in L(A)$

Eliminating λ-Transitions

- Clearly any NFA can be viewed as a λ-NFA (not using λ transitions)

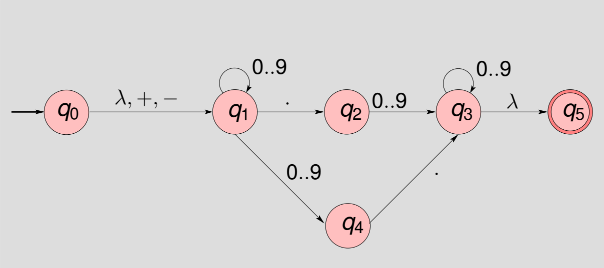
Eliminating λ-Transitions

- Clearly any NFA can be viewed as a λ-NFA (not using λ transitions)
- Next we will see for any λ-NFA there is an NFA with the same language

Eliminating λ-Transitions

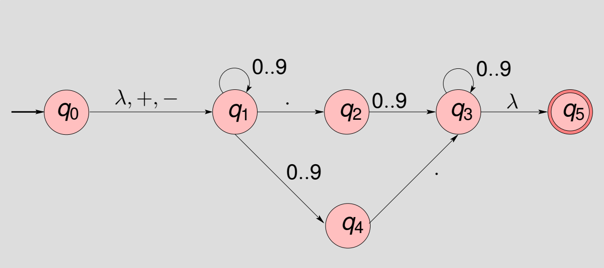
- Clearly any NFA can be viewed as a λ-NFA (not using λ transitions)
- Next we will see for any λ-NFA there is an NFA with the same language
- So the class of languages accepted by λ-NFA are the regular languages

Eliminating λ-Transitions



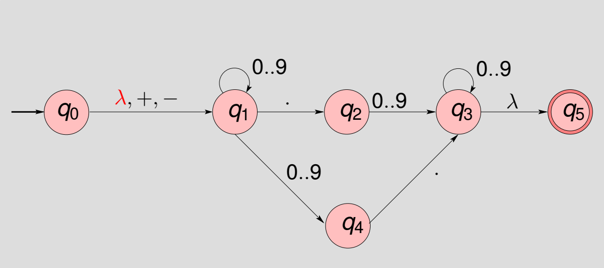
- Given a λ-NFA $A = (Q, \Sigma, \delta, q_0, F)$,
  to build an NFA with the same language we will eliminate λ-transitions

Eliminating λ-Transitions



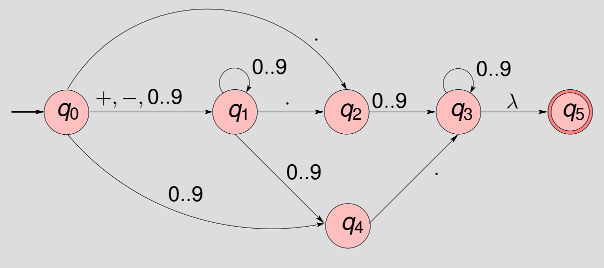
- Given a λ-NFA $A = (Q, \Sigma, \delta, q_0, F)$,
  to build an NFA with the same language we will eliminate λ-transitions
  - by possibly adding new non-λ transitions
    (a transition from p with symbol a to any state reachable following a path of λ-transitions from p ending with a transition with symbol a)

Eliminating λ-Transitions



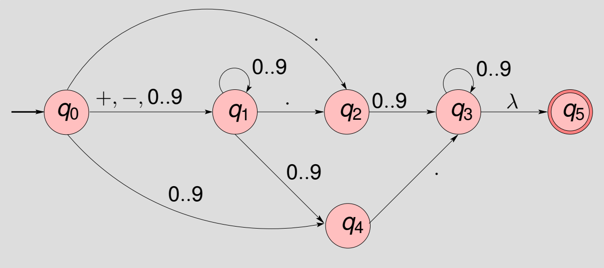
- Given a λ-NFA $A = (Q, \Sigma, \delta, q_0, F)$,
  to build an NFA with the same language we will eliminate λ-transitions
  - by possibly adding new non-λ transitions
    (a transition from p with symbol a to any state reachable following a path of λ-transitions from p ending with a transition with symbol a)

Eliminating λ-Transitions



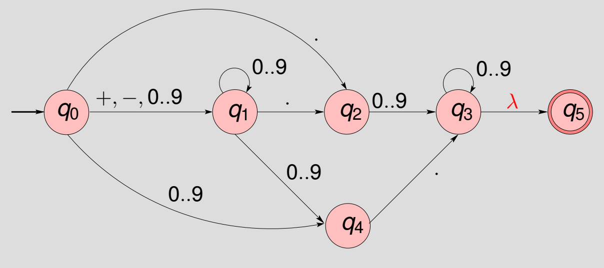
- Given a λ-NFA $A = (Q, \Sigma, \delta, q_0, F)$,
  to build an NFA with the same language we will eliminate λ-transitions
  - by possibly adding new non-λ transitions
    (a transition from p with symbol a to any state reachable following a path of λ-transitions from p ending with a transition with symbol a)

Eliminating λ-Transitions



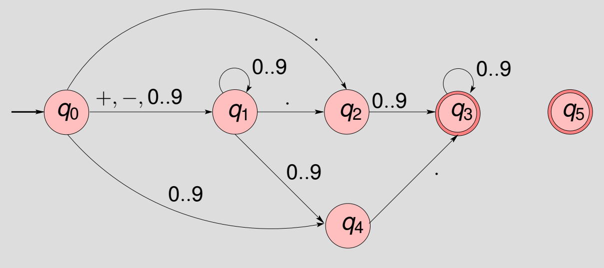
- Given a λ-NFA $A = (Q, \Sigma, \delta, q_0, F)$,
  to build an NFA with the same language we will eliminate λ-transitions
  - by possibly adding new non-λ transitions
    (a transition from p with symbol a to any state reachable following a path of λ-transitions from p ending with a transition with symbol a)
  - by possibly making some states accepting
    (those states from which an state from F can be reached following a path of λ-transitions)

Eliminating λ-Transitions

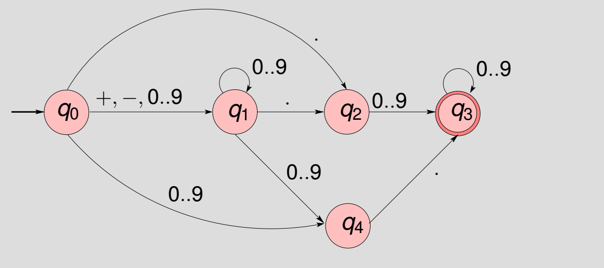


- Given a λ-NFA $A = (Q, \Sigma, \delta, q_0, F)$,
  to build an NFA with the same language we will eliminate λ-transitions
  - by possibly adding new non-λ transitions
    (a transition from p with symbol a to any state reachable following a path of λ-transitions from p ending with a transition with symbol a)
  - by possibly making some states accepting
    (those states from which an state from F can be reached following a path of λ-transitions)

Eliminating λ-Transitions



- Given a λ-NFA $A = (Q, \Sigma, \delta, q_0, F)$,
  to build an NFA with the same language we will eliminate λ-transitions
  - by possibly adding new non-λ transitions
    (a transition from p with symbol a to any state reachable following a path of λ-transitions from p ending with a transition with symbol a)
  - by possibly making some states accepting
    (those states from which an state from F can be reached following a path of λ-transitions)



- Given a λ-NFA $A = (Q, \Sigma, \delta, q_0, F)$,
  to build an NFA with the same language we will eliminate λ-transitions
  - by possibly adding new non-λ transitions
    (a transition from p with symbol a to any state reachable following a path of λ-transitions from p ending with a transition with symbol a)
  - by possibly making some states accepting
    (those states from which an state from F can be reached following a path of λ-transitions)

Eliminating λ-Transitions

- Let $A = (Q, \Sigma, \delta_A, q_0, F_A)$ be a λ-NFA
- Let $N = (Q, \Sigma, \delta_N, q_0, F_N)$ be an NFA where:

Eliminating λ-Transitions

- Let $A = (Q, \Sigma, \delta_A, q_0, F_A)$ be a λ-NFA
- Let $N = (Q, \Sigma, \delta_N, q_0, F_N)$ be an NFA where:
  - The input alphabets, the states and the initial states are the same

Eliminating λ-Transitions

- Let $A = (Q, \Sigma, \delta_A, q_0, F_A)$ be a λ-NFA
- Let $N = (Q, \Sigma, \delta_N, q_0, F_N)$ be an NFA where:
  - The input alphabets, the states and the initial states are the same
  - For each $S \in Q$ and for each $a \in \Sigma$,

$$\delta_N(p, a) = \bigcup_{q \in \Lambda(p)} \delta_A(q, a)$$

(*N* has a transition from *p* with symbol *a* to any state reachable in *A* following a path of λ-transitions from *p* ending in a transition with *a*)

Eliminating λ-Transitions

- Let $A = (Q, \Sigma, \delta_A, q_0, F_A)$ be a λ-NFA
- Let $N = (Q, \Sigma, \delta_N, q_0, F_N)$ be an NFA where:
  - The input alphabets, the states and the initial states are the same
  - For each $S \in Q$ and for each $a \in \Sigma$,

    $$\delta_N(p, a) = \bigcup_{q \in \Lambda(p)} \delta_A(q, a)$$

    (N has a transition from p with symbol a to any state reachable in A following a path of λ-transitions from p ending in a transition with a)
  - F_N is the set of $q \subseteq Q$ such that $\Lambda(q) \cap F_A \neq \emptyset$

    (accepting states of N are those from which an accepting state of A can be reached following a path of λ-transitions)

Eliminating λ-Transitions

- Let $A = (Q, \Sigma, \delta_A, q_0, F_A)$ be a λ-NFA
- Let $N = (Q, \Sigma, \delta_N, q_0, F_N)$ be an NFA where:
  - The input alphabets, the states and the initial states are the same
  - For each $S \in Q$ and for each $a \in \Sigma$,

    $$\delta_N(p, a) = \bigcup_{q \in \Lambda(p)} \delta_A(q, a)$$

    (N has a transition from p with symbol a to any state reachable in A following a path of λ-transitions from p ending in a transition with a)
  - F_N is the set of $q \subseteq Q$ such that $\Lambda(q) \cap F_A \neq \emptyset$

    (accepting states of N are those from which an accepting state of A can be reached following a path of λ-transitions)
- Both automata accept the same language: $L(A) = L(N)$

Regular Expressions

- Finite automata can be viewed as mechanisms for defining languages
- Regular expressions (regexps) are another language-defining notation
- Both describe exactly the same class of languages: regular languages
- So why do we need regexps?

Regular Expressions

- Finite automata can be viewed as mechanisms for defining languages
- Regular expressions (regexps) are another language-defining notation
- Both describe exactly the same class of languages: regular languages
- So why do we need regexps?
- Regexps provide a declarative way to express the words to be accepted
- They allow a higher level of abstraction and so are easier to work with

Regular Expressions

- Finite automata can be viewed as mechanisms for defining languages
- Regular expressions (regexps) are another language-defining notation
- Both describe exactly the same class of languages: regular languages
- So why do we need regexps?
- Regexps provide a declarative way to express the words to be accepted
- They allow a higher level of abstraction and so are easier to work with
- Typical workflow when searching a text for a pattern (e.g., in `grep`):
  1. Specify the pattern with a regexp
  2. Translate the regexp into an equivalent DFA (e.g., using `lex`)
  3. Run the DFA

Regular Expressions

- Finite automata can be viewed as mechanisms for defining languages
- Regular expressions (regexps) are another language-defining notation
- Both describe exactly the same class of languages: regular languages
- So why do we need regexps?
- Regexps provide a declarative way to express the words to be accepted
- They allow a higher level of abstraction and so are easier to work with
- Typical workflow when searching a text for a pattern (e.g., in `grep`):
  1. Specify the pattern with a regexp
  2. Translate the regexp into an equivalent DFA (e.g., using `lex`)
  3. Run the DFA
- Next we will see that regexps can be easily translated into λ-NFA's
  (which in turn can be translated into NFA's,
  which in turn can be translated into DFA's)

Regular Expressions

- For example:

```
[a-z][a-z]*(\.[a-z][a-z]*)*@(est\.fib|estudiant)\.upc\.edu
```

is a regexp for recognizing the emails of students at AP3

(written in Linux *extended regular expression notation*)

  - `[a-z]` represents any character `a`, `b`, `c`, ..., `x`, `y`, `z`
  - `*` represents repetition (zero, one, two, ... times)
  - `\.` represents the character dot `.`
  - `|` represents alternative

Regular Expressions

- For example:

`[a-z][a-z]*(\.[a-z][a-z]*)*@(est\.fib|estudiant)\.upc\.edu`

  is a regexp for recognizing the emails of students at AP3

  (written in Linux *extended regular expression notation*)

  - `[a-z]` represents any character `a`, `b`, `c`, ..., `x`, `y`, `z`
  - `*` represents repetition (zero, one, two, ... times)
  - `\.` represents the character dot `.`
  - `|` represents alternative

- Verbally the described pattern is:
  "a string of one or more letters followed by any number of strings whose first character is a dot and then one or more letters, followed by `@`, followed by either `est.fib` or `estudiant`, and then `.upc.edu`"

Regular Expressions

- For example:

```
[a-z][a-z]*(\.[a-z][a-z]*)*@(est\.fib|estudiant)\.upc\.edu
```

  is a regexp for recognizing the emails of students at AP3

  (written in Linux *extended regular expression notation*)

  - `[a-z]` represents any character `a`, `b`, `c`, ..., `x`, `y`, `z`
  - `*` represents repetition (zero, one, two, ... times)
  - `\.` represents the character dot `.`
  - `|` represents alternative

- Verbally the described pattern is:
  "a string of one or more letters followed by any number of strings whose first character is a dot and then one or more letters, followed by `@`, followed by either `est.fib` or `estudiant`, and then `.upc.edu`"

- To find the lines of file `p.txt` containing an email, using `grep`:

```
grep -E [a-z][a-z]*(\.[a-z][a-z]*)*@(est\.fib|estudiant)\.upc\.edu p.txt
```

Regular Expressions

- Before defining regexps, we'll introduce some operations on languages

Regular Expressions

- Before defining regexps, we'll introduce some operations on languages
- The union of two languages L and M is $L \cup M = \{\omega \mid \omega \in L \text{ or } \omega \in M\}$
- It is the set of strings that are either in L or M, or both
- If $L = \{00, 01\}$, $M = \{10, 11\}$, then $L \cup M$ are the 0-1 strings of length 2

Regular Expressions

- Before defining regexps, we'll introduce some operations on languages

- The union of two languages L and M is $L \cup M = \{\omega \mid \omega \in L \text{ or } \omega \in M\}$
- It is the set of strings that are either in L or M, or both
- If $L = \{00, 01\}$, $M = \{10, 11\}$, then $L \cup M$ are the 0-1 strings of length 2

- The concatenation of L and M is $LM = \{\omega_1\omega_2 \mid \omega_1 \in L, \omega_2 \in M\}$
- It is the set of strings that can be formed
  by taking any string in L and concatenating it with any string in M
- If $L = \{00, 01\}$, $M = \{10, 11\}$, then $LM = \{0010, 0011, 0110, 0111\}$

Regular Expressions

- Before defining regexps, we'll introduce some operations on languages

- The union of two languages L and M is $L \cup M = \{\omega \mid \omega \in L \text{ or } \omega \in M\}$
- It is the set of strings that are either in L or M, or both
- If $L = \{00, 01\}$, $M = \{10, 11\}$, then $L \cup M$ are the 0-1 strings of length 2

- The concatenation of L and M is $LM = \{\omega_1\omega_2 \mid \omega_1 \in L, \omega_2 \in M\}$
- It is the set of strings that can be formed
  by taking any string in L and concatenating it with any string in M
- If $L = \{00, 01\}$, $M = \{10, 11\}$, then $LM = \{0010, 0011, 0110, 0111\}$
- We write LL as L^2, LLL as L^3, etc.

Regular Expressions

- Before defining regexps, we'll introduce some operations on languages

- The union of two languages L and M is $L \cup M = \{\omega \mid \omega \in L \text{ or } \omega \in M\}$
- It is the set of strings that are either in L or M, or both
- If $L = \{00, 01\}$, $M = \{10, 11\}$, then $L \cup M$ are the 0-1 strings of length 2

- The concatenation of L and M is $LM = \{\omega_1\omega_2 \mid \omega_1 \in L, \omega_2 \in M\}$
- It is the set of strings that can be formed
  by taking any string in L and concatenating it with any string in M
- If $L = \{00, 01\}$, $M = \{10, 11\}$, then $LM = \{0010, 0011, 0110, 0111\}$
- We write LL as L^2, LLL as L^3, etc.

- The star (or Kleene closure) of L is $L^* = \{\lambda\} \cup L \cup L^2 \cup L^3 \cup \cdots = \bigcup_{i=0}^{\infty} L^i$
- It is the set of the strings that can be formed by taking any number of strings from L, possibly with repetitions, and concatenating them
- If $L = \{00, 01\}$, then $L^* = \{\lambda, 00, 01, 0000, 0100, 0001, 0101, \ldots\}$

Regular Expressions

- Regexps are built by applying a certain set of operators to elementary expressions and previously constructed expressions

Regular Expressions

- Regexps are built by applying a certain set of operators to elementary expressions and previously constructed expressions
- Given a regexp E, we will represent the language it accepts as $L(E)$

Regular Expressions

- Regexps are built by applying a certain set of operators to elementary expressions and previously constructed expressions
- Given a regexp E, we will represent the language it accepts as $L(E)$
- Given an alphabet Σ, regexps are defined recursively as follows:

Regular Expressions

- Regexps are built by applying a certain set of operators to elementary expressions and previously constructed expressions
- Given a regexp E, we will represent the language it accepts as $L(E)$
- Given an alphabet Σ, regexps are defined recursively as follows:
  - If $s \in \Sigma$, then s is a regexp, and $L(s) = \{s\}$

Regular Expressions

- Regexps are built by applying a certain set of operators to elementary expressions and previously constructed expressions
- Given a regexp E, we will represent the language it accepts as $L(E)$
- Given an alphabet Σ, regexps are defined recursively as follows:
  - If $s \in \Sigma$, then s is a regexp, and $L(s) = \{s\}$
  - $\emptyset$ is a regexp, and $L(\emptyset) = \emptyset$

Regular Expressions

- Regexps are built by applying a certain set of operators to elementary expressions and previously constructed expressions
- Given a regexp E, we will represent the language it accepts as $L(E)$
- Given an alphabet Σ, regexps are defined recursively as follows:
  - If $s \in \Sigma$, then s is a regexp, and $L(s) = \{s\}$
  - $\emptyset$ is a regexp, and $L(\emptyset) = \emptyset$
  - λ is a regexp, and $L(\lambda) = \{\lambda\}$

Regular Expressions

- Regexps are built by applying a certain set of operators to elementary expressions and previously constructed expressions
- Given a regexp E, we will represent the language it accepts as $L(E)$
- Given an alphabet Σ, regexps are defined recursively as follows:
  - If $s \in \Sigma$, then s is a regexp, and $L(s) = \{s\}$
  - $\emptyset$ is a regexp, and $L(\emptyset) = \emptyset$
  - λ is a regexp, and $L(\lambda) = \{\lambda\}$
  - If E and F are regexps, then $E + F$ is a regexp, and $L(E + F) = L(E) \cup L(F)$

Regular Expressions

- Regexps are built by applying a certain set of operators to elementary expressions and previously constructed expressions
- Given a regexp E, we will represent the language it accepts as $L(E)$
- Given an alphabet Σ, regexps are defined recursively as follows:
  - If $s \in \Sigma$, then s is a regexp, and $L(s) = \{s\}$
  - $\emptyset$ is a regexp, and $L(\emptyset) = \emptyset$
  - λ is a regexp, and $L(\lambda) = \{\lambda\}$
  - If E and F are regexps, then $E + F$ is a regexp, and $L(E + F) = L(E) \cup L(F)$
  - If E and F are regexps, then EF is a regexp, and $L(EF) = L(E)L(F)$

Regular Expressions

- Regexps are built by applying a certain set of operators to elementary expressions and previously constructed expressions
- Given a regexp E, we will represent the language it accepts as $L(E)$
- Given an alphabet Σ, regexps are defined recursively as follows:
  - If $s \in \Sigma$, then s is a regexp, and $L(s) = \{s\}$
  - $\emptyset$ is a regexp, and $L(\emptyset) = \emptyset$
  - λ is a regexp, and $L(\lambda) = \{\lambda\}$
  - If E and F are regexps, then $E + F$ is a regexp, and $L(E + F) = L(E) \cup L(F)$
  - If E and F are regexps, then EF is a regexp, and $L(EF) = L(E)L(F)$
  - If E is a regexp, then E^* is a regexp, and $L(E^*) = L(E)^*$

- Could you find a regular expression
  for the set of strings that consist of alternating 0's and 1's?

Regular Expressions

- Could you find a regular expression
  for the set of strings that consist of alternating 0's and 1's?
- A possible solution is $(01)^* + (10)^* + 0(10)^* + 1(01)^*$

- Could you find a regular expression
  for the set of strings that consist of alternating 0's and 1's?
- A possible solution is $(01)^* + (10)^* + 0(10)^* + 1(01)^*$
- Another solution is $(\lambda + 1)(01)^*(\lambda + 0)$

- Could you find a regular expression
  for the set of strings that consist of alternating 0's and 1's?
- A possible solution is $(01)^* + (10)^* + 0(10)^* + 1(01)^*$
- Another solution is $(\lambda + 1)(01)^*(\lambda + 0)$
- The precedences of operators are as follows:

Regular Expressions

- Could you find a regular expression
  for the set of strings that consist of alternating 0's and 1's?
- A possible solution is $(01)^* + (10)^* + 0(10)^* + 1(01)^*$
- Another solution is $(\lambda + 1)(01)^*(\lambda + 0)$

- The precedences of operators are as follows:
  1. The star operator is of highest precedence.
     It applies only to the smallest sequence of symbols to its left
     that is a well-formed regular expression

- Could you find a regular expression
  for the set of strings that consist of alternating 0's and 1's?
- A possible solution is $(01)^* + (10)^* + 0(10)^* + 1(01)^*$
- Another solution is $(\lambda + 1)(01)^*(\lambda + 0)$

- The precedences of operators are as follows:
  1. The star operator is of highest precedence.
     It applies only to the smallest sequence of symbols to its left
     that is a well-formed regular expression
  2. Next in precedence comes the concatenation operator.
     After grouping all stars to their operands,
     all expressions that are juxtaposed are grouped together

- Could you find a regular expression
  for the set of strings that consist of alternating 0's and 1's?
- A possible solution is $(01)^* + (10)^* + 0(10)^* + 1(01)^*$
- Another solution is $(\lambda + 1)(01)^*(\lambda + 0)$

- The precedences of operators are as follows:
  1. The star operator is of highest precedence.
     It applies only to the smallest sequence of symbols to its left
     that is a well-formed regular expression
  2. Next in precedence comes the concatenation operator.
     After grouping all stars to their operands,
     all expressions that are juxtaposed are grouped together
  3. Finally, all unions are grouped with their operands

Regular Expressions

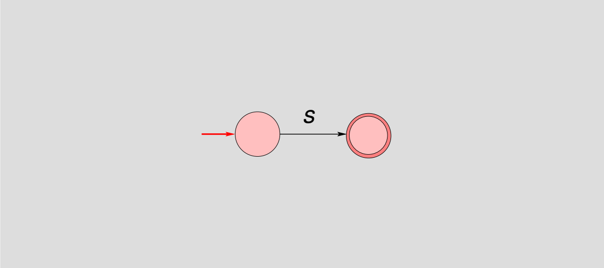
- Given a regexp R, let us find a λ-NFA A such that $L(A) = L(R)$

Regular Expressions

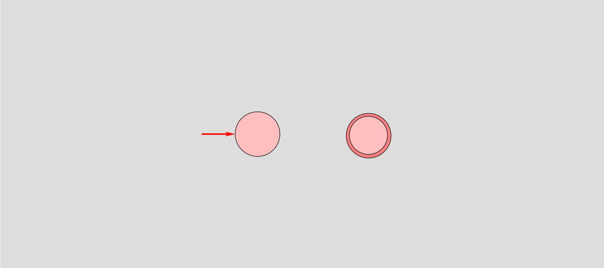
- Given a regexp R, let us find a λ-NFA A such that $L(A) = L(R)$
- All of the automata we construct will be λ-NFA's with
  - Exactly one accepting state
  - No arcs into the initial state
  - No arcs out of the accepting state

- Given a regexp R, let us find a λ-NFA A such that $L(A) = L(R)$
- All of the automata we construct will be λ-NFA's with
  - Exactly one accepting state
  - No arcs into the initial state
  - No arcs out of the accepting state
- Let us consider all possible cases according to the definition of regexps

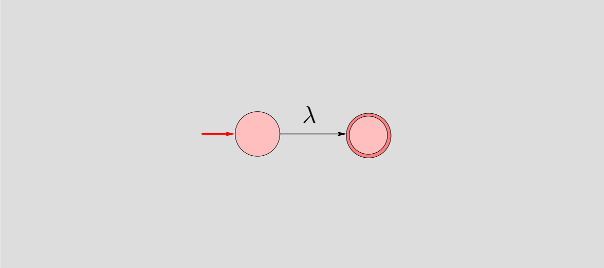
- Case $R = s$ for some symbol $s \in \Sigma$



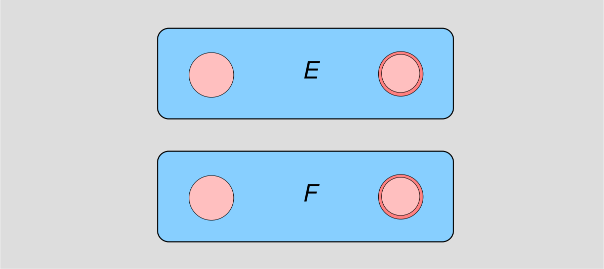
- Case $R = \emptyset$



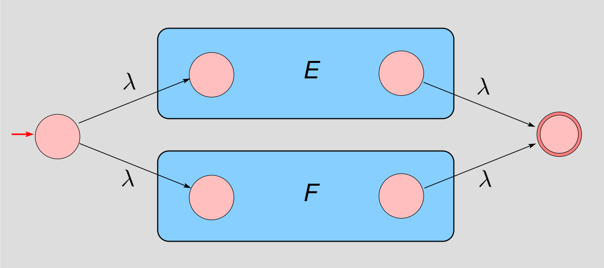
- Case $R = \lambda$



- Case $R = E + F$ for some regexps E, F

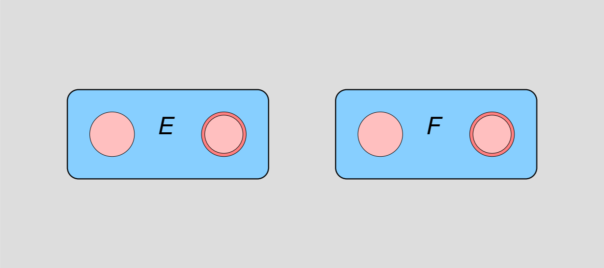


- Case $R = E + F$ for some regexps E, F

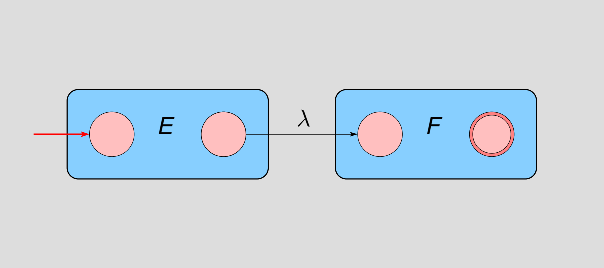


Regular Expressions

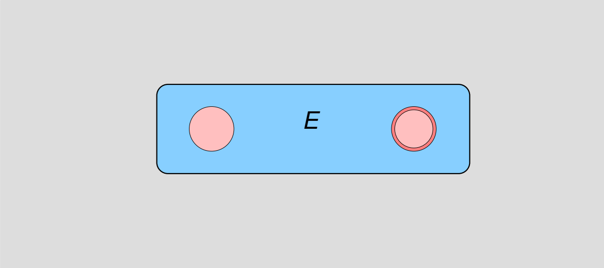
- Case $R = EF$ for some regexps E, F



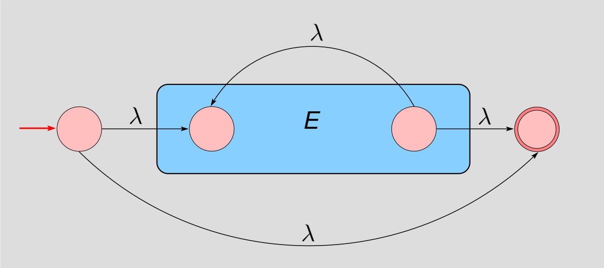
- Case $R = EF$ for some regexps E, F



- Case $R = E^*$ for some regexp E



- Case $R = E^*$ for some regexp E

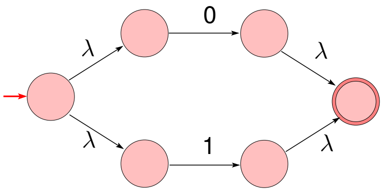


Regular Expressions

- Let us convert the regular expression $(0+1)^*1(0+1)$ into an λ-NFA

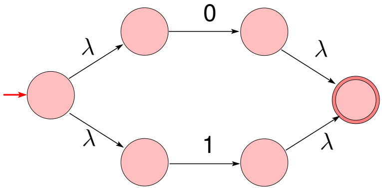
- Let us convert the regular expression $(0+1)^*1(0+1)$ into an λ-NFA
- First let us construct an automaton for $0+1$

- Let us convert the regular expression $(0+1)^*1(0+1)$ into an λ-NFA
- First let us construct an automaton for $0+1$



Regular Expressions

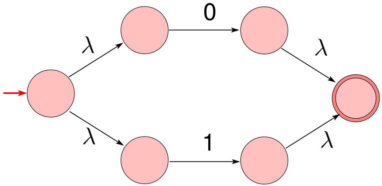
- Let us convert the regular expression $(0+1)^*1(0+1)$ into an λ-NFA
- First let us construct an automaton for $0+1$



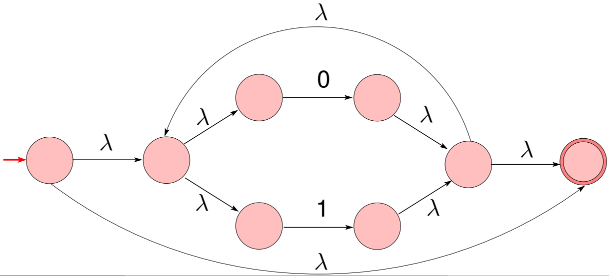
- Now let us construct an automaton for $(0+1)^*$

Regular Expressions

- Let us convert the regular expression $(0+1)^*1(0+1)$ into an λ-NFA
- First let us construct an automaton for $0+1$

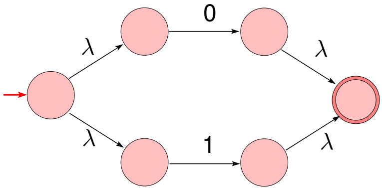


- Now let us construct an automaton for $(0+1)^*$



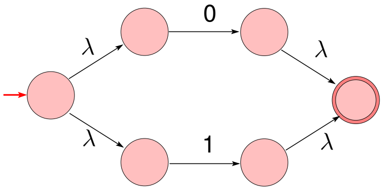
Regular Expressions

- Let us convert the regular expression $(0+1)^*1(0+1)$ into an λ-NFA
- First let us construct an automaton for $0+1$

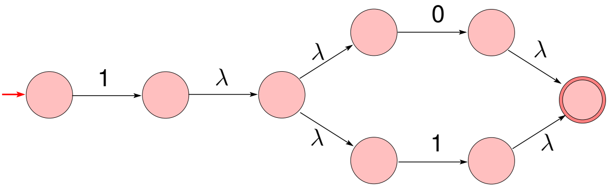


- And let us construct an automaton for $1(0+1)$

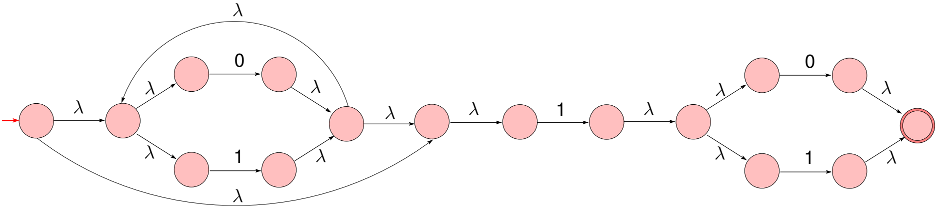
- Let us convert the regular expression $(0+1)^*1(0+1)$ into an λ-NFA
- First let us construct an automaton for $0+1$



- And let us construct an automaton for $1(0+1)$



- Let us convert the regular expression $(0+1)^*1(0+1)$ into an λ-NFA



Regular Expressions

In Linux extended regular expression (ERE) notation:

- Character classes represent large sets of characters succinctly
  - The symbol `.` (dot) stands for any character (except newline)
  - The sequence `[` $a_1 a_2 \ldots a_n$ `]` stands for regexp $a_1 + a_2 + ... + a_n$
  - Between `[]` we can put a range of the form $x - y$ to mean all the characters from x to y in the ASCII sequence
  - Special characters are scaped with a backslash, e.g. `\.` for dot

For example, `[A-Za-z0-9]` represents an alphanumeric character

Regular Expressions

In Linux extended regular expression (ERE) notation:

- Character classes represent large sets of characters succinctly
  - The symbol `.` (dot) stands for any character (except newline)
  - The sequence `[` $a_1 a_2 \ldots a_n$ `]` stands for regexp $a_1 + a_2 + ... + a_n$
  - Between `[]` we can put a range of the form $x - y$ to mean all the characters from x to y in the ASCII sequence
  - Special characters are scaped with a backslash, e.g. `\.` for dot

  For example, `[A-Za-z0-9]` represents an alphanumeric character
- Additional operators sometimes make it easier to express what we want
  - The operator `|` is used in place of $+$ to denote union
  - The operator `?` means *"zero or one of"*
  - The operator `+` means *"one or more of"*
  - The operator `{`n`}` means *"n copies of"*

  E.g., `[+-]?[0-9]+\.[0-9]{2}` are numbers with 2 decimal digits

Regular Expressions

- For a more extensive explanation of the ERE notation, type `man grep`

Regular Expressions

- For a more extensive explanation of the ERE notation, type `man grep`
- ERE notation (or similar with slight changes) is used:
  - In lexical-analyzer generators, such as `lex` or `flex`
  - In Linux tools for finding patterns in text, such as `grep` (short for `global search regular expression & print`)
  - In text editors, such as `emacs`, `sed`, ...

Minimization of DFA

- Given a DFA,
  is there another DFA accepting the same language with fewer states?

Minimization of DFA

- Given a DFA,
  is there another DFA accepting the same language with fewer states?
- Next: how to find an equivalent DFA with the minimum number of states

Testing Equivalence of States

- But first: when two distinct states p and q of a DFA can be replaced by a single state that behaves like both p and q?

Testing Equivalence of States

- But first: when two distinct states p and q of a DFA can be replaced by a single state that behaves like both p and q?
- We say that states p and q are equivalent if
  for all words ω, the state $\delta(p, \omega)$ is accepting iff $\delta(q, \omega)$ is accepting
- That is: we cannot tell the difference between states p and q by starting in one of the states and reading a word

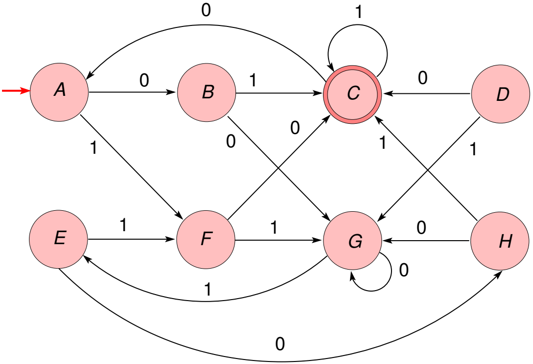
Testing Equivalence of States

- But first: when two distinct states p and q of a DFA can be replaced by a single state that behaves like both p and q?
- We say that states p and q are equivalent if for all words ω, the state $\delta(p, \omega)$ is accepting iff $\delta(q, \omega)$ is accepting
- That is: we cannot tell the difference between states p and q by starting in one of the states and reading a word
- Note we do not require that $\delta(p, \omega)$ and $\delta(q, \omega)$ are the same state, only that either both are accepting or both are nonaccepting

Testing Equivalence of States

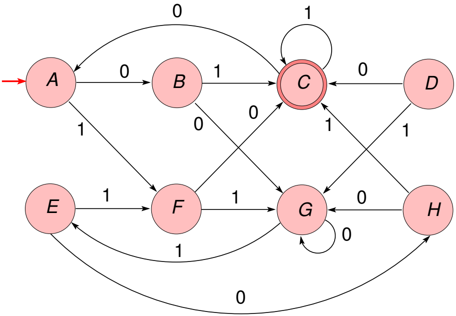
- But first: when two distinct states p and q of a DFA can be replaced by a single state that behaves like both p and q?
- We say that states p and q are equivalent if
  for all words ω, the state $\delta(p, \omega)$ is accepting iff $\delta(q, \omega)$ is accepting
- That is: we cannot tell the difference between states p and q
  by starting in one of the states and reading a word
- Note we do not require that $\delta(p, \omega)$ and $\delta(q, \omega)$ are the same state, only that either both are accepting or both are nonaccepting
- If two states are not equivalent, then we say they are distinguishable

Testing Equivalence of States



- C and H are distinguishable since one is accepting and the other is not
- So are E and F, as on input 0, E and F go to states C and H, resp.
- So are A and G, as on input 1, E and F go to states C and H, resp.

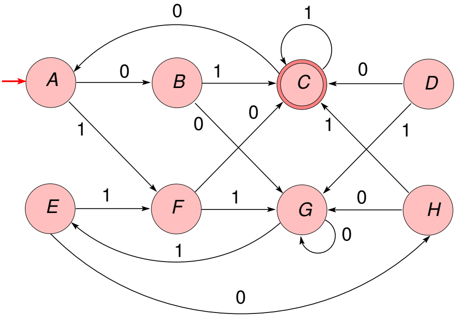
Testing Equivalence of States



Algorithm for testing equivalence of states

1. For every pair of states p and q such that one is accepting and the other is not, mark (p, q) as distinguishable
2. For every pair of states (p, q) and symbol a, if $\delta(p, a)$ and $\delta(q, a)$ are distinguishable then mark (p, q) as distinguishable
3. Repeat 2. till no new pairs of states are marked

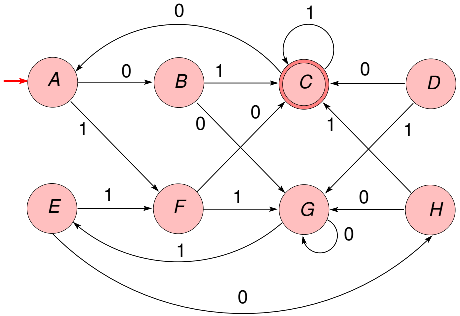
B							
C							
D							
E							
F							
G							
H							
	A	*B*	*C*	*D*	*E*	*F*	*G*



Algorithm for testing equivalence of states

1. For every pair of states p and q such that one is accepting and the other is not, mark (p, q) as distinguishable
2. For every pair of states (p, q) and symbol a, if $\delta(p, a)$ and $\delta(q, a)$ are distinguishable then mark (p, q) as distinguishable
3. Repeat 2. till no new pairs of states are marked

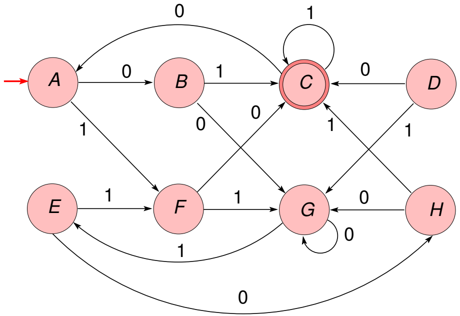
B							
C	λ	λ					
D			λ				
E			λ				
F			λ				
G			λ				
H			λ				
	A	B	C	D	E	F	G



Algorithm for testing equivalence of states

1. For every pair of states p and q such that one is accepting and the other is not, mark (p, q) as distinguishable
2. For every pair of states (p, q) and symbol a, if $\delta(p, a)$ and $\delta(q, a)$ are distinguishable then mark (p, q) as distinguishable
3. Repeat 2. till no new pairs of states are marked

	A	B	C	D	E	F	G
B	1						
C	λ	λ					
D	0	1	λ				
E		1	λ	0			
F	0	1	λ		0		
G		1	λ	0		0	
H	1		λ	0	1	0	1

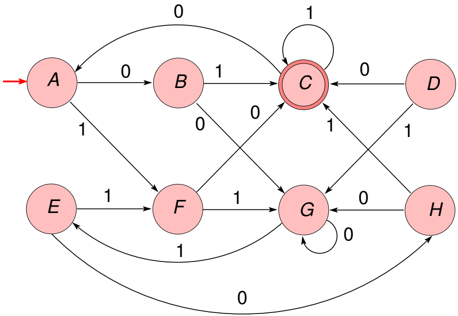


Algorithm for testing equivalence of states

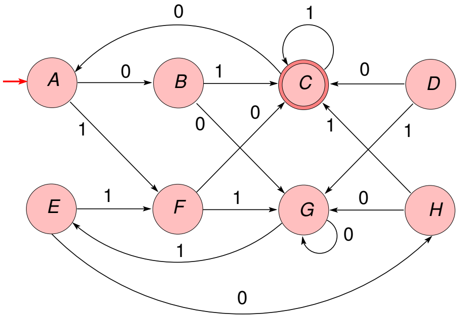
1. For every pair of states p and q such that one is accepting and the other is not, mark (p, q) as distinguishable
2. For every pair of states (p, q) and symbol a, if $\delta(p, a)$ and $\delta(q, a)$ are distinguishable then mark (p, q) as distinguishable
3. Repeat 2. till no new pairs of states are marked

	A	B	C	D	E	F	G
B	1						
C	λ	λ					
D	0	1	λ				
E		1	λ	0			
F	0	1	λ		0		
G	1	1	λ	0	1	0	
H	1		λ	0	1	0	1

Testing Equivalence of States

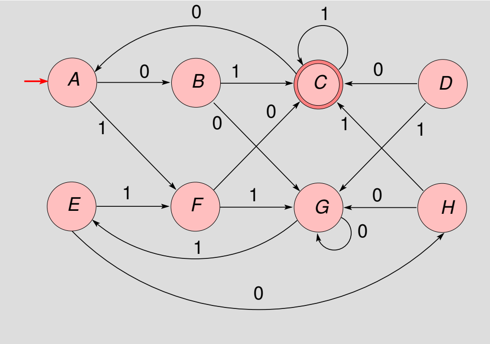


	A	B	C	D	E	F	G
B	1						
C	λ	λ					
D	0	1	λ				
E		1	λ	0			
F	0	1	λ		0		
G	1	1	λ	0	1	0	
H	1		λ	0	1	0	1



	A	B	C	D	E	F	G
B	1						
C	λ	λ					
D	0	1	λ				
E		1	λ	0			
F	0	1	λ		0		
G	1	1	λ	0	1	0	
H	1		λ	0	1	0	1

- We can partition the states into equivalence classes
  - A, E
  - B, H
  - D, F
  - C
  - G

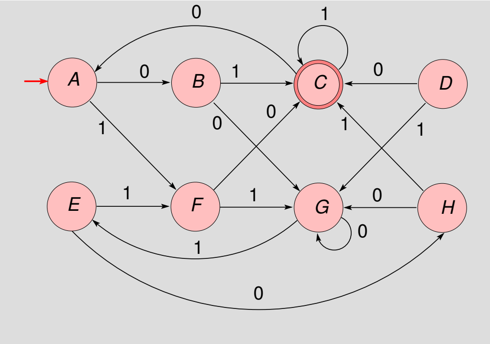


Equivalence classes

- A, E
- B, H
- D, F
- C
- G

Testing Equivalence of States

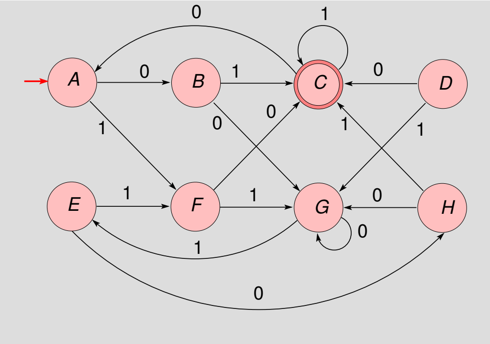
- We can consider the DFA were states are these equivalence classes



Equivalence classes

- A, E
- B, H
- D, F
- C
- G

- We can consider the DFA were states are these equivalence classes
- Let us merge the states in each equivalence class

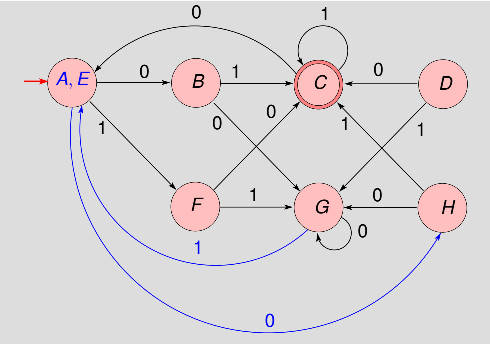


Equivalence classes

- A, E
- B, H
- D, F
- C
- G

Testing Equivalence of States

- We can consider the DFA were states are these equivalence classes
- Let us merge the states in each equivalence class

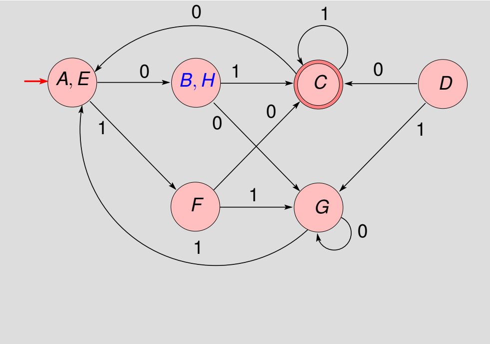


Equivalence classes

- A, E
- B, H
- D, F
- C
- G

Testing Equivalence of States

- We can consider the DFA were states are these equivalence classes
- Let us merge the states in each equivalence class

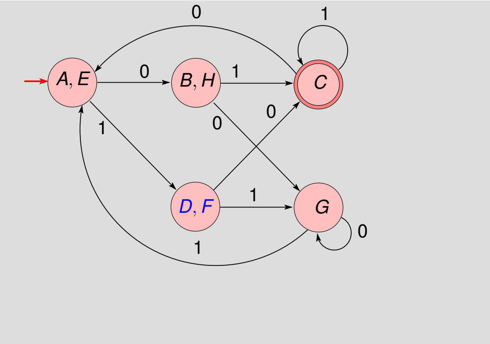


Equivalence classes

- A, E
- B, H
- D, F
- C
- G

Testing Equivalence of States

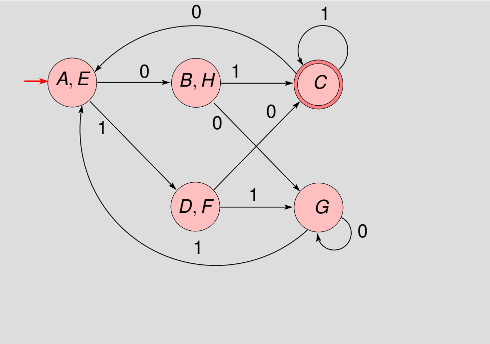
- We can consider the DFA were states are these equivalence classes
- Let us merge the states in each equivalence class



Equivalence classes

- A, E
- B, H
- D, F
- C
- G

- We can consider the DFA were states are these equivalence classes
- Let us merge the states in each equivalence class



Equivalence classes

- A, E
- B, H
- D, F
- C
- G

Quotient Automaton

- Given a DFA A, the quotient automaton of A merges equivalent states
- Let $A = (Q_A, \Sigma, \delta_A, q_0, F_A)$ be a DFA
- Let $M = (Q_M, \Sigma, \delta_M, C_0, F_M)$ be the DFA where:

Quotient Automaton

- Given a DFA A, the quotient automaton of A merges equivalent states
- Let $A = (Q_A, \Sigma, \delta_A, q_0, F_A)$ be a DFA
- Let $M = (Q_M, \Sigma, \delta_M, C_0, F_M)$ be the DFA where:
  - The input alphabets are the same

Quotient Automaton

- Given a DFA A, the quotient automaton of A merges equivalent states
- Let $A = (Q_A, \Sigma, \delta_A, q_0, F_A)$ be a DFA
- Let $M = (Q_M, \Sigma, \delta_M, C_0, F_M)$ be the DFA where:
  - The input alphabets are the same
  - Q_M are the equivalence classes of Q_A

Quotient Automaton

- Given a DFA A, the quotient automaton of A merges equivalent states
- Let $A = (Q_A, \Sigma, \delta_A, q_0, F_A)$ be a DFA
- Let $M = (Q_M, \Sigma, \delta_M, C_0, F_M)$ be the DFA where:
  - The input alphabets are the same
  - Q_M are the equivalence classes of Q_A
  - The initial state C_0 of M is the class of the initial state q_0 of A

Quotient Automaton

- Given a DFA A, the quotient automaton of A merges equivalent states
- Let $A = (Q_A, \Sigma, \delta_A, q_0, F_A)$ be a DFA
- Let $M = (Q_M, \Sigma, \delta_M, C_0, F_M)$ be the DFA where:
  - The input alphabets are the same
  - Q_M are the equivalence classes of Q_A
  - The initial state C_0 of M is the class of the initial state q_0 of A
  - F_M is the set of $C \in Q_M$ such that $C \cap F_A \neq \emptyset$
    (the classes of accepting states of A are the accepting states in M)

Quotient Automaton

- Given a DFA A, the quotient automaton of A merges equivalent states
- Let $A = (Q_A, \Sigma, \delta_A, q_0, F_A)$ be a DFA
- Let $M = (Q_M, \Sigma, \delta_M, C_0, F_M)$ be the DFA where:
  - The input alphabets are the same
  - Q_M are the equivalence classes of Q_A
  - The initial state C_0 of M is the class of the initial state q_0 of A
  - F_M is the set of $C \in Q_M$ such that $C \cap F_A \neq \emptyset$
    (the classes of accepting states of A are the accepting states in M)
  - For each $C \in Q_M$ and for each $a \in \Sigma$,
    $\delta_M(C, a)$ is the class of $\delta_A(q, a)$, with q any state whose class is C

Quotient Automaton

Properties of the quotient automaton:

- The quotient automaton M of a DFA A accepts the same language as A

Quotient Automaton

Properties of the quotient automaton:

- The quotient automaton M of a DFA A accepts the same language as A
- M is the equivalent automaton with the minimum number of states: if B is a DFA such that $L(B) = L(A)$ then $|Q_B| \geq |Q_M|$

Properties of the quotient automaton:

- The quotient automaton M of a DFA A accepts the same language as A
- M is the equivalent automaton with the minimum number of states:
  if B is a DFA such that $L(B) = L(A)$ then $|Q_B| \geq |Q_M|$
- There is a one-to-one correspondence
  between the states of any minimum-state equivalent automaton and M